MATHEMATICA®
TIME SERIES

A FULLY INTEGRATED ENVIRONMENT
FOR TIME-DEPENDENT DATA ANALYSIS

WOLFRAMRESEARCH
www.wolfram.com

Version 1.4

July 2007
First edition
Intended for use with *Mathematica* 6 or higher

Software and manual: Yu He, John Novak, Darren Glosemeyer
Product manager: Nirmal Malapaka
Project manager: Nirmal Malapaka
Editor: Jan Progen
Software quality assurance: Cindie Strater
Document quality assurance: Rebecca Bigelow and Jan Projen
Graphic design: Jeremy Davis and Megan Gillette

Published by Wolfram Research, Inc., 100 Trade Center Drive, Champaign, Illinois 61820-7237, USA
phone: +1-217-398-0700; fax: +1-217-398-0747; email: info@wolfram.com; web: www.wolfram.com

Table of Contents

Getting Started

About *Time Series*

Time Series is designed specifically to study and analyze linear time series, both univariate and multivariate, using *Mathematica*. It consists of this documentation, one *Mathematica* package file, and data files.

Mathematica package files are collections of programs written in the *Mathematica* language, so *Time Series* can only be used in conjunction with *Mathematica*. The *Mathematica* package file provided with *Time Series* is `TimeSeries.m`. It contains many of the functions and utilities necessary for time series analysis. `MovingAverage`, `MovingMedian` and `ExponentialMovingAverage`, commonly used for smoothing data, are included in *Mathematica*.

The primary purpose of the manual is to introduce and illustrate how to use the functions contained in the package. Part 1, User's Guide to *Time Series*, serves as a more detailed guide to the time series subject. Relevant concepts, methods, and formulas of linear time series analysis as well as more detailed examples are presented so as to make the whole *Time Series* as self-contained as possible. It is hoped that *Time Series* can serve as both an instructional resource and a practical tool so it can be used for pedagogical purposes as well as for analysis of real data. For those who want to pursue the detailed derivations and assumptions of different techniques, appropriate references to standard literature are given at the end of the manual. Part 2, Summary of *Time Series* Functions, summarizes the *Mathematica* functions provided by `TimeSeries.m` . It gives the definitions of the functions and examples illustrating their usage. Only those formulas that help define terms and notations are included. This concise summary is meant to be a quick and handy reference for the more advanced user or one familiar with the application package.

The organization of Part 1 is as follows. We introduce the commonly used stationary time series models and the basic theoretical quantities such as covariance and correlation functions in Section 1.2. Nonstationary and seasonal models are discussed in Section 1.3. Various elementary functions that check for stationarity and invertibility and compute correlations both in the univariate and multivariate cases are described in these two sections. A variety of transformations including linear filtering, simple exponential smoothing, and the Box-Cox transformation, which prepare data for modeling, are presented in Section 1.4. Model identification (*i.e.*, selecting the orders of an ARMA model) is dealt with in Section 1.5. The calculation of sample correlations and applications of information criteria to both univariate and multivariate cases are described. Different algorithms for estimating ARMA parameters (the Yule-Walker method, the Levinson-Durbin algorithm, Burg's algorithm, the innovations algorithm, the long AR method, the Hannan-Rissanen procedure, the maximum likelihood method, and the conditional maximum likelihood method) are presented in Section 1.6. Other useful functions and diagnostic checking capabilities are also developed in this section. Section 1.7 is devoted to forecasting using the exact and approximate best linear predictors. Spectral analysis is the theme of Section 1.8. Functions to estimate the power spectrum and smoothing of spectra in time and frequency domains using a variety of windows are provided. In Section 1.9 we present functions to implement the Kalman filter technique. Structural models and univariate ARCH, GARCH, ARCH-in-mean, and GARCH-in-mean models are discussed in Section 1.10. The procedures and functions discussed in earlier sections are used to analyze four different data sets in Section 1.11.

Data sets used in the illustrative examples are also provided with the application package so the results of the examples can be reproduced if desired. These data sets are contained in data files; they can be found in the `Data` subdirectory of the `TimeSeries` directory.

Part 1.
User's Guide to *Time Series*

1.1 Introduction

A discrete *time series* is a set of time-ordered data $\{x_{t_1}, x_{t_2}, \ldots, x_{t_i}, \ldots, x_{t_n}\}$ obtained from observations of some phenomenon over time. Throughout this documentation we will assume, as is commonly done, that the observations are made at equally spaced time intervals. This assumption enables us to use the interval between two successive observations as the unit of time and, without any loss of generality, we will denote the time series by $\{x_1, x_2, \ldots x_t, \ldots x_n\}$. The subscript t can now be referred to as time, so x_t is the observed value of the time series at time t. The total number of observations in a time series (here n) is called the length of the time series (or the length of the data). We will also assume that the observations result in real numbers, so that if at each time t a single quantity is observed, the resulting x_t is a real number, and $\{x_1, x_2, \ldots, x_n\}$ is called a scalar or univariate time series. If at each time t several related quantities are observed, x_t is a real vector and $\{x_1, x_2, \ldots, x_n\}$ corresponds to a vector or multivariate time series.

The fundamental aim of time series analysis is to understand the underlying mechanism that generates the observed data and, in turn, to forecast future values of the series. Given the unknowns that affect the observed values in time series, it is natural to suppose that the generating mechanism is probabilistic and to model time series as stochastic processes. By this we mean that the observation x_t is presumed to be a realized value of some random variable X_t; the time series $\{x_1, x_2, \ldots x_t, \ldots\}$, a single realization of a stochastic process (*i.e.*, a sequence of random variables) $\{X_1, X_2, \ldots X_t, \ldots\}$. In the following we will use the term time series to refer both to the observed data and to the stochastic process; however, X will denote a random variable and x a particular realization of X.

Examples of time series abound around us. Daily closing stock prices, monthly unemployment figures, the annual precipitation index, crime rates, and earthquake aftershock frequencies are all examples of time series we encounter. Virtually any quantity recorded over time yields a time series. To "visualize" a time series we plot our observations $\{x_t\}$ as a function of the time t. This is called a *time plot*. The following examples of time plots illustrate some typical types of time series.

> **Example 1.1** The file `lynx.dat` in the directory `TimeSeries/Data` contains the annual number of lynx trapped in northwest Canada from 1821 to 1934. To plot the series we first need to read in the data; this is done by using `ReadList`. (See Section 1.4.1 for more on how to read in data from a file.)

This loads the `TimeSeries` package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

This reads in the data.

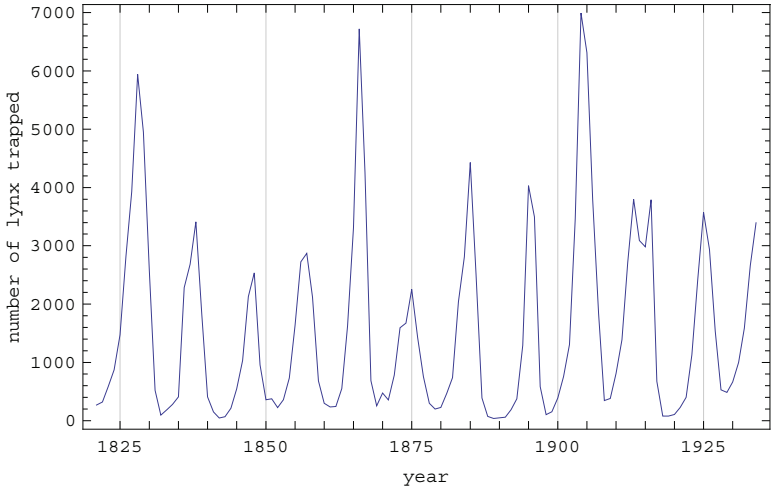
```
In[2]:= lynxdata = ReadList[ToFileName[{"TimeSeries", "Data"}, "lynx.dat"], Number];
```

To plot the data we use the *Mathematica* function `DateListPlot`.

Here is a plot of the lynx data. We see that there is a periodic oscillation with an approximate period of ten years.

```
In[3]:= DateListPlot[lynxdata, {{1821}, {1934}},
         Joined -> True, FrameLabel -> {"year", "number of lynx trapped"}]
```

Out[3]=



Example 1.2 The file `airline.dat` contains the data of monthly totals of international airline passengers in thousands from January 1949 to December 1960. It is read and plotted as a function of time.

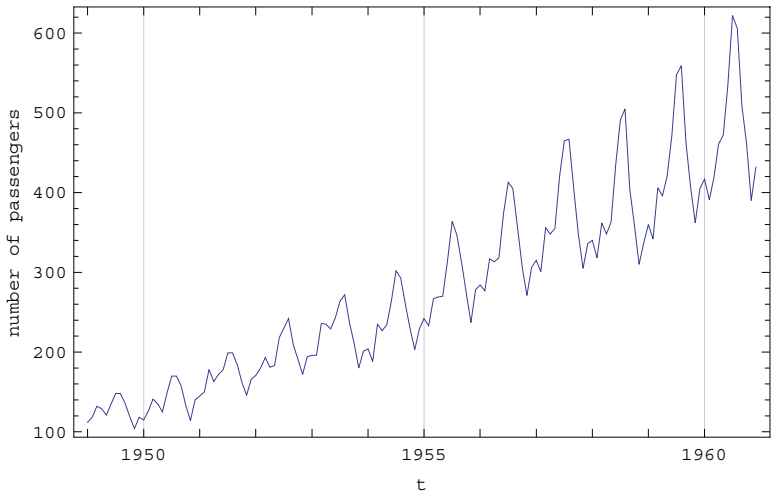
The airline passenger data is read in here.

```
In[4]:= aldata = ReadList[ToFileName[{"TimeSeries", "Data"}, "airline.dat"], Number];
```

A plot of the airline passenger data shows a cyclic component on top of a rising trend.

```
In[5]:= DateListPlot[aldata, {{1949, 1}, {1960, 12}},
         Joined -> True, FrameLabel -> {"t", "number of passengers"}]
```

Out[5]=



Example 1.3 Here is the time plot of common stock price from 1871 to 1970.

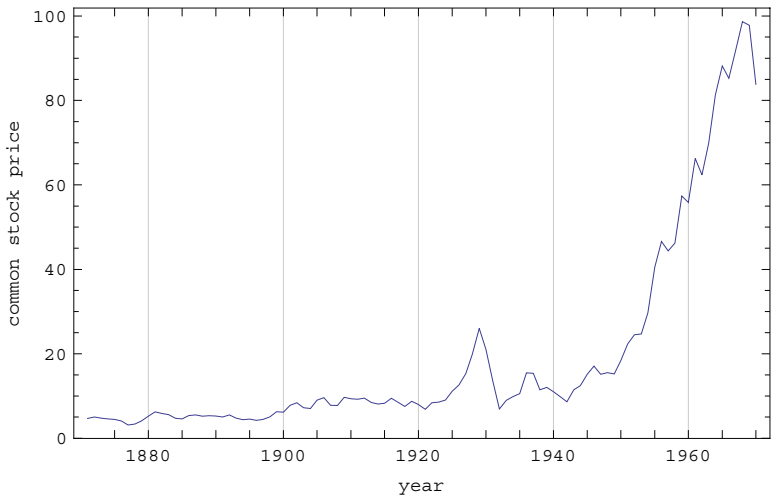
This reads in the data.

```
In[6]:= csp = ReadList[ToFileName[{"TimeSeries", "Data"}, "csp.dat"], Number];
```

The plot of the common stock price shows no apparent periodic structure with marked increase in later years.

```
In[7]:= DateListPlot[csp, {{1871}, {1970}}, Joined -> True,
         PlotRange -> All, FrameLabel -> {"year", "common stock price"}]
```

Out[7]=



The *Time Series* application package provides many of the basic tools used in the analysis of both univariate and multivariate time series. Several excellent textbooks exist that the reader may consult for detailed expositions and proofs. In this part of the manual we provide succinct summaries of the concepts and methods, introduce the functions that perform different tasks, and illustrate their usage. We first introduce some basic theoretical concepts and models used in time series analysis in Sections 1.2 and 1.3. Then in Section 1.4 we prepare data for modeling. Sections 1.5 and 1.6 deal with fitting a model to a given set of data, and Section 1.7 describes forecasting. Spectral analysis is the theme of Section 1.8. Section 1.9 introduces structural models and the Kalman filter. Section 1.10 deals with ARCH models and in the final section four data sets are analyzed using the methods introduced in this manual.

1.2 Stationary Time Series Models

In this section the commonly used linear time series models (AR, MA, and ARMA models) are defined and the objects that represent them in *Time Series* are introduced. We outline the key concepts of weak stationarity and invertibility and state the conditions on the model parameters that ensure these properties. Functions that check for stationarity and invertibility of a given ARMA model and that expand a stationary model as an approximate MA model and an invertible model as an approximate AR model are then defined. We devote a considerable portion of the section to the discussion of the fundamental quantities, covariance, correlation, and partial correlation functions. We introduce and illustrate functions that calculate these quantities. Finally, we generalize the models and concepts to multivariate time series; all the functions defined in the univariate case can also be used for multivariate models as is demonstrated in a few examples.

1.2.1 Autoregressive Moving Average Models

The fundamental assumption of time series modeling is that the value of the series at time t, X_t, depends only on its previous values (deterministic part) and on a random disturbance (stochastic part). Furthermore, if this dependence of X_t on the previous p values is assumed to be linear, we can write

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \tilde{Z}_t, \tag{2.1}$$

where $\{\phi_1, \phi_2, \ldots, \phi_p\}$ are real constants. $\tilde{Z}_t$ is the disturbance at time t, and it is usually modeled as a linear combination of zero-mean, uncorrelated random variables or a zero-mean *white noise process* $\{Z_t\}$

$$\tilde{Z}_t = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \ldots + \theta_q Z_{t-q}. \tag{2.2}$$

($\{Z_t\}$ is a white noise process with mean 0 and variance σ^2 if and only if $E\,Z_t = 0$, $E\,Z_t^2 = \sigma^2$ for all t, and $E\,Z_s\,Z_t = 0$ if $s \neq t$, where E denotes the expectation.) Z_t is often referred to as the *random error* or *noise* at time t. The constants $\{\phi_1, \phi_2, \ldots, \phi_p\}$ and $\{\theta_1, \theta_2, \ldots, \theta_q\}$ are called *autoregressive (AR) coefficients* and *moving average (MA) coefficients*, respectively, for the obvious reason that (2.1) resembles a regression model and (2.2) a moving average. Combining (2.1) and (2.2) we get

$$X_t - \phi_1 X_{t-1} - \phi_2 X_{t-2} - \ldots - \phi_p X_{t-p} = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \ldots + \theta_q Z_{t-q}. \tag{2.3}$$

This defines a zero-mean *autoregressive moving average (ARMA) process* of orders p and q, or ARMA(p, q). In general, a constant term can occur on the right-hand side of (2.3) signalling a nonzero mean process. However, any stationary ARMA process with a nonzero mean μ can be transformed into one with mean zero simply by subtracting the mean from the process. (See Section 1.2.2 for the definition of stationarity and an illustrative example.) Therefore, without any loss of generality we restrict our attention to zero-mean ARMA processes.

It is useful to introduce the *backward shift operator B* defined by

$$B^j X_t = X_{t-j}.$$

This allows us to express compactly the model described by (2.3). We define the *autoregressive polynomial* $\phi(x)$ as

$$\phi(x) = 1 - \phi_1 x - \phi_2 x^2 - \ldots - \phi_p x^p \tag{2.4}$$

and the *moving average polynomial* $\theta(x)$ as

$$\theta(x) = 1 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_q x^q, \tag{2.5}$$

and assume that $\phi(x)$ and $\theta(x)$ have no common factors. (Note the negative signs in the definition of the AR polynomial.) Equation (2.3) can be cast in the form

$$\phi(B) X_t = \theta(B) Z_t. \tag{2.6}$$

When $q = 0$ only the AR part remains and (2.3) reduces to a pure *autoregressive process* of order p denoted by AR(p)

$$X_t - \phi_1 X_{t-1} - \phi_2 X_{t-2} - \ldots - \phi_p X_{t-p} = \phi(B) X_t = Z_t. \tag{2.7}$$

Similarly, if $p = 0$, we obtain a pure *moving average process* of order q, MA(q),

$$X_t = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \ldots + \theta_q Z_{t-q} = \theta(B) Z_t.$$

When neither p nor q is zero, an ARMA(p, q) model is sometimes referred to as a "mixed model".

The commonly used time series models are represented in this package by objects of the generic form `model[`$param_1$`, `$param_2$`, ...]`. Since an ARMA model is defined by its AR and MA coefficients and the white noise variance (the noise is assumed to be normally distributed), the object

$$\texttt{ARMAModel}[\{\phi_1, \phi_2, \ldots, \phi_p\}, \{\theta_1, \theta_2, \ldots, \theta_q\}, \sigma^2]$$

specifies an ARMA(p, q) model with AR coefficients $\{\phi_1, \phi_2, \ldots, \phi_p\}$ and MA coefficients $\{\theta_1, \theta_2, \ldots, \theta_q\}$ and noise variance σ^2. Note that the AR and MA coefficients are enclosed in lists. Similarly, the object

$$\texttt{ARModel}[\{\phi_1, \phi_2, \ldots, \phi_p\}, \sigma^2]$$

specifies an AR(p) model and

$$\texttt{MAModel}[\{\theta_1, \theta_2, \ldots, \theta_q\}, \sigma^2]$$

denotes an MA(q) model. Each of these objects provides a convenient way of organizing the parameters and serves to specify a particular model. It cannot itself be evaluated. For example, if we enter an `MAModel` object it is returned unevaluated.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

The object is not evaluated.

```
In[2]:= MAModel[{theta1, theta2}, sigma2]

Out[2]= MAModel[{theta1, theta2}, sigma2]
```

These objects are either used as arguments of time series functions or generated as output, as we will see later in examples.

1.2.2 Stationarity

In order to make any kind of statistical inference from a single realization of a random process, stationarity of the process is often assumed. Intuitively, a process $\{X_t\}$ is stationary if its statistical properties do not change over time. More precisely, the probability distributions of the process are time-invariant. In practice, a much weaker definition of stationarity called *second-order stationarity* or *weak stationarity* is employed. Let E denote the expectation of a random process. The mean, variance, and covariance of the process are defined as follows:

mean: $\mu(t) = E\, X_t$,

variance: $\sigma^2\,(t) = Var\,(X_t) = E\,(X_t - \mu(t))^2$, and

covariance: $\gamma\,(s, r) = Cov\,(X_s, X_r) = E\,(X_s - \mu(s))\,(X_r - \mu(r))$.

A time series is second-order stationary if it satisfies the following conditions:

(a) $\mu(t) = \mu$, and $\sigma^2(t) = \sigma^2$ for all t, and

(b) $\gamma(s, r)$ is a function of $(s - r)$ only.

Henceforth, we will drop the qualifier "second-order" and a stationary process will always refer to a second-order or weak stationary process.

By definition, stationarity implies that the process has a constant mean μ. This allows us without loss of generality to consider only zero-mean processes since a constant mean can be transformed away by subtracting the mean value from the process as illustrated in the following example.

Example 2.1 Transform a nonzero mean stationary ARMA(p, q) process to a zero-mean one.

A nonzero mean stationary ARMA(p, q) model is defined by $\phi(B)\, X_t = \delta + \theta(B)\, Z_t$, where δ is a constant and $\phi(x)$ and $\theta(x)$ are AR and MA polynomials defined earlier. Taking the expectation on both sides we have $\phi(1)\,\mu = \delta$ or $\mu = \delta\,/\,\phi\,(1)$. (For a stationary model we have $\phi(1) \neq 0$. See Example 2.2.) Now if we denote $X_t - \mu$ as Y_t, the process $\{Y_t\}$ is a zero-mean, stationary process satisfying $\phi(B)\, Y_t = \theta(B)\, Z_t$.

There is another important consequence of stationarity. The fact that the covariance $Cov(X_s, X_r)$ of a stationary process is only a function of the time difference $s - r$ (which is termed *lag*) allows us to define two fundamental quantities of time series analysis: *covariance function* and *correlation function*. The covariance function $\gamma(k)$ is defined by

$$\gamma(k) = Cov(X_{k+t}, X_t) = E(X_{k+t} - \mu)(X_t - \mu) \tag{2.8}$$

and the correlation function $\rho(k)$ is

$$\rho(k) = Corr(X_{t+k}, X_t) = Cov(X_{t+k}, X_t) \Big/ \sqrt{Var(X_{t+k})\, Var(X_t)} = \gamma(k) \Big/ \gamma(0).$$

Consequently, a correlation function is simply a normalized version of the covariance function. It is worth noting the following properties of $\rho(k)$: $\rho(k) = \rho(-k)$, $\rho(0) = 1$, and $|\rho(k)| \leq 1$.

Before we discuss the calculation of these functions in the next section, we first turn to the ARMA models defined earlier and see what restrictions stationarity imposes on the model parameters. This can be seen from the following simple example.

Example 2.2 Derive the covariance and correlation functions of an AR(1) process $X_t = \phi_1 X_{t-1} + Z_t$.

From (2.8) we obtain the covariance function at lag zero $\gamma(0) = E(\phi_1 X_{t-1} + Z_t)^2 = \phi_1^2 \gamma(0) + \sigma^2$, and hence, $\gamma(0) = \sigma^2 \Big/ \left(1 - \phi_1^2\right)$. Now $\gamma(k) = E(X_{t+k} X_t) = E((\phi_1 X_{t+k-1} + Z_{t+k}) X_t) = \phi_1 \gamma(k-1)$. Iterating this we get $\gamma(k) = \phi_1^k \gamma(0) = \phi_1^k \sigma^2 \Big/ \left(1 - \phi_1^2\right)$ and the correlation function $\rho(k) = \phi_1^k$. (Note that we have used $E X_t Z_{t+k} = 0$ for $k > 0$.)

In the above calculation we have assumed stationarity. This is true only if $|\phi_1| < 1$ or, equivalently, the magnitude of the zero of the AR polynomial $\phi(x) = 1 - \phi_1 x$ is greater than one so that $\gamma(0)$ is positive. This condition of stationarity is, in fact, general. An ARMA model is stationary if and only if all the zeros of the AR polynomial $\phi(x)$ lie outside the unit circle in the complex plane. In contrast, some authors refer to this condition as the causality condition: an ARMA model is causal if all the zeros of its AR polynomial lie outside the unit circle. They define a model to be stationary if its AR polynomial has no zero *on* the unit circle. See for example, Brockwell and Davis (1987), Chapter 3.

A stationary ARMA model can be expanded formally as an MA(∞) model by inverting the AR polynomial and expanding $\phi^{-1}(B)$. From (2.6), we have

$$X_t = \phi^{-1}(B)\, \theta(B)\, Z_t = \sum_{j=0}^{\infty} \psi_j Z_{t-j}, \tag{2.9}$$

where $\{\psi_j\}$ are the coefficients of the equivalent MA(∞) model and are often referred to as ψ weights. For example, an AR(1) model can be written as $X_t = (1 - \phi_1 B)^{-1} Z_t = \sum_{i=0}^{\infty} \phi_1^i Z_{t-i}$, *i.e.*, $\psi_j = \phi_1^j$.

Similarly, we say an ARMA model is *invertible* if all the zeros of its MA polynomial lie outside the unit circle, and an invertible ARMA model in turn can be expanded as an AR(∞) model

$$Z_t = \theta^{-1}(B)\,\phi(B)\,X_t = \sum_{j=0}^{\infty} \pi_j\, X_{t-j}. \tag{2.10}$$

Note the symmetry or duality between the AR and MA parts of an ARMA process. We will encounter this duality again later when we discuss the correlation function and the partial correlation function in the next two sections.

To check if a particular model is stationary or invertible, the following functions can be used:

`StationaryQ[`*model*`]` or `StationaryQ[{`ϕ_1`, … , `ϕ_p`}]`

or

`InvertibleQ[`*model*`]` or `InvertibleQ[{`θ_1`, … , `θ_q`}]`.

(Henceforth, when *model* is used as a *Mathematica* function argument it means the model object.) When the model coefficients are numerical, these functions solve the equations $\phi(x) = 0$ and $\theta(x) = 0$, respectively, check whether any root has an absolute value less than or equal to one, and give `True` or `False` as the output.

Example 2.3 Check to see if the ARMA(2, 1) model $X_t - 0.5\,X_{t-1} + 1.2\,X_{t-2} = Z_t + 0.7\,Z_{t-1}$ is stationary and invertible. (The noise variance is 1.)

The model is not stationary.

```
In[3]:= StationaryQ[ARMAModel[{0.5, -1.2}, {0.7}, 1]]

Out[3]= False
```

Since the stationarity condition depends only on the AR coefficients, we can also simply input the list of AR coefficients.

This gives the same result as above.

```
In[4]:= StationaryQ[{0.5, -1.2}]

Out[4]= False
```

We can, of course, use *Mathematica* to explicitly solve the equation $\phi(x) = 0$ and check that there are indeed roots inside the unit circle.

This solves the equation $\phi(x) = 0$.

```
In[5]:= Solve[1 - 0.5 x + 1.2 x^2 == 0, x]

Out[5]= {{x → 0.208333 - 0.88878 i}, {x → 0.208333 + 0.88878 i}}
```

This gives the absolute values of the two roots. In *Mathematica*, `%` represents the last output and `Abs[x /. %]` substitutes the roots in x and finds the absolute values.

```
In[6]:= Abs[x /. %]

Out[6]= {0.912871, 0.912871}
```

We can check invertibility using the function `InvertibleQ`.

The model is found to be invertible.

```
In[7]:= InvertibleQ[ARMAModel[{0.5, -1.2}, {0.7}, 1]]

Out[7]= True
```

We can also just input the MA coefficients to check invertibility.

```
In[8]:= InvertibleQ[{0.7}]

Out[8]= True
```

Thus the model under consideration is invertible but not stationary.

The functions `StationaryQ` and `InvertibleQ` give `True` or `False` only when the corresponding AR or MA parameters are numerical. The presence of symbolic parameters prevents the determination of the locations of the zeros of the corresponding polynomials and, therefore, stationarity or invertibility cannot be determined, as in the following example.

No `True` or `False` is returned when the coefficients of the polynomial are not numerical.

```
In[9]:= StationaryQ[{p1, p2}]

Out[9]= StationaryQ[{p1, p2}]
```

Next we define the functions that allow us to expand a stationary ARMA model as an approximate MA(q) model ($X_t \approx \sum_{j=0}^{q} \psi_j Z_{t-j}$) using (2.9) or an invertible ARMA model as an approximate AR(p) model ($\sum_{j=0}^{p} \pi_j X_{t-j} \approx Z_t$) using (2.10). The function

`ToARModel[`*model*`, `p`]`

gives the order p truncation of the AR(∞) expansion of *model*. Similarly,

`ToMAModel[`*model*`, `q`]`

yields the order q truncation of the MA(∞) expansion of *model*. The usage of these functions is illustrated in the following example.

Example 2.4 Expand the model $X_t - 0.9\,X_{t-1} + 0.3\,X_{t-2} = Z_t$ as an approximate MA(5) model using (2.9) and the model $X_t - 0.7\,X_{t-1} = Z_t - 0.5\,Z_{t-1}$ as an approximate AR(6) model using (2.10). The noise variance is 1.

This expands the given AR(2) model as an MA(5) model.

```
In[10]:= ToMAModel[ARModel[{0.9, -0.3}, 1], 5]

Out[10]= MAModel[{0.9, 0.51, 0.189, 0.0171, -0.04131}, 1]
```

In the above calculation, as in some others, the value of the noise variance σ^2 is not used. In this case we can omit the noise variance from the model objects.

Here we suppress the noise variance. This does not affect the expansion coefficients.

```
In[11]:= ToARModel[ARMAModel[{0.7}, {-0.5}], 6]

Out[11]= ARModel[{0.2, 0.1, 0.05, 0.025, 0.0125, 0.00625}]
```

We can, of course, include the variance back in the model object using `Append`.

The noise variance is inserted back into the model object.

```
In[12]:= Append[%, var]

Out[12]= ARModel[{0.2, 0.1, 0.05, 0.025, 0.0125, 0.00625}, var]
```

If the model is not stationary or invertible, the corresponding expansion is not valid. If we insist on doing the expansion anyway, a warning message will appear along with the formal expansion as seen below.

A warning message comes with the expansion result.

```
In[13]:= ToARModel[MAModel[{1.2}, 1], 4]

       ToARModel::nonin : Warning: The model MAModel[{1.2}, 1] is not invertible.

Out[13]= ARModel[{1.2, -1.44, 1.728, -2.0736}, 1]
```

These functions can also be used to expand models with symbolic parameters, but bear in mind that it is usually slower to do symbolic calculations than to do numerical ones.

This expands an ARMA(1, 1) model with symbolic parameters.

```
In[14]:= ToMAModel[ARMAModel[{p1}, {t1}, s], 4]

Out[14]= MAModel[{p1 + t1, p1 (p1 + t1), p1^2 (p1 + t1), p1^3 (p1 + t1)}, s]
```

1.2.3 Covariance and Correlation Functions

We have defined the covariance and correlation functions of a stationary process in Section 1.2.2. We now illustrate how to obtain the covariance and the correlation functions of a given model using this package. The functions

`CovarianceFunction[`*model*`, h]` and `CorrelationFunction[`*model*`, h]`

give, respectively, the covariance and the correlation functions of the given model up to lag h, that is, $\{\gamma(0), \gamma(1), \ldots, \gamma(h)\}$ and $\{\rho(0), \rho(1), \ldots, \rho(h)\}$. The code for each function solves internally a set of difference equations obtained by multiplying both sides of (2.3) by X_{t-k} ($k = 0, 1, \ldots$) and taking expectations. For mathematical details see Brockwell and Davis (1987), p. 92.

We begin by considering the behavior of the covariance and correlation functions of AR models.

> **Example 2.5** In Example 2.2 we derived the covariance and correlation functions of AR(1) model. Now we can obtain the same results using `CovarianceFunction` and `CorrelationFunction`.

This calculates the covariance function of an AR(1) model up to lag 4.

```
In[15]:= CovarianceFunction[ARModel[{p1}, s], 4]
```

$$\text{Out[15]}= \left\{-\frac{s}{-1+p1^2}, -\frac{p1\,s}{-1+p1^2}, -\frac{p1^2\,s}{-1+p1^2}, -\frac{p1^3\,s}{-1+p1^2}, -\frac{p1^4\,s}{-1+p1^2}\right\}$$

These are the covariances at lags 0, 1, …, 4. Note that the first entry in the output is $\gamma(0)$, the variance of the series. To get the correlation function we need to divide the above result by its first entry $\gamma(0)$. This can be done explicitly as follows.

The correlation function of the AR(1) model is calculated up to lag 4. The expression `%[[1]]` represents the first element of the last output.

```
In[16]:= %/%[[1]]
```

$$\text{Out[16]}= \left\{1, p1, p1^2, p1^3, p1^4\right\}$$

We may also obtain the correlation function directly.

This gives the same correlation function up to lag 4.

```
In[17]:= corr = CorrelationFunction[ARModel[{p1}, s], 4]
```

$$\text{Out[17]}= \left\{1, p1, p1^2, p1^3, p1^4\right\}$$

Both `CorrelationFunction` and `CovarianceFunction` use the function `StationaryQ` to check the stationarity of the model before computing the correlation or covariance function. If the model is manifestly nonstationary, the covariance or the correlation is not calculated.

When the model is not stationary, no correlation is calculated.

```
In[18]:= CorrelationFunction[ARModel[{-0.8, 0.7}, 1], 4]

       CovarianceFunction::nonst :
        The model ARMAModel[{-0.8, 0.7}, {0.}, 1] is not stationary.

Out[18]= CorrelationFunction[ARModel[{-0.8, 0.7}, 1], 4]
```

When symbolic coefficients are used, `StationaryQ` will not give `True` or `False` and the covariance and the correlation functions are calculated assuming the model is stationary, as we have seen in Example 2.5.

Example 2.5 shows that the correlation function of an AR(1) process decays exponentially for a stationary model, and when $\phi_1 < 0$ it oscillates between positive and negative values. In order to visualize the "shape" of the correlation function, we plot it as a function of the lag. Since the correlation function is a list of discrete values, we use the *Mathematica* function `ListPlot` to plot it. `ListLinePlot` could be used instead if a line plot through the values is desired.

The output of `CorrelationFunction` or `CovarianceFunction` corresponds to lags 0, 1, 2, …, while `ListPlot` assumes x coordinates of 1, 2, … if x coordiantes are not explictly given. To match the lags with the correct correlation terms, we can either drop the first entry of the output of `CorrelationFunction` and plot $\{\rho(1), \rho(2), \ldots, \rho(h)\}$ using `ListPlot`, plot the output using `ListPlot` with an additional `DataRange` option, or input the correlation data to `ListPlot` in the form $\{\{0, \rho(0)\}, \{1, \rho(1)\}, \ldots, \{h, \rho(h)\}\}$. This form can be obtained by using `Transpose`.

Here we recast the output of `CorrelationFunction` in the desired form.

```
In[19]:= Transpose[{Range[0, 4], corr}]

Out[19]= {{0, 1}, {1, p1}, {2, p1^2}, {3, p1^3}, {4, p1^4}}
```

The output can then be used as the argument of `ListPlot`. It is convenient to define a function that takes the output of `CorrelationFunction` and does the appropriate plot so the data does not need to be manually processed every time we plot the correlation function. We will choose to use the `DataRange` option to `ListPlot` and define the function `plotcorr` as follows.

This defines `plotcorr` for plotting correlation functions.

```
In[20]:= plotcorr[corr_, opts___] := ListPlot[corr, DataRange -> {0, Length[corr] - 1}, opts]
```

The first argument in `plotcorr` is the output of `CorrelationFunction` and the second is the usual set of options for `ListPlot`. Next we display the two typical forms of the correlation function of an AR(1) process, one for positive ϕ_1(= 0.7) and one for negative ϕ_1(= −0.7) using the function `plotcorr`. (Since the correlation function of a univariate time series is independent of the noise variance, we set $\sigma^2 = 1$.)

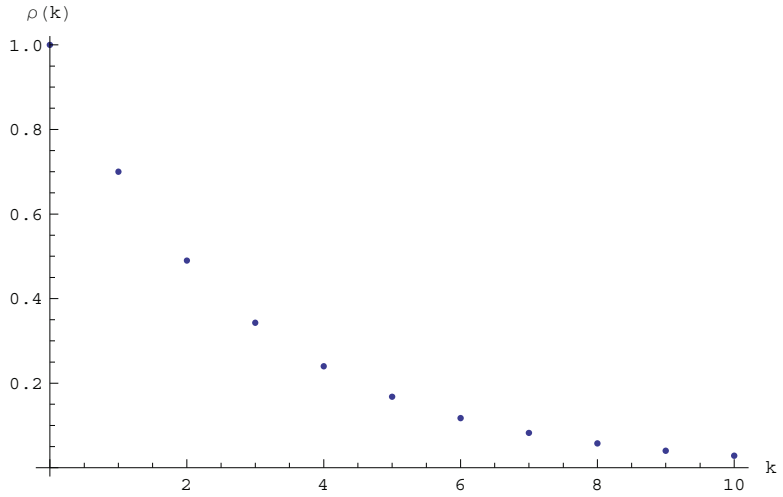
This calculates the correlation function up to lag 10. The semicolon at the end of the command prevents the display of the output.

```
In[21]:= corr1 = CorrelationFunction[ARModel[{0.7}, 1], 10];
```

This is the plot of the correlation function.

```
In[22]:= plotcorr[corr1, AxesLabel -> {"k", "ρ(k)"}]
```

Out[22]=

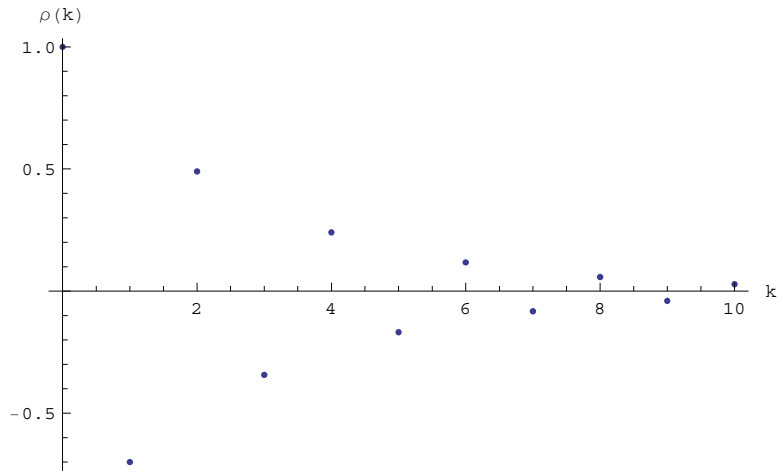


Note that if we do not need the correlation function for other purposes we can directly include the calculation of the correlation function inside the function `plotcorr` as shown below.

This gives the plot of the correlation function for $\phi_1 = -0.7$.

```
In[23]:= plotcorr[CorrelationFunction[ARModel[{-0.7}, 1], 10], AxesLabel -> {"k", "ρ(k)"}]
```

Out[23]=



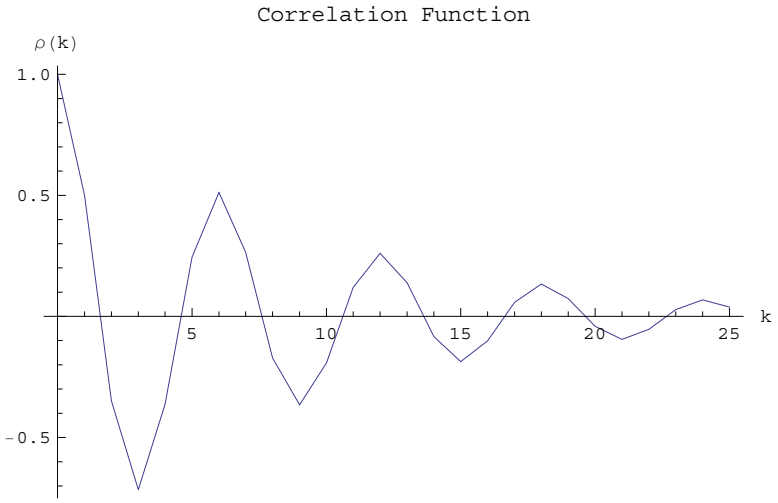
We have given the option `AxesLabel -> {"k", "ρ(k)"}` to label the axes. We can also specify other options of `ListPlot`. (To find out all the options of `ListPlot` use `Options[ListPlot]`.) For example, if we want to join the points plotted, then the option `Joined -> True` should be given, and if we want to label the plot we use `PlotLabel -> "label"` as in the following example.

Example 2.6 Plot the correlation function of the Yule, or AR(2), process: $X_t = 0.9\, X_{t-1} - 0.8\, X_{t-2} + Z_t$.

The correlation function of the given AR(2) process is plotted. For future re-display, we have called this graph `g1` (see Example 5.1).

```
In[24]:= g1 = plotcorr[CorrelationFunction[ARModel[{0.9, -0.8}, 1], 25],
            AxesLabel -> {"k", "ρ(k)"}, Joined -> True, PlotLabel -> "Correlation Function"]
```

Out[24]=



The way the correlation function decays is intimately related to the roots of $\phi(x) = 0$. Complex roots give rise to oscillatory behavior of the correlation function as we observe in this example. For the explicit expression of the covariance function $\gamma(k)$ in terms of the zeros of $\phi(x)$, see Brockwell and Davis (1987), Section 3.3.

Next we study the behavior of MA models. Recall that the covariance function $\gamma(k)$ of an ARMA process is calculated by multiplying both sides of (2.3) by X_{t-k} and computing expectations. Note that for an MA(q) process when $k > q$ there is no overlap on the right-hand side of (2.3). Thus $\gamma(k) = 0$ ($\rho(k) = 0$) for $k > q$. This is characteristic of the MA correlation function, and it is, in fact, often used to identify the order of an MA process, as we shall see in Section 1.5.

Example 2.7 Find the correlation function up to lag 4 of an MA(2) process: $X_t = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2}$.

This calculates the correlation function up to lag 4 of an MA(2) process.

```
In[25]:= CorrelationFunction[MAModel[{t1, t2}, var], 4]
```

$$\text{Out[25]} = \left\{1,\ \frac{t1\ (1+t2)}{1+t1^2+t2^2},\ \frac{t2}{1+t1^2+t2^2},\ 0,\ 0\right\}$$

We see that $\rho(k) = 0$ for $k > 2$.

In fact, the correlation function of an MA model can be easily worked out analytically (see Brockwell and Davis (1977), p. 93). In particular, when an MA(q) model has equal θ weights (*i.e.*, $\theta_1 = \theta_2 = \ldots = \theta_q = \theta$), the correlation function is given by $\gamma(0) = 1$, $\gamma(k) = 0$ for $k > q$, and $\gamma(k) = \left(\theta + \theta^2(q-k)\right) / \left(1 + \theta^2 q\right)$ for $0 < k \le q$. In particular, when $\theta = \theta_0 = 1$, the correlation function is $\gamma(k) = (1 + q - k)/(1 + q)$ for $k \le q$, a straight line with slope $-1/(1+q)$.

(For convenience, we define $\theta_0 = 1$ so that the MA polynomial can be written as $\theta(x) = \sum_{i=0}^{q} \theta_i x^i$. Similarly we write the AR polynomial as $\phi(x) = \sum_{i=0}^{p} \phi_i x^i$ with $\phi_0 = 1$.)

Example 2.8 Find the correlation function of an MA(8) model with equal θ weights.

This calculates the correlation function of an MA(8) model with equal θ weights.

```
In[26]:= CorrelationFunction[MAModel[Table[t1, {8}], 1], 10]
```

$$\text{Out[26]} = \left\{1, \frac{\text{t1}\,(1+7\,\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}\,(1+6\,\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}\,(1+5\,\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}\,(1+4\,\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}\,(1+3\,\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}\,(1+2\,\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}\,(1+\text{t1})}{1+8\,\text{t1}^2}, \frac{\text{t1}}{1+8\,\text{t1}^2}, 0, 0\right\}$$

Note that we have avoided typing `t1` eight times by using `Table[t1, {8}]` to generate the eight identical MA coefficients. To get the correlation function for $\theta = 1$ we can simply substitute `t1=1` in the above expression using `% /. t1 -> 1`.

This gives the correlation function for $\theta = 1$.

```
In[27]:= corr = % /. t1 -> 1
```

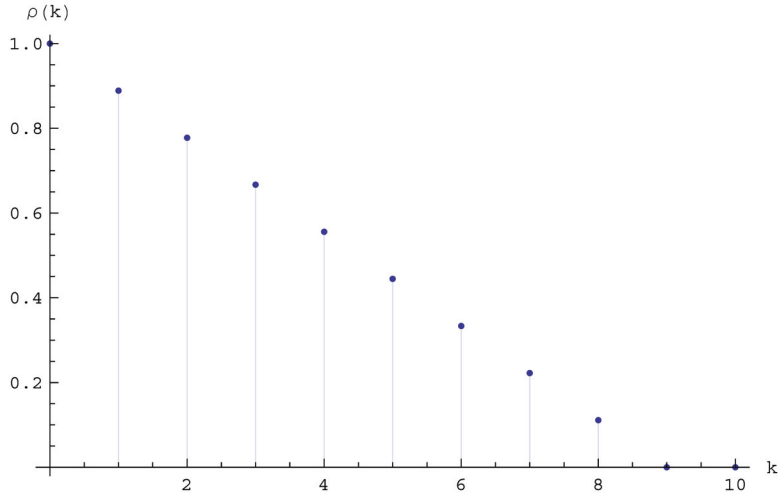
$$\text{Out[27]} = \left\{1, \frac{8}{9}, \frac{7}{9}, \frac{2}{3}, \frac{5}{9}, \frac{4}{9}, \frac{1}{3}, \frac{2}{9}, \frac{1}{9}, 0, 0\right\}$$

To emphasize the discrete nature of the correlation function some people prefer to plot the correlation function as discrete lines joining the points $\{i, 0\}$ and $\{i, \rho(i)\}$ for $i = 0, 1, \ldots, h$. It is easy to implement this type of plot in *Mathematica*, via `ListPlot` with a `Filling` option.

The function defined above is used to plot the correlation function of the MA(8) model with equal θ weights.

```
In[28]:= ListPlot[corr, Filling -> Axis, DataRange -> {0, 10}, AxesLabel -> {"k", "ρ(k)"}]
```

Out[28] =

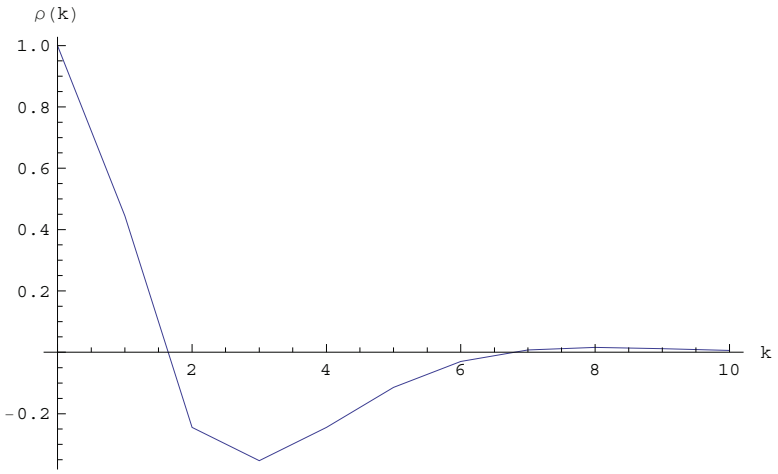


Example 2.9 The correlation function of a stationary ARMA(p, q) process in general decays exponentially. Here we plot the correlation function up to lag 10 of the ARMA(2, 2) model $X_t - 0.9\,X_{t-1} + 0.3\,X_{t-2} = Z_t + 0.2\,Z_{t-1} - 1.2\,Z_{t-2}$ using the function `plotcorr` we defined earlier. The noise variance $\sigma^2 = 1$.

This shows the correlation function of the ARMA(2, 2) model.

```
In[29]:= plotcorr[CorrelationFunction[ARMAModel[{0.9, -0.3}, {0.2, -1.2}, 1], 10],
          AxesLabel -> {"k", "ρ(k)"}, Joined -> True]
```

Out[29]=



Using `CorrelationFunction` we can generate the correlation functions of different models; plotting the correlation functions enables us to develop intuition about different processes. The reader is urged to try a few examples.

1.2.4 Partial Correlation Functions

In Example 2.1 we calculated the correlation function of an AR(1) process. Although X_t depends only on X_{t-1}, the correlations at large lags are, nevertheless, nonzero. This should not be surprising because X_{t-1} depends on X_{t-2}, and in turn X_{t-2} on X_{t-3}, and so on, leading to an indirect dependence of X_t on X_{t-k}. This can also be understood by inverting the AR polynomial and writing

$$X_t = (1 - \phi_1 B)^{-1} Z_t = \sum_{i=0}^{\infty} \phi_1^i B^i Z_t = \sum_{i=0}^{\infty} \phi_1^i Z_{t-i}.$$

Multiplying both sides of the above equation by X_{t-k} and taking expectations, we see that the right-hand side is not strictly zero no matter how large k is. In other words, X_t and X_{t-k} are correlated for all k. This is true for all AR(p) and ARMA(p, q) processes with $p \neq 0$, and we say that the correlation of an AR or ARMA process has no "sharp cutoff" beyond which it becomes zero.

However, consider the conditional expectation $E(X_t\,X_{t-2} \mid X_{t-1})$ of an AR(1) process, that is, given X_{t-1}, what is the correlation between X_t and X_{t-2}? It is clearly zero since $X_t = \phi_1 X_{t-1} + Z_t$ is not influenced by X_{t-2} given X_{t-1}. The partial correlation between X_t and X_{t-k} is defined as the correlation between the two random variables with

all variables in the intervening time $\{X_{t-1}, X_{t-2}, \ldots, X_{t-k+1}\}$ assumed to be fixed. Clearly, for an AR(p) process the partial correlation so defined is zero at lags greater than the AR order p. This fact is often used in attempts to identify the order of an AR process. Therefore, we introduce the function

`PartialCorrelationFunction[`*model*`, `*h*`]`,

which gives the partial correlation $\phi_{k,k}$ of the given model for $k = 1, 2, \ldots, h$. It uses the Levinson-Durbin algorithm, which will be presented briefly in Section 1.6. For details of the algorithm and more about the partial correlation function, see Brockwell and Davis (1987), pp. 162–164.

Example 2.10 Let us compute the partial correlation function $\phi_{k,k}$ of an AR(2) process up to lag 4. Note that in contrast to the correlation function the output of `PartialCorrelationFunction` starts from lag 1, and $\phi_{1,1} = \rho(1)$.

This gives the partial correlation function of an AR(2) model up to lag 4.

```
In[30]:= PartialCorrelationFunction[ARModel[{p1, p2}, s], 4]
```

$$\text{Out[30]} = \left\{-\frac{\text{p1}}{-1+\text{p2}}, \text{p2}, 0, 0\right\}$$

We observe that for an AR(p) process $\phi_{k,k} = 0$ for $k > p$.

Example 2.11 Find the partial correlation function of an MA(1) model.

Here the partial correlation function of an MA(1) model up to lag 4 is computed.

```
In[31]:= PartialCorrelationFunction[MAModel[{t1}, 1], 4]
```

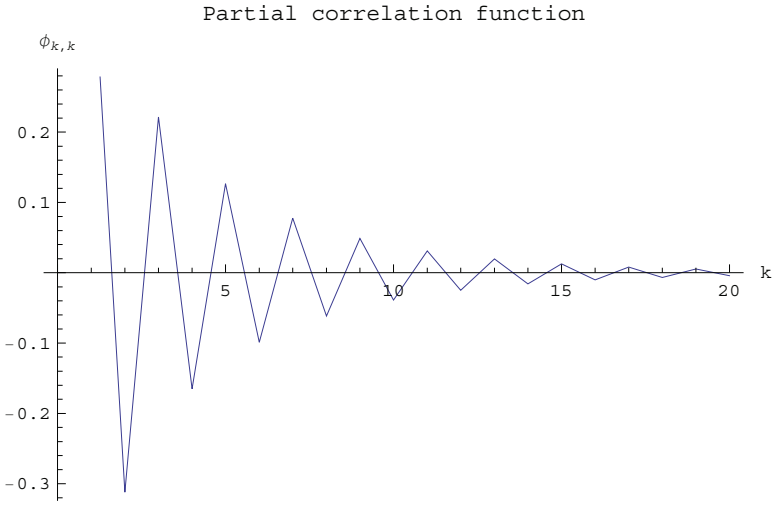
$$\text{Out[31]} = \left\{\frac{\text{t1}}{1+\text{t1}^2}, -\frac{\text{t1}^2}{1+\text{t1}^2+\text{t1}^4}, \frac{\text{t1}^3}{1+\text{t1}^2+\text{t1}^4+\text{t1}^6}, -\frac{\text{t1}^4}{1+\text{t1}^2+\text{t1}^4+\text{t1}^6+\text{t1}^8}\right\}$$

The analytic expression for the partial correlation function is $\phi_{k,k} = -(-\theta)^k \left(1-\theta^2\right) \Big/ \left(1-\theta^{2(k+1)}\right)$, and we see that there is no sharp cutoff in the partial correlation. This property is, in fact, shared by all the MA(q) and ARMA(p, q) models with $q \neq 0$. It can be understood by expanding an invertible MA model as an AR(∞) model. X_t is always related to X_{t-k} with the intervening variables fixed for all k. Observe the duality between the AR and MA models: for an AR(p) model, the partial correlation function $\phi_{k,k}$ is zero for $k > p$ and the correlation function does not have a sharp cutoff, whereas for an MA(q) model the correlation function $\gamma(k)$ is zero for $k > q$ and the partial correlation function has no sharp cutoff.

Here is the plot of the partial correlation function of the MA(1) model in Example 2.11 with $\theta_1 = 0.8$. Since the partial correlation function starts from lag 1, in contrast to the correlation function we can use `ListLinePlot` directly.

```
In[32]:= ListLinePlot[PartialCorrelationFunction[MAModel[{0.8}, 1], 20],
          AxesLabel -> {"k", "ϕk,k"}, PlotLabel -> "Partial correlation function"]
```

Out[32]=



1.2.5 Multivariate ARMA Models

In some cases, at each time t, several related quantities are observed and, therefore, we want to study these quantities simultaneously by grouping them together to form a vector. By so doing we have a vector or multivariate process. It is straightforward to generalize the definition of a univariate ARMA model to the multivariate case. Let $\mathbf{X}_t = (X_{t1}, X_{t2}, \ldots, X_{tm})'$ and $\mathbf{Z}_t = (Z_{t1}, Z_{t2}, \ldots, Z_{tm})'$ be m-dimensional random vectors (here $'$ denotes transpose). A zero-mean, m-variate ARMA(p, q) model is defined by

$$\mathbf{X}_t - \Phi_1 \mathbf{X}_{t-1} - \ldots - \Phi_p \mathbf{X}_{t-p} = \mathbf{Z}_t + \Theta_1 \mathbf{Z}_{t-1} + \ldots + \Theta_q \mathbf{Z}_{t-q} \tag{2.11}$$

where the AR and MA coefficients $\{\Phi_i\}$ ($i = 1, 2, \ldots, p$) and $\{\Theta_i\}$, ($i = 1, 2, \ldots, q$) are all real $m \times m$ matrices and the zero-mean white noise $\{\mathbf{Z}_t\}$ is characterized by a covariance matrix Σ. Again, it is useful to define the matrix AR polynomial $\Phi(x)$ and the matrix MA polynomial $\Theta(x)$ by

$$\Phi(x) = I - \Phi_1 x - \Phi_2 x^2 - \ldots - \Phi_p x^p$$

and

$$\Theta(x) = I + \Theta_1 x + \Theta_2 x + \ldots + \Theta_q x^q,$$

where I denotes the $m \times m$ identity matrix. Now (2.11) can be written as $\Phi(B)\mathbf{X}_t = \Theta(B)\mathbf{Z}_t$, where B is the backward shift operator.

As in the univariate case, a multivariate ARMA(p, q) model is represented by the object

$$\texttt{ARMAModel}[\{\Phi_1, \Phi_2, \ldots, \Phi_p\}, \{\Theta_1, \Theta_2, \ldots, \Theta_q\}, \Sigma]$$

where each parameter matrix must be entered according to *Mathematica* convention. For example, the AR(1) model

$$\begin{pmatrix} X_{t1} \\ X_{t2} \end{pmatrix} = \begin{pmatrix} \phi_{11} & \phi_{12} \\ \phi_{21} & \phi_{22} \end{pmatrix} \begin{pmatrix} X_{t-11} \\ X_{t-12} \end{pmatrix} + \begin{pmatrix} Z_{t1} \\ Z_{t2} \end{pmatrix}$$

with noise covariance Σ ($\Sigma_{ij} = \sigma_{ij}$) is represented as

$$\texttt{ARModel}[\{\{\{\phi_{11}, \phi_{12}\}, \{\phi_{21}, \phi_{22}\}\}\}, \{\{\sigma_{11}, \sigma_{12}\}, \{\sigma_{21}, \sigma_{22}\}\}].$$

The various quantities defined in the univariate case can be extended to the multivariate case. We proceed to do this and illustrate how they are computed.

Stationarity and Invertibility

The definitions of mean, variance, and covariance, and the stationarity condition can be extended straightforwardly to a multivariate process. Now the mean is a vector and the variance and covariance are matrices. They are defined as follows:

mean: $\mu(t) = E\,\mathbf{X}_t$,

variance: $\Sigma(t) = Var(\mathbf{X}_t) = E(\mathbf{X}_t - \mu(t))(\mathbf{X}_t - \mu(t))'$, and

covariance: $\Gamma(s, r) = Cov(\mathbf{X}_s, \mathbf{X}_r) = E(\mathbf{X}_s - \mu(s))(\mathbf{X}_r - \mu(r))'$.

The stationarity condition for a multivariate ARMA model can be translated into the algebraic condition that all the roots of the determinantal equation $|\Phi(x)| = 0$ lie outside the unit circle.

Example 2.12 As an exercise in using *Mathematica*, we illustrate how to explicitly check if the bivariate AR(1) model $\mathbf{X}_t - \Phi_1 \mathbf{X}_{t-1} = \mathbf{Z}_t$ is stationary where $\Phi_1 = \{\{0.6, -0.7\}, \{1.2, -0.5\}\}$.

We first define the determinant. Here `IdentityMatrix[2]` generates the 2×2 identity matrix.

```
In[33]:= eq = Det[IdentityMatrix[2] - {{0.6, -0.7}, {1.2, -0.5}} x];
```

This solves the equation.

```
In[34]:= Solve[eq == 0, x]

Out[34]= {{x → 0.0925926 - 1.35767 i}, {x → 0.0925926 + 1.35767 i}}
```

We now find the absolute values of the roots.

```
In[35]:= Abs[x /. %]

Out[35]= {1.36083, 1.36083}
```

Since both roots lie outside the unit circle, we conclude that the model is stationary.

In fact, all the *Mathematica* functions introduced previously for the univariate case have been designed so that they can be used with appropriate input for multivariate time series as well. Specifically, here we use the function `StationaryQ` directly to check the stationarity of the bivariate AR(1) model of Example 2.12.

The model is stationary.

```
In[36]:= StationaryQ[ARModel[{{{0.6, -0.7}, {1.2, -0.5}}}, {{1, 0}, {0, 1}}]]

Out[36]= True
```

Similarly, a multivariate ARMA model is invertible if all the roots of $|\Theta(x)| = 0$ are outside the unit circle. The invertibility of a model can be checked using `InvertibleQ` as in the univariate case.

Example 2.13 Check if the bivariate MA(2) model is invertible where $\Theta_1 = \{\{-1.2, 0.5\}, \{-0.3, 0.87\}\}$, and $\Theta_2 = \{\{0.2, 0.76\}, \{1.1, -0.8\}\}$.

The model is not invertible.

```
In[37]:= InvertibleQ[MAModel[
           {{{-1.2, 0.5}, {-0.3, 0.87}}, {{0.2, 0.76}, {1.1, -0.8}}}, {{1, 0}, {0, 1}}]]

Out[37]= False
```

A stationary ARMA model can be expanded as an MA(∞) model $\mathbf{X}_t = \sum_{j=0}^{\infty} \Psi_j \mathbf{Z}_{t-j}$ with $\Psi(B) = \Phi^{-1}(B)\,\Theta(B)$; similarly, an invertible ARMA model can be expressed as an AR(∞) model $\sum_{j=0}^{\infty} \Pi_j \mathbf{X}_{t-j} = \mathbf{Z}_t$ with $\Pi(B) = \Theta^{-1}(B)\,\Phi(B)$. Again, the function `ToARModel[`*model*`, p]` can be used to obtain the order p truncation of an AR(∞) expansion. Similarly `ToMAModel[`*model*`, q]` yields the order q truncation of an MA(∞) expansion. $\{\Psi_j\}$ and $\{\Pi_j\}$ are matrices and they are determined by equating the coefficients of corresponding powers of B in $\Phi(B)\,\Psi(B) = \Theta(B)$ and $\Phi(B) = \Theta(B)\,\Pi(B)$, respectively.

Example 2.14 Expand the following AR(1) model as an approximate MA(3) model.

This expands the given AR(1) as an MA(3) model.

```
In[38]:= ToMAModel[ARModel[{{{0.2, 0}, {-0.1, 0.8}}}, {{1, 0}, {0, 1}}], 3]

Out[38]= MAModel[{{{0.2, 0.}, {-0.1, 0.8}}, {{0.04, 0.}, {-0.1, 0.64}},
           {{0.008, 0.}, {-0.084, 0.512}}}, {{1, 0}, {0, 1}}]
```

Covariance and Correlation Functions

The matrix covariance function of a stationary multivariate process is defined by

$$\Gamma(k) = E\,(\mathbf{X}_{t+k} - \mu)\,(\mathbf{X}_t - \mu)'.$$

Note that now $\Gamma(k) \neq \Gamma(-k)$; instead we have $\Gamma(k) = \Gamma(-k)'$. It is easy to see that the i^{th} diagonal element of $\Gamma(k)$, $\Gamma((k))_{ii} \equiv \gamma_{ii}(k) = E\,X_{t+k,i}\,X_{t\,i}$, is simply the (auto)covariance of the univariate time series $\{X_{t\,i}\}$ and $\Gamma((k))_{ij} \equiv \gamma_{ij}(k) = E\,X_{t+k,i}\,X_{t\,j}$ is the cross-covariance of the series $\{X_{t\,i}\}$ and $\{X_{t\,j}\}$. The matrix correlation function $R(k)$ is defined by

$$R((k))_{ij} = \rho_{ij}(k) = \gamma_{ij}(k) \Big/ \sqrt{\gamma_{ii}(0)\,\gamma_{jj}(0)}\,.$$

Note that unlike the univariate case, we cannot simply divide $\Gamma(k)$ by $\Gamma(0)$ to get the correlation function.

We can get the covariance or correlation function of a multivariate ARMA model up to lag h simply by using `CovarianceFunction[`*model*`, h]` or `CorrelationFunction[`*model*`, h]`; the output consists of a list of matrices $\{\Gamma(0), \Gamma(1), \ldots \Gamma(h)\}$ or $\{R(0), R(1), \ldots, R(h)\}$.

Example 2.15 Find the covariance function of a bivariate MA(1) model up to lag 3.

This gives the covariance function of a bivariate MA(1) model up to lag 3.

```
In[39]:= cov = CovarianceFunction[MAModel[{{{t1, t2}, {t3, t4}}}, IdentityMatrix[2]], 3]

Out[39]= {{{1 + t1^2 + t2^2, t1 t3 + t2 t4}, {t1 t3 + t2 t4, 1 + t3^2 + t4^2}},
           {{t1, t2}, {t3, t4}}, {{0, 0}, {0, 0}}, {{0, 0}, {0, 0}}}
```

These are covariance matrices $\{\Gamma(0), \Gamma(1), \Gamma(2), \Gamma(3)\}$ and they can be put into a more readable form using `TableForm`.

The covariance function is displayed in a table form.

```
In[40]:= TableForm[Transpose[{Table[gamma[i], {i, 0, 3}], cov}]]

Out[40]//TableForm=
gamma[0]   1 + t1^2 + t2^2   t1 t3 + t2 t4
           t1 t3 + t2 t4     1 + t3^2 + t4^2
gamma[1]   t1  t2
           t3  t4
gamma[2]   0  0
           0  0
gamma[3]   0  0
           0  0
```

Again, the covariance function vanishes for lags greater than q (=1 in this case).

Often we want to get the cross-covariance or cross-correlation of two processes i and j by extracting the (i, j) element of each matrix in the output of `CovarianceFunction` or `CorrelationFunction`, respectively. Whenever we want to do the same operation on each entry of a list, we can use the *Mathematica* function `Map`. For example, to get the cross-covariance function of processes 1 and 2, $\gamma_{12}(k)$, for $k \geq 0$ in the above bivariate MA(1) model, we can do the following.

We extract the cross-covariance function using `Map`.

```
In[41]:= cov[[All, 1, 2]]

Out[41]= {t1 t3 + t2 t4, t2, 0, 0}
```

Similarly, we can get the autocovariance of process 2 of the bivariate MA model in Example 2.15 by extracting the (2, 2) element of each covariance matrix.

We use `Map` again to extract the covariance function of process 2.

```
In[42]:= cov[[All, 2, 2]]

Out[42]= {1 + t3^2 + t4^2, t4, 0, 0}
```

Example 2.16 Compute the cross-correlation of a bivariate ARMA(1, 1) process and plot it.

The correlation function of the given ARMA(1, 1) process is computed. Recall that the semicolon is used at the end of an expression to suppress the display of the output.

```
In[43]:= corr = CorrelationFunction[ARMAModel[{{{0.5, -0.9}, {1.1, -0.7}}},
              {{{0.4, -0.8}, {1.1, -0.3}}}, {{1, 0.5}, {0.5, 1.2}}], 20];
```

Notice that AR and MA parameter matrices are enclosed in separate lists and the noise variance is a symmetric positive-definite matrix. First we extract the cross-correlation $\gamma_{12}(k)$ and then plot it.

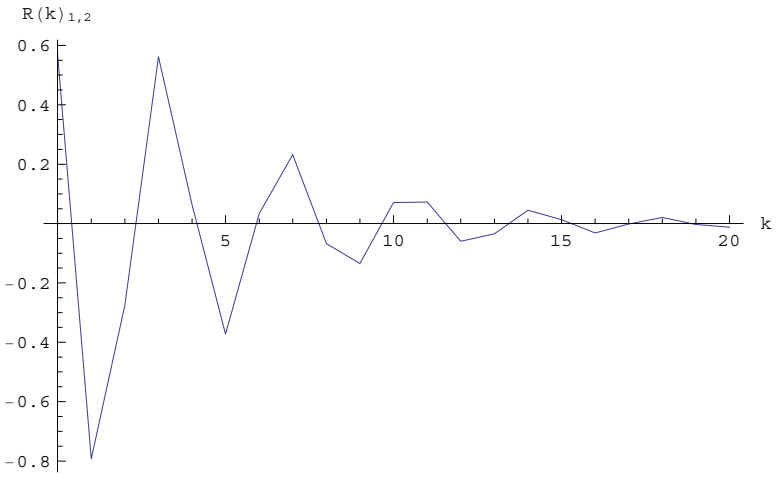
This extracts the cross-correlation function.

```
In[44]:= %[[All, 1, 2]]

Out[44]= {0.573964, -0.792494, -0.27461, 0.562118, 0.0633266, -0.372421, 0.0339551, 0.231558,
          -0.0680429, -0.134589, 0.0704652, 0.0720437, -0.0595065, -0.0342067, 0.0449255,
          0.0129072, -0.0313338, -0.00199385, 0.0204524, -0.00281441, -0.0125266}
```

Here is the cross-correlation plot. The option `PlotRange` is set to `All` to prevent the truncation of any large values in the plot.

```
In[45]:= plotcorr[%, Joined -> True, PlotRange -> All, AxesLabel -> {"k", "R(k)1,2"}]
```

Out[45]=



In contrast to autocorrelation, the cross-correlation at lag 0 is, in general, not equal to 1. Note that we have plotted the cross-correlation only for $k \geq 0$. Here we demonstrate how to plot the cross-correlation function from lags $-h$ to h. First we must obtain $\{\gamma_{12}(-h), \gamma_{12}(-h+1), \ldots, \gamma_{12}(0), \gamma_{12}(1), \ldots, \gamma_{12}(h)\}$. This can be accomplished using the fact that $\gamma_{12}(-k) = \gamma_{21}(k)$.

This gives the cross-correlations from lag -20 to lag 20. `Drop[`*list*`, `*n*`]` gives *list* with its first *n* elements dropped and `Drop[`*list*`, -`*n*`]` gives *list* with its last *n* elements dropped.

```
In[46]:= gamma12 = Join[Reverse[corr[[All, 2, 1]]], Drop[corr[[All, 1, 2]], 1]];
```

Using `gamma12` as an example, it is convenient to define a function, say, `plotmulticorr`, to plot multivariate correlations. The argument should include the multivariate correlation function and i and j indicating which cross-correlation or autocorrelation is to be plotted.

This defines the function `plotmulticorr`.

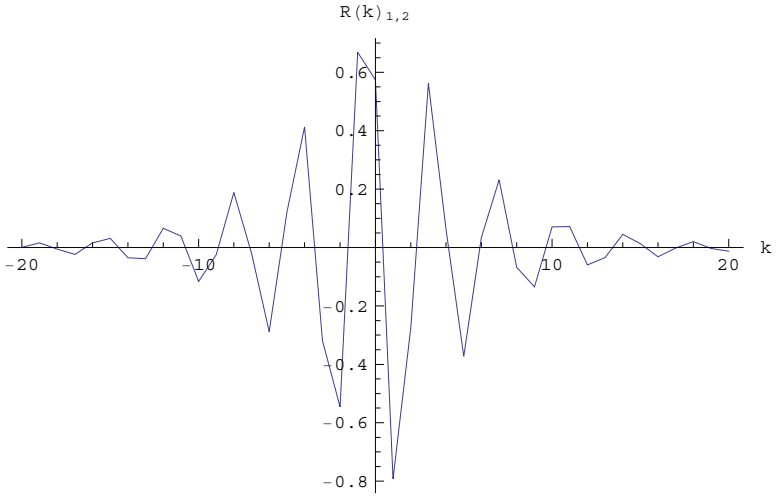
```
In[47]:= plotmulticorr[corr_, i_, j_, opts___] :=
          ListPlot[Join[Reverse[corr[[All, j, i]]], Drop[corr[[All, i, j]], 1]],
           DataRange -> ({-1, 1} (Length[corr] - 1)), opts]
```

Now we can plot the cross-correlation function of the above example using `plotmulticorr`.

This plots the same cross-correlation function but now from lag −20 to 20. Note that it is not symmetric about the origin.

```
In[48]:= plotmulticorr[corr, 1, 2, Joined -> True,
          PlotRange -> All, AxesLabel -> {"k", "R(k)1,2"}]
```

Out[48]=

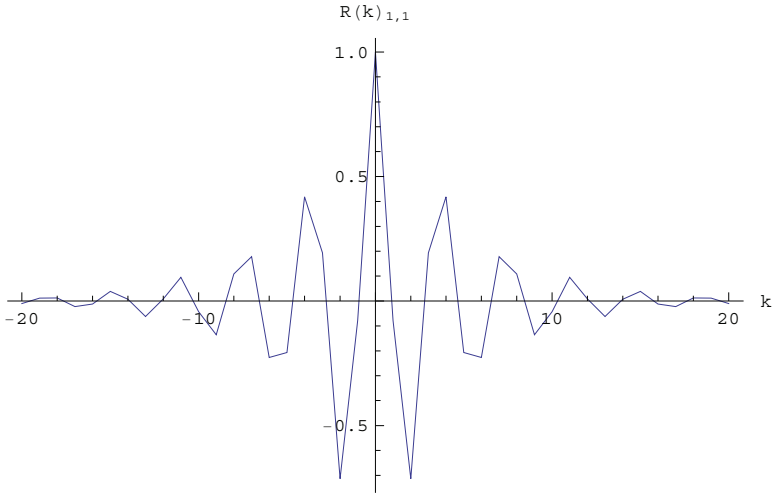


We can also plot the correlation of series 1 of the bivariate ARMA(1, 1) process of Example 2.16 for lags from −20 to 20 using `plotmulticorr`.

This plots the correlation function of series 1. In contrast to the plot of the cross-correlation function, the graph is, of course, symmetric.

```
In[49]:= plotmulticorr[corr, 1, 1, Joined -> True,
          PlotRange -> All, AxesLabel -> {"k", "R(k)1,1"}]
```

Out[49]=



Partial Correlation Function

The direct extension of the partial correlation function to the multivariate case leads to what is often called *partial autoregressive matrices*. The partial autoregressive matrix at lag k is the solution $\Phi_{k,k}$ to the Yule-Walker equations of order k. (See Section 1.6 for a description of Yule-Walker equations and the Levinson-Durbin algorithm.) However, here we will refer to them as the partial correlation function and use `PartialCorrelationFunction[model, h]` to obtain these matrices up to lag h, but bear in mind that some authors define partial correlation function for a multivariate process differently, for example, Granger and Newbold (1986), p. 246.

Example 2.17 Find the partial correlation function (partial autoregressive matrices) of an AR(2) model.

This gives the partial correlation function of an AR(2) model.

```
In[50]:= PartialCorrelationFunction[
          ARModel[{{{0.4, -0.6}, {0.5, 0.2}}, {{0.6, 0.4}, {0.2, -0.6}}},
           {{1, -0.2}, {-0.2, 0.8}}], 3]

Out[50]= {{{0.724501, -0.152084}, {0.660539, 0.143961}},
          {{0.6, 0.4}, {0.2, -0.6}}, {{0, 0}, {0, 0}}}
```

Note again that the partial correlation function vanishes for lags greater than p.

1.3 Nonstationary and Seasonal Models

In this section we first introduce a special class of nonstationary ARMA processes called the *autoregressive integrated moving average (ARIMA) process*. Then we define seasonal ARIMA (SARIMA) processes. After presenting the objects that define these processes we proceed to illustrate how the various functions introduced in Section 1.2 in the context of ARMA models can be applied directly to ARIMA and SARIMA models. The function that converts them to ARMA models is also introduced.

1.3.1 ARIMA Process

When the ARMA model $\phi(B)\, X_t = \theta(B)\, Z_t$ is not stationary, the equation $\phi(x) = 0$ (or $|\Phi(x)| = 0$ in the multivariate case) will have at least one root inside or on the unit circle. In this case, the methods of analyzing stationary time series cannot be used directly. However, the stationary ARMA models introduced in Section 1.2 can be generalized to incorporate a special class of nonstationary time series models. This class of models is characterized by all the zeros of the AR polynomial being outside the unit circle with the exception of d of them which are 1. In other words, this class of nonstationary models is defined by

$$(1 - B)^d\, \phi(B)\, X_t = \theta(B)\, Z_t, \tag{3.1}$$

where d is a non-negative integer, $\phi(x)$ and $\theta(x)$ are polynomials of degrees p and q, respectively, and all the roots of $\phi(x) = 0$ are outside the unit circle. Equation (3.1) defines an autoregressive integrated moving average process of orders p, d, q, or simply, ARIMA(p, d, q).

Using the definition of the backward shift operator B, we have $(1 - B)\, X_t = X_t - X_{t-1}$. This operation is for obvious reasons called *differencing* the time series. (We use $(1 - B)^2\, X_t = (1 - B)\,(X_t - X_{t-1}) = X_t - 2\, X_{t-1} + X_{t-2}$ to difference the time series twice.) Equation (3.1) says that if $\{X_t\}$ is nonstationary and satisfies (3.1), then after differencing the time series d times the differenced series $\{Y_t\}$ ($Y_t = (1 - B)^d\, X_t$) is stationary and satisfies $\phi(B)\, Y_t = \theta(B)\, Z_t$, that is, an ARMA($p$, q) process. Note that we can view $\{Y_t\}$ as a filtered version of $\{X_t\}$ (see Section 1.4.3).

Therefore, any ARIMA(p, d, q) series can be transformed into an ARMA(p, q) series by differencing it d times and, thus, the analysis of an ARIMA process does not pose any special difficulty as long as we know the number of times to difference (*i.e.*, d) the series. We will see in Section 1.4.3 how the differencing is done in practice.

An ARIMA(p, d, q) model is represented by the object

$$\texttt{ARIMAModel}[d, \{\phi_1, \phi_2, \ldots, \phi_p\}, \{\theta_1, \theta_2, \ldots, \theta_q\}, \sigma^2].$$

An ARIMA(p, 0, q) process is simply an ARMA(p, q) process.

1.3.2 Seasonal ARIMA Process

Sometimes there can be seasonal or cyclic components in a time series. By this we mean the recurrence of some recognizable pattern after some regular interval that we call the *seasonal period* and denote by s. For example, in the monthly data of international airline passengers there is clearly a recurring pattern with a seasonal period of 12.

A pure seasonal model is characterized by nonzero correlations only at lags that are multiples of the seasonal period s. This means that the time series at time t, X_t, depends on X_{t-s}, X_{t-2s}, X_{t-3s}, … only. In general, we can define a pure seasonal ARMA model of orders P and Q and of seasonal period s by

$$X_t - \Phi_1 X_{t-s} - \Phi_2 X_{t-2s} - \ldots - \Phi_P X_{t-Ps} = Z_t + \Theta_1 Z_{t-s} + \ldots + \Theta_Q Z_{t-Qs}. \tag{3.2}$$

If we define the seasonal AR polynomial $\Phi(x^s)$ as

$$\Phi(x^s) = 1 - \Phi_1 x^s - \Phi_2 x^{2s} - \ldots - \Phi_P x^{Ps}$$

and the seasonal MA polynomial $\Theta(x^s)$ as

$$\Theta(x^s) = 1 + \Theta_1 x^s + \Theta_2 x^{2s} + \ldots + \Theta_Q x^{Qs},$$

(3.2) can be rendered more compactly using the backward shift operator B as

$$\Phi(B^s) X_t = \Theta(B^s) Z_t.$$

(Note that although we use the same notation Φ and Θ for seasonal model parameters as for multivariate ARMA model parameters, their meaning should be clear from the context.)

The pure seasonal models defined by (3.2) are often not very realistic since they are completely decoupled from each other. That is, (3.2) represents s identical but separate models for X_t, X_{t+1}, … , X_{t+s-1}. In reality, of course, few time series are purely seasonal and we need to take into account the interactions or correlations between the time series values within each period. This can be done by combining the seasonal and regular effects into a single model. A multiplicative seasonal ARMA model of seasonal period s and of seasonal orders P and Q and regular orders p and q is defined by

$$\phi(B)\,\Phi(B^s)\,X_t = \theta(B)\,\Theta(B^s)\,Z_t. \tag{3.3}$$

Here $\phi(x)$ and $\theta(x)$ are regular AR and MA polynomials defined in (2.4) and (2.5).

To generalize the model defined by (3.3) to include nonstationary cases we define the seasonal difference to be $(1 - B^s)\,X_t = X_t - X_{t-s}$. A *multiplicative seasonal autoregressive integrated moving average (SARIMA) process* of period s, with regular and seasonal AR orders p and P, regular and seasonal MA orders q and Q, and regular and seasonal differences d and D is defined by

$$(1 - B)^d\,(1 - B^s)^D\,\phi(B)\,\Phi(B^s)\,X_t = \theta(B)\,\Theta(B^s)\,Z_t. \tag{3.4}$$

We will use SARIMA$(p, d, q)(P, D, Q)_s$ to refer to the model defined by (3.4). In typical applications, $D = 1$ and P and Q are small.

A SARIMA$(p, d, q)(P, D, Q)_s$ model is represented by the object

$$\texttt{SARIMAModel}[\{d, D\}, s, \{\phi_1, \ldots, \phi_p\}, \{\Phi_1, \ldots, \Phi_p\}, \{\theta_1, \ldots, \theta_q\},$$
$$\{\Theta_1, \ldots, \Theta_Q\}, \sigma^2].$$

Note that a pure seasonal model (3.2) and a seasonal ARMA model (3.3) are special cases of the SARIMA model, and we do not use different objects to represent them separately. If a particular order is zero, for example, $p = 0$, the parameter list $\{\phi_1, \ldots, \phi_p\}$ is {} or {0}. For example, `ARMAModel[{},{θ}, 1]` is the same as `ARMAModel[{0},{θ},1]` and as `MAModel[{θ}, 1]`.

Any SARIMA process can be thought of as a particular case of a general ARMA process. This is because we can expand the polynomials on both sides of (3.4) and obtain an ARMA$(p + P\,s + d + s\,D,\ q + s\,Q)$ model whose ARMA coefficients satisfy certain relationships. Similarly, an ARIMA model defined in (3.1) is a special case of the ARMA$(p + d,\ q)$ model. If we wish to expand a SARIMA or an ARIMA model as an equivalent ARMA model, we can invoke the function

$$\texttt{ToARMAModel}[\textit{model}].$$

Example 3.1 Convert the ARIMA(1, 3, 2) model $(1 - B)^3 (1 - \phi_1 B) X_t = \left(1 + \theta_1 B + \theta_2 B^2\right) Z_t$ to the equivalent ARMA(4, 2) model.

We load the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

This converts the given ARIMA model to the equivalent ARMA model.

```
In[2]:= ToARMAModel[ARIMAModel[3, {ϕ1}, {θ1, θ2}, σ^2]]
Out[2]= ARMAModel[{3 + ϕ1, -3 - 3 ϕ1, 1 + 3 ϕ1, -ϕ1}, {Θ1, Θ2}, σ^2]
```

Example 3.2 Convert the SARIMA$(1, 2, 1)(1, 0, 1)_3$ model $(1 - B)^2 (1 - \phi_1 B)\left(1 - \Phi_1 B^3\right) X_t = (1 + \theta_1 B)\left(1 + \Theta_1 B^3\right) Z_t$ to the equivalent ARMA model.

This converts the SARIMA model to the equivalent ARMA model.

```
In[3]:= ToARMAModel[SARIMAModel[{2, 0}, 3, {ϕ1}, {Φ1}, {θ1}, {Θ1}, σ^2]]
Out[3]= ARMAModel[{2 + ϕ1, -1 - 2 ϕ1, ϕ1 + Φ1, -2 Φ1 - ϕ1 Φ1, Φ1 + 2 ϕ1 Φ1, -ϕ1 Φ1}, {Θ1, 0, Θ1, Θ1 Θ1}, σ^2]
```

This is an ARMA(6, 4) model. In contrast to an arbitrary ARMA(6, 4) model, this model has only four independent ARMA parameters.

The functions introduced earlier for ARMA models are applicable to ARIMA and SARIMA models as well. These functions convert the ARIMA and SARIMA models to ARMA models internally and then compute the desired quantities. In the following we provide a few illustrative examples of the usage.

Example 3.3 Check if the SARIMA$(0, 0, 1)(0, 1, 2)_2$ model

$(1 - B^2) X_t = (1 - 0.5\,B)(1 - 0.5\,B^2 + 0.9\,B^4) Z_t$ is invertible.

This SARIMA model is invertible.

```
In[4]:= InvertibleQ[SARIMAModel[{0, 1}, 2, {0}, {0}, {-0.5}, {-0.5, 0.9}, 1]]

Out[4]= True
```

Example 3.4 Expand the above SARIMA model as an approximate AR(7) model.

This expands the SARIMA model as an AR(7) model.

```
In[5]:= ToARModel[SARIMAModel[{0, 1}, 2, {0}, {0}, {-0.5}, {-0.5, 0.9}, 1], 7]

Out[5]= ARModel[{-0.5, 0.25, 0.125, 1.2125, 0.60625, 0.428125, 0.214063}, 1]
```

Since the covariance or correlation function is defined only for stationary models, any attempt to calculate the covariance or correlation function of a SARIMA or an ARIMA model with $d \neq 0$ or $D \neq 0$ will fail.

The model is nonstationary and no correlation is calculated.

```
In[6]:= CorrelationFunction[SARIMAModel[{0, 1}, 12, {0}, {0.5}, {0.8}, {0}, 1], 5]

        CovarianceFunction::nonst :
         The model SARIMAModel[{0, 1}, 12, {0}, {0.5}, {0.8}, {0}, 1] is not stationary.

Out[6]= CorrelationFunction[SARIMAModel[{0, 1}, 12, {0}, {0.5}, {0.8}, {0}, 1], 5]
```

Example 3.5 Find the covariance function of a pure seasonal model $(1 - \Phi_1 B^3) X_t = (1 + \Theta_1 B^3) Z_t$.

This is the covariance function of a pure seasonal model.

```
In[7]:= CovarianceFunction[SARIMAModel[{0, 0}, 3, {}, {Φ1}, {}, {Θ1}, σ^2], 7]
```

$$\text{Out}[7]= \left\{-\frac{\sigma^2\left(1+\Theta_1^2+2\,\Theta_1\,\Phi_1\right)}{-1+\Phi_1^2},\ 0,\ 0,\right.$$
$$\left.-\frac{\sigma^2\left(\Phi_1+\Theta_1^2\,\Phi_1+\Theta_1\left(1+\Phi_1^2\right)\right)}{-1+\Phi_1^2},\ 0,\ 0,\ -\frac{\sigma^2\,\Phi_1\left(\Phi_1+\Theta_1^2\,\Phi_1+\Theta_1\left(1+\Phi_1^2\right)\right)}{-1+\Phi_1^2},\ 0\right\}$$

As expected, the covariance function of a pure seasonal model is nonzero only at lags that are multiples of the seasonal period s. The values of the covariance function at lags 0, s, $2s$, … are the same as those of a regular ARMA(P, Q) model with the same parameters. In fact, the two models are identical if we rescale time by a factor s in the pure seasonal model. In our current example, the values of the covariance function at lags 0, 3, 6, … are the same as those at lags 0, 1, 2, … of the regular ARMA(1, 1) model $(1 - \Phi_1 B) X_t = (1 + \Theta_1 B) Z_t$.

This is the covariance function of the regular ARMA(1, 1) model. Compare it with the previous expression.

```
In[8]:= CovarianceFunction[ARMAModel[{Φ1}, {Θ1}, σ^2], 2]
```

$$\text{Out[8]}= \left\{-\frac{\sigma^2\left(1+\Theta_1^2+2\,\Theta_1\,\Phi_1\right)}{-1+\Phi_1^2},\ -\frac{\sigma^2\left(\Phi_1+\Theta_1^2\,\Phi_1+\Theta_1\left(1+\Phi_1^2\right)\right)}{-1+\Phi_1^2},\ -\frac{\sigma^2\,\Phi_1\left(\Phi_1+\Theta_1^2\,\Phi_1+\Theta_1\left(1+\Phi_1^2\right)\right)}{-1+\Phi_1^2}\right\}$$

Since a SARIMA$(p, 0, q)(P, 0, Q)_s$ model is an ARMA$(p + s\,P, q + s\,Q)$ model, all the properties of the correlation and partial correlation functions we discussed in Sections 1.2.3 and 1.2.4 remain true. For example, the partial correlation function, $\phi_{k,k}$, of a SARIMA$(p, 0, 0)(P, 0, 0)_s$ model vanishes for $k > p + s\,P$, and the correlation function, $\rho(k)$, of a SARIMA$(0, 0, q)(0, 0, Q)_s$ model vanishes for $k > q + s\,Q$. However, in some special cases we can say more about the correlation function of a seasonal ARMA model due to the relationships that exist between the coefficients. For example, the $(q + s\,Q)^{\text{th}}$ degree polynomial $\theta(B)\,\Theta(B^s)$ can be expanded as

$$\theta(B)\,\Theta(B^s) = \theta(B) + \Theta_1\,B^s\,\theta(B) + \Theta_2\,B^{2\,s}\,\theta(B) + \ldots + \Theta_Q\,B^{Q\,s}\,\theta(B).$$

It is clear from the above expression that if $s > q + 1$ there are "gaps" in the above polynomial, that is, terms from B^{q+1} to B^{s-1}, from B^{s+q+1} to $B^{2\,s-1}$, … are absent. Now consider a seasonal ARMA model with $p = P = 0$. The covariance function is given by

$$\gamma(k) = E(\theta(B)\,\Theta(B^s)\,Z_t\,\theta(B)\,\Theta(B^s)\,Z_{t-k}). \tag{3.5}$$

If these "gaps" are large enough, for some values of k the covariance function $\gamma(k)$ vanishes simply because there is no overlap between the polynomials on the right-hand side of (3.5). In fact, if the "gap" $(s-1)-(q+1)+1$ is larger than q or $s \geq 2\,(q+1)$, we have $\gamma(k) = 0$ for $q < k < s - q$, $s + q < k < 2\,s - q$, … , $(Q-1)\,s + q < k < Q\,s - q$, and, of course, we always have $\gamma(k) = 0$ for $k > s\,Q + q$.

It is also easy to show from (3.5) that as long as "gaps" exist (*i.e.*, $s > q + 1$) in the expansion of the MA polynomials, the covariance function is symmetric about the lags that are multiples of the seasonal period. In other words, $\gamma(s - i) = \gamma(s + i)$, $\gamma(2\,s - i) = \gamma(2\,s + i)$, … for $i = 1, 2, \ldots, q$.

Example 3.6 Find the correlation function of the SARIMA$(0, 0, 1)(0, 0, 2)_6$ model with $\theta_1 = 0.9$, $\Theta_1 = 0.6$, and $\Theta_2 = 0.5$.

Since $p = P = 0$ and $s = 6 \geq 2\,(q + 1) = 4$, we expect the correlation function has the properties described above.

This calculates the correlation function up to lag 20 from the SARIMA model.

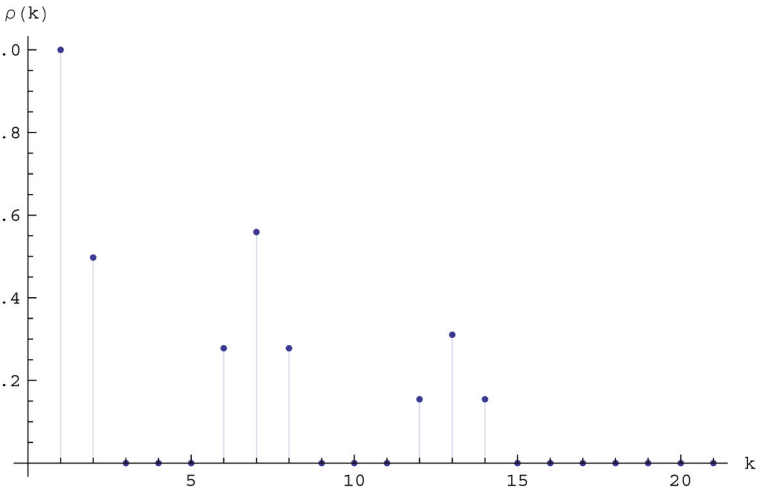
```
In[9]:= corr = CorrelationFunction[SARIMAModel[{0, 0}, 6, {}, {}, {0.9}, {0.6, 0.5}, 1], 20]

Out[9]= {1., 0.497238, 0., 0., 0., 0.277959, 0.559006, 0.277959, 0.,
          0., 0., 0.154422, 0.310559, 0.154422, 0., 0., 0., 0., 0., 0., 0.}
```

We observe that (a) $\rho(k) = 0$ for $k > s\,Q + q = 13$; (b) $\rho(k) = 0$ for $q = 1 < k < 5 = s - q$ and $s + q = 7 < k < 11 = 2\,s - q$; and (c) $\gamma(5) = \gamma(7)$, $\gamma(11) = \gamma(13)$. We plot this correlation function.

Here is the plot of the correlation function.

```
In[10]:= ListPlot[corr, Filling -> Axis, PlotRange -> All, AxesLabel -> {"k", "ρ(k)"}]
```

Out[10]=



The symmetry of the correlation $\gamma(k)$ about the lags that are multiples of the seasonal period s leads directly to the conclusion that at lags ks (k integer) the correlations are in general symmetric local extrema. This is seen clearly in the above graph.

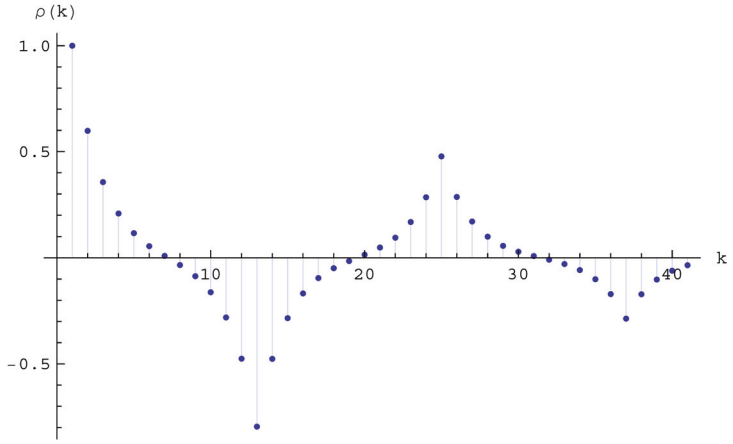
The properties derived above for the correlation function of seasonal ARMA models with $p = P = 0$ will not hold for general seasonal ARMA models. However, for some low-order seasonal models, which are often what we encounter in practice, the correlations at lags that are multiples of the seasonal period tend to be local extrema as shown in the following example. It is helpful to keep this in mind when identifying a seasonal model although fluctuations often wash out all but the very pronounced local extrema in practice. (See Section 1.5 on model identification.)

Example 3.7 Find the correlation function of the SARIMA$(1, 0, 0)(1, 0, 1)_{12}$ model with $\phi_1 = 0.6$, $\Phi_1 = -0.6$, and $\Theta_1 = -0.8$.

This is the plot of the correlation function.

```
In[11]:= ListPlot[
          CorrelationFunction[SARIMAModel[{0, 0}, 12, {0.6}, {-0.6}, {}, {-0.8}, 1], 40],
          Filling -> Axis, PlotRange -> All, AxesLabel -> {"k", "ρ(k)"}]
```

Out[11]=



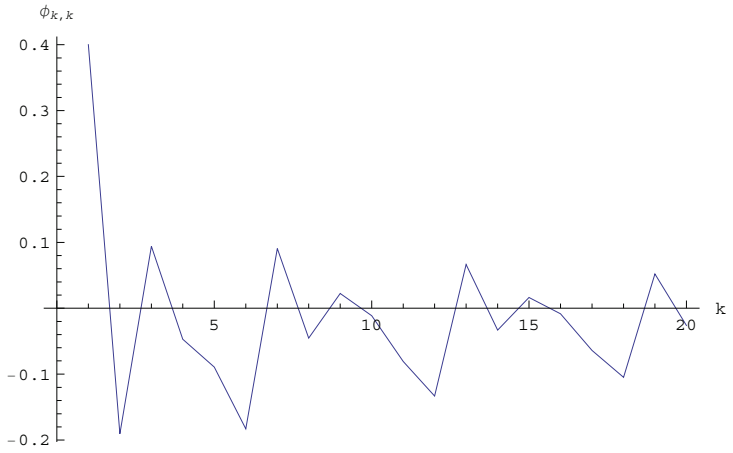
The calculation of the partial correlation function of a SARIMA model proceeds as in the case of an ARMA model. Again for $q = Q = 0$, the partial correlation function $\phi_{k,k} = 0$ for $k > p + s\,P$.

Example 3.8 Find the partial correlation function of a SARIMA model.

This is the plot of the partial correlation function of a SARIMA model.

```
In[12]:= ListLinePlot[
          PartialCorrelationFunction[SARIMAModel[{0, 0}, 6, {}, {0.5}, {0.5}, {-1.2}, 1],
           20], AxesLabel -> {"k", "ϕk,k"}]
```

Out[12]=



Multivariate ARIMA and SARIMA models are again defined by (3.1) and (3.4), respectively, with $\phi(B)$, $\Phi(B^s)$, $\theta(B)$, and $\Theta(B^s)$ being matrix polynomials. All the functions illustrated above can be used for multivariate models; however, we will not demonstrate them here.

1.4 Preparing Data for Modeling

In Sections 1.2 and 1.3 we introduced some commonly used stochastic time series models. In this section we turn our attention to actual time series data. These data can be obtained from real experiments or observations over time or generated from numerical simulations of specified time series models. We consider these data to be particular realizations of stochastic processes. Although we call both the stochastic process and its realization time series, we distinguish between them by using lower-case letters to denote the actual data and the corresponding upper-case letters to denote the random variables.

Several ways of transforming the raw data into a form suitable for modeling are presented in this section. These transformations include linear filtering, simple exponential smoothing, differencing, moving average, and the Box-Cox transformation. We demonstrate how to generate normally distributed random sequences and time series from specified models and also show how to read in data from a file and plot them.

1.4.1 Plotting the Data

The first thing to do in analyzing time series data is to plot them since visual inspection of the graph can provide the first clues to the nature of the series: we can "spot" trends, seasonality, and nonstationary effects. Often the data are stored in a file and we need to read in the data from the file and put them in the appropriate format for plotting using *Mathematica*. We provide several examples below.

> **Example 4.1** As an illustrative example of how to read in data from a file, let us suppose that we have a file called `file1.dat` in the directory `TimeSeries/Data`. (Note that specification of files and directories depends on the system being used.) The file consists of two columns of numbers. The numbers in the first column are the times when the observations were made and those in the second column are the outcomes of observations, that is, the time series. We can look at the contents of `file1.dat` using `FilePrint`.

We load the package first.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

This displays the contents of the file `file1.dat` in the directory `TimeSeries/Data`.

```
In[2]:= FilePrint[ToFileName[{"TimeSeries", "Data"}, "file1.dat"]]
```

```
0.5     0.091
1.      0.595
1.5     0.572
2.      0.291
2.5     0.865
3.      0.623
3.5     0.412
4.      0.934
4.5     0.657
5.      0.723
```

We use the `ReadList` command to read in these numbers and put them in a list.

We read in the time series data from the file `file1.dat`. The specification {`Number`, `Number`} in `ReadList` makes each entry in the list a list of two numbers $\{t, x_t\}$.

```
In[3]:= data1 = ReadList[ToFileName[{"TimeSeries", "Data"}, "file1.dat"], {Number, Number}]
```

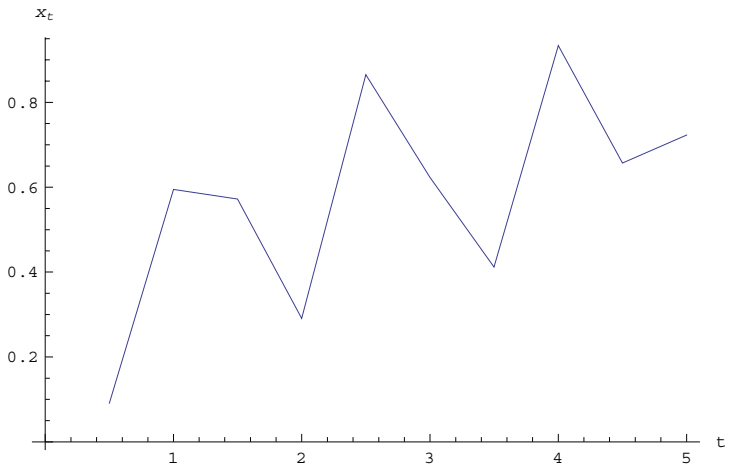
```
Out[3]= {{0.5, 0.091}, {1., 0.595}, {1.5, 0.572}, {2., 0.291}, {2.5, 0.865},
  {3., 0.623}, {3.5, 0.412}, {4., 0.934}, {4.5, 0.657}, {5., 0.723}}
```

Now `data1` defined above is in the right format for `ListLinePlot`.

We plot the data using `ListLinePlot`.

```
In[4]:= ListLinePlot[data1, AxesLabel -> {"t", "x_t"}]
```

Out[4]=



We can check if the data were in fact taken at equally spaced intervals by doing the following. First extract the time coordinates.

The time coordinates are the first element of each data point.

```
In[5]:= tcoord = data1[[All, 1]]
```

```
Out[5]= {0.5, 1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5, 5.}
```

Now take the differences of adjacent time coordinates and see if they give the same number.

The differences are the same.

```
In[6]:= Union[Drop[tcoord, 1] - Drop[tcoord, -1]]

Out[6]= {0.5}
```

So we see that in this case the data were indeed taken at constant time intervals. Since we have assumed all along that the data are taken at equally spaced time intervals, it is often convenient to drop the time coordinate and consider just the time series. All time series data to be input to time series functions should be in the form $\{x_1, x_2, \ldots, x_n\}$ where x_i is a number for a scalar time series and a list, $x_i = \{x_{i1}, x_{i2}, \ldots, x_{im}\}$, for an m-variate time series.

We can cast the above data in the form appropriate for input to time series functions by taking the second (*i.e.*, the last) entry from each data point.

This extracts the time series.

```
In[7]:= data = data1[[All, 2]]

Out[7]= {0.091, 0.595, 0.572, 0.291, 0.865, 0.623, 0.412, 0.934, 0.657, 0.723}
```

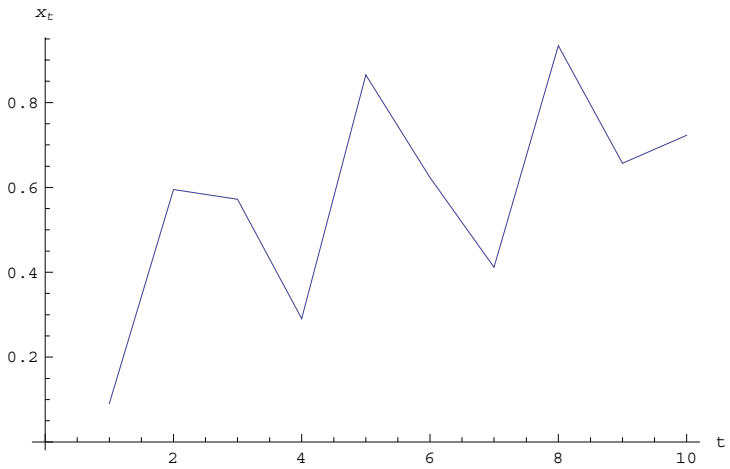
Now the data are in the right form to be used in time series functions.

If we plot the above data using `ListLinePlot` we will get the same plot as before except (a) the origin of time will have been shifted and (b) it will be in different time units with the first entry in data corresponding to time 1, and the second to time 2, and so on.

Here is the time plot for the same series.

```
In[8]:= ListLinePlot[data, AxesLabel -> {"t", "x_t"}]
```

Out[8]=



Example 4.2 Often the time series data are stored in a file without the time coordinates, as in the case of the lynx data in the file `lynx.dat` in the directory `TimeSeries/Data`. The file contains a series of numbers separated by blanks.

This reads in the contents of the file `lynx.dat` in the directory `TimeSeries/Data`.

```
In[9]:= lynxdata = ReadList[ToFileName[{"TimeSeries", "Data"}, "lynx.dat"], Number];
```

It is convenient to use the *Mathematica* function `Short` to find out if the data are in the appropriate format without printing out all the numbers.

This gives the short form of `lynxdata`.

```
In[10]:= Short[lynxdata, 4]

Out[10]//Short=
     {269, 321, 585, 871, 1475, 2821, 3928, 5943, 4950, 2577,
      523, 98, 184, 279, 409, 2285, 2685, <<80>>, 81, 80, 108, 229, 399,
      1132, 2432, 3574, 2935, 1537, 529, 485, 662, 1000, 1590, 2657, 3396}
```

We can plot `lynxdata` directly using `ListLinePlot` as we have demonstrated in the previous plot. On the other hand, if we want the time plot to show the real times at which the data were taken, we can reverse the above procedure of transforming `data1` to `data`. Suppose the data were taken from time t_1 to t_n in intervals of Δt, `Range[`t_1`,` t_n`,` *deltat*`]` will generate a list of all the times at which data were taken. If Δt is omitted the default value of 1 is assumed. In our lynx example, the data were taken annually from the year 1821 to the year 1934.

This combines the time series with the time coordinates.

```
In[11]:= data1 = Transpose[{Range[1821, 1934], lynxdata}];
```

We use `Short` to display `data1`.

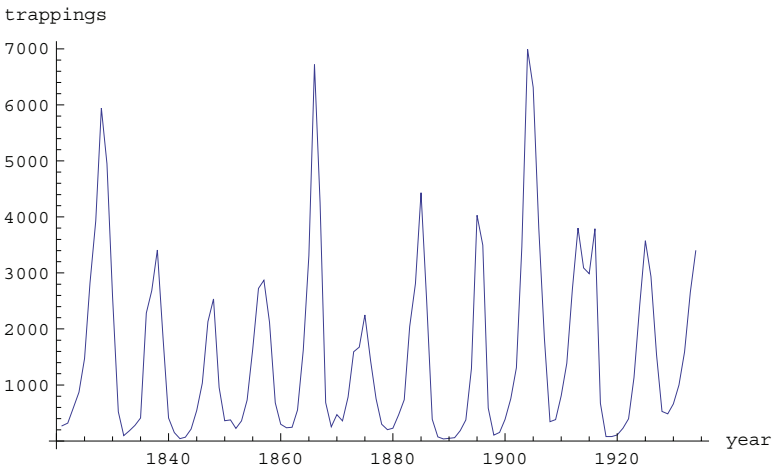
```
In[12]:= Short[data1, 4]

Out[12]//Short=
     {{1821, 269}, {1822, 321}, {1823, 585}, {1824, 871}, {1825, 1475}, {1826, 2821}, <<102>>,
      {1929, 485}, {1930, 662}, {1931, 1000}, {1932, 1590}, {1933, 2657}, {1934, 3396}}
```

Now if we plot `data1` using `ListLinePlot` the horizontal axis corresponds to the "real" time.

Here is the time plot of the lynx data.

```
In[13]:= ListLinePlot[data1, AxesLabel -> {"year", "trappings"}]
```

Out[13]=



Example 4.3 The file `file2.dat` has a set of bivariate data with column 1 containing series 1 and column 2 containing series 2, separated by commas. Again we employ `ReadList` to read the data.

We read in the data from `file2.dat`.

```
In[14]:= data1 = ReadList[ToFileName[{"TimeSeries", "Data"}, "file2.dat"],
            Number, RecordSeparators -> {","}];
```

We have used the option `RecordSeparators -> ","` because of the presence of commas between the numbers. To convert `data1`, which is now a long list of numbers, into the correct time series data format we put every pair of numbers into a list using `Partition`.

We use `Partition` to put the data in the right form for a multivariate series.

```
In[15]:= data = Partition[data1, 2];
```

Here are a few elements of `data`.

```
In[16]:= Take[data, 5]

Out[16]= {{0.830546, 1.66549}, {-0.353715, 3.68722},
          {-0.63938, 6.08595}, {-2.31622, 8.87918}, {-4.08102, 8.14158}}
```

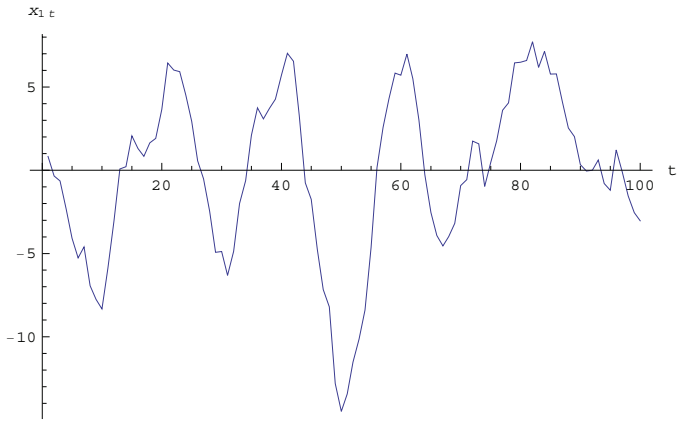
We see that `data` is a bivariate time series of length 100. Again it is in the correct format of the form $\{x_1, x_2, \ldots, x_n\}$ with the i^{th} entry x_i being a list of length 2.

To extract the i^{th} time series from multivariate data we can use *data*`[[All,i]]`. Here are the plots of the two series of the bivariate series `data`.

This is the time plot of series 1 of the bivariate series in `file2.dat`.

```
In[17]:= ListLinePlot[data[[All, 1]], AxesLabel -> {"t", "x1 t"}]
```

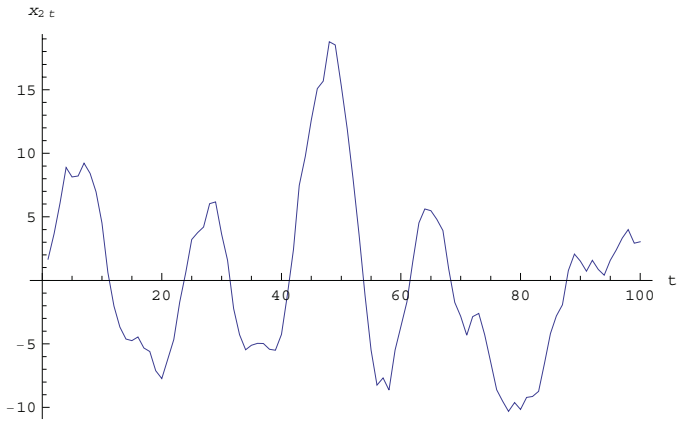
Out[17]=



This is the time plot of series 2 of the bivariate series in `file2.dat`.

```
In[18]:= ListLinePlot[data[[All, 2]], AxesLabel -> {"t", "x2 t"}]
```

Out[18]=



1.4.2 Generating Time Series

Sometimes we may need to generate a time series $\{x_1, x_2, \ldots, x_n\}$ from a given process in order to explore it theoretically or gain intuition about it. The simplest time series is a sequence of independently and identically distributed (IID) random numbers. We can use the function

$$\texttt{RandomSequence}[\mu, \sigma^2, n]$$

to generate a sequence of n random numbers distributed according to the normal distribution with mean μ and variance σ^2.

Example 4.4 Generate a random sequence of length 6 distributed according to the normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$ (*i.e.*, $N(0, 1)$).

This generates a random sequence of length 6.

```
In[19]:= RandomSequence[0, 1, 6]

Out[19]= {-1.85264, -1.81359, -0.922332, 0.963026, 1.80357, -0.06322}
```

It is usually a good idea to seed the random number generator using `SeedRandom[seed]` before generating any random numbers so the calculation can be repeated if necessary.

Example 4.5 Generate a tri-variate random sequence of length 5 distributed normally with mean μ = {0, 1, 0.5} and the covariance matrix Σ = {{1, 0.5, 0}, {0.5, 1.2, 0.8}, {0, 0.8, 1.5}}.

The random number generator is seeded first.

```
In[20]:= SeedRandom[294 587]
```

This generates the random sequence.

```
In[21]:= RandomSequence[{0, 1, 0.5}, {{1, 0.5, 0}, {0.5, 1.2, 0.8}, {0, 0.8, 1.5}}, 5]

Out[21]= {{0.811364, 3.09889, 1.91463},
          {-0.838268, 2.18402, 2.44013}, {0.0939811, -0.127805, -1.28733},
          {0.0562101, 1.40061, 2.57561}, {-0.534473, -0.45115, -0.812483}}
```

Note that we have used `SeedRandom` so the above series can be reproduced if so desired. We reiterate that the covariance matrix must be a symmetric, positive definite matrix. Otherwise no random numbers will be generated.

No random numbers are generated when the covariance matrix is not symmetric or positive definite.

```
In[22]:= RandomSequence[{0, 0}, {{0.5, 0.2}, {0.3, 1}}, 5]

       RandomSequence::symm : Covariance matrix {{0.5, 0.2}, {0.3, 1}} is not symmetric.

Out[22]= RandomSequence[{0, 0}, {{0.5, 0.2}, {0.3, 1}}, 5]
```

Internally, `RandomSequence` uses `RandomReal` to generate a univariate random sequence of length n. The current default random number generator is used. Sequences generated prior to Version 6.0 of *Mathematica* can be obtained by using the legacy generator via `SeedRandom[Method->"Legacy"]` or `SeedRandom[seed, Method->"Legacy"]`.

We can similarly generate univariate random sequences according to the distributions of our choice. Please consult the *Mathematica* documentation for distributions included with *Mathematica*.

To generate a time series of length n, $\{x_1, x_2, \ldots, x_n\}$, according to one of the models defined in earlier sections, we can use the function

`TimeSeries[model, n]`.

It generates the series by iterating

$$x_t = \phi_1 x_{t-1} + \ldots + \phi_p x_{t-p} + z_t + \theta_1 z_{t-1} + \ldots + \theta_q z_{t-q}.$$

The noise $\{z_t\}$ is generated from the normal distribution with zero mean and variance or covariance specified in *model*. For models with $p \neq 0$, p random numbers from the same distribution as the noise are used as the initial values $\{x_{-p+1}, x_{-p+2}, \ldots, x_0\}$. We can also generate time series with specified initial values of the series using

$$\texttt{TimeSeries}[\textit{model}, n, \{x_{-p+1}, x_{-p+2}, \ldots, x_0\}]$$

and with a specified noise sequence $\{z_{-q+1}, z_{-q+2}, \ldots, z_n\}$ using

$$\texttt{TimeSeries}[\textit{model}, \{z_{-q+1}, z_{-q+2}, \ldots, z_n\}, \{x_{-p+1}, x_{-p+2}, \ldots, x_0\}].$$

Note that both the p initial values and the noise sequence are enclosed in lists. For an m-variate series each x_i or z_i is a list of m numbers.

Like `RandomSequence`, `TimeSeries` uses the current default random number generator. Series generated prior to Version 6.0 of *Mathematica* can be obtained by using the legacy generator via `SeedRandom[Method->"Legacy"]` or `SeedRandom[seed, Method->"Legacy"]`.

Example 4.6 Generate a time series of length 10 according to the ARIMA(2, 1, 1) model $(1 - B)\left(1 - 0.5\,B + 0.8\,B^2\right) X_t = (1 - 0.5\,B)\,Z_t$ where the noise variance is 1.2.

The random number generator is seeded first.

```
In[23]:= SeedRandom[93 846]
```

This generates the time series of length 10 from the given ARIMA(2, 1, 1) model.

```
In[24]:= TimeSeries[ARIMAModel[1, {0.5, -0.8}, {-0.5}, 1.2], 10]

Out[24]= {7.42594, 9.02073, 5.46151, 0.68873,
          3.07068, 7.17666, 6.36394, 3.06564, 3.01135, 5.1414}
```

We can also generate a time series with a specified noise sequence.

Here we generate a random sequence of length 9 distributed uniformly on the interval [-1, 1].

```
In[25]:= (SeedRandom[2847];
          RandomReal[{-1, 1}, 9])

Out[25]= {0.5414, -0.272838, -0.673503, 0.306298,
          0.383461, -0.765174, 0.457336, -0.0909457, -0.59793}
```

This generates a time series with the noise sequence generated above. Note that here the noise variance can be omitted in the argument of the model object.

```
In[26]:= TimeSeries[ARMAModel[{0.2, -0.5}, {0.7}], %, {0.5, -0.3}]

Out[26]= {-0.203858, -0.755261, -0.214277, 0.932645, -0.203083, -0.585225, 0.213686, -0.326243}
```

Example 4.7 Generate a bivariate AR(1) series of length 6 with $\Phi_1 = \{\{0.3, -0.4\}, \{0.2, 1.2\}\}$, noise covariance $\Sigma = \{\{0.5, 0.2\}, \{0.2, 1\}\}$, and initial value $x_0 = \{0.5, -0.1\}$.

The random number generator is seeded first.

```
In[27]:= SeedRandom[3857]
```

We generate the required bivariate AR(1) time series of length 6. Note that the initial value x_0 is enclosed in a list.

```
In[28]:= TimeSeries[
          ARModel[{{{0.3, -0.4}, {0.2, 1.2}}}, {{0.5, 0.2}, {0.2, 1}}], 6, {{0.5, -0.1}}]

Out[28]= {{0.546434, -0.217747}, {-0.161597, 0.188712}, {0.248326, -0.319944},
          {0.0985543, 0.0741225}, {0.718557, 1.49708}, {0.171007, 2.33348}}
```

1.4.3 Transformation of Data

In order to fit a time series model to data, we often need to first transform the data to render them "well-behaved". By this we mean that the transformed data can be modeled by a zero-mean, stationary ARMA type of process. We can usually decide if a particular time series is stationary by looking at its time plot. Intuitively, a time series "looks" stationary if the time plot of the series appears "similar" at different points along the time axis. Any nonconstant mean or variability should be removed before modeling. In this section we introduce a variety of methods of transforming the data.

Example 4.8 The time plot of the data contained in the file `file3.dat` is shown below.

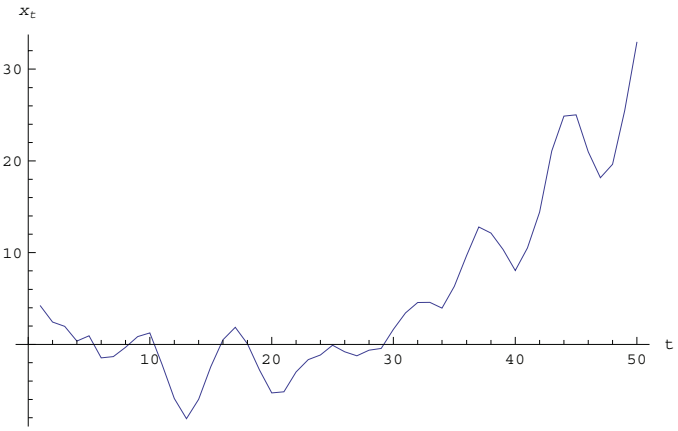
This reads in the data.

```
In[29]:= data = ReadList[ToFileName[{"TimeSeries", "Data"}, "file3.dat"], Number];
```

Here is the time plot of `data`.

```
In[30]:= plot1 = ListLinePlot[data, AxesLabel -> {"t", "x_t"}]
```

Out[30]=



It is clear from the time plot that the series is not stationary since the graph shows a tendency to increase with t, signaling a possible nonconstant mean $\mu(t)$. A nonconstant mean $\mu(t)$ is often referred to as a *trend*. A trend of a time series can be thought of as an underlying deterministic component that changes relatively slowly with time; the time series itself can be viewed as fluctuations superimposed on top of this deterministic trend. Estimat·

ing the trend by removing the fluctuations is often called *smoothing*, whereas eliminating the trend in order to obtain a series with a constant mean is often called *detrending*.

There are various ways of estimating a trend. For example, we can assume that the trend can be approximated by a smooth function and use a simple curve-fitting method. The *Mathematica* function `Fit` can be used to fit a linear combination of functions to the data using linear least squares method. In this example, we fit a quadratic function of t (*i.e.*, a linear combination of $\{1, t, t^2\}$) to the above data. Fitting a polynomial in t to time series data using the least squares method is often referred to as *polynomial regression*.

This fits the data to a quadratic polynomial.

```
In[31]:= trend = Fit[data, {1, t, t^2}, t]

Out[31]= 4.458 - 0.931377 t + 0.0284097 t^2
```

This is the estimate of the trend. We can remove the trend and get the "detrended" data or residuals by subtracting the trend from the data.

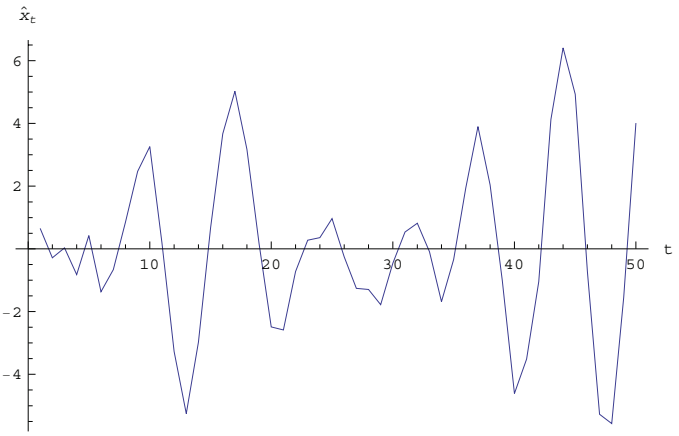
The trend is removed.

```
In[32]:= data = Table[trend, {t, 1, Length[data]}];
```

Here is the time plot of the "detrended" data. Compared to the previous plot no apparent systematic trend remains.

```
In[33]:= ListLinePlot[%, AxesLabel -> {"t", "x̂t"}]
```

Out[33]=



This series now appears suitable for modeling as a stationary process.

In using the curve fitting method for estimating the trend described above we implicitly assumed that a global function fits the entire data. This is not always desirable and often methods of modeling the trend locally are preferable. For example, a simple moving average can be used to estimate the trend. Let y_t be the smoothed value (*i.e.*, estimated trend) at t. A *simple moving average* of order r is defined by

$$y_t = \sum_{i=0}^{r-1} x_{t-i}/r \text{ for } t = r, r+1, \ldots n .$$

Thus each new data point is the average of r previous data points. Short time fluctuations are averaged out leaving smoothed data, which we use as an estimate of the trend. The function

```
MovingAverage[data, r]
```

performs a simple moving average of order r on data and gives the smoothed data $\{y_t\}$.

Instead of using identical weights in the above sum, we can generalize the simple moving average by allowing arbitrary weights $\{c_0, c_1, \ldots, c_r\}$. This leads to a *weighted moving average* or *linear filter*. (An analysis of linear filters in the frequency domain will be found in Section 1.8.2.) Application of a linear filter of weights $\{c_0, c_1, \ldots, c_r\}$ to $\{x_t\}$ produces the filtered series $\{y_t\}$ given by

$$y_t = \sum_{i=0}^{r} c_i \, x_{t-i} \text{ for } t = r+1, r+2, \ldots, n. \tag{4.1}$$

This is sometimes referred to as a one-sided or causal filter and $y_t = \sum_{i=-r}^{r} c_i \, x_{t-i}$ as a two-sided filter. The filter defined above is no less general than a two-sided one since a simple "time shifting" converts one kind of filter to the other.

The function

```
MovingAverage[data, {c0, c1, ... , cr}]
```

applies a moving average with weights $\{c_0, c_1, \ldots, c_r\}$ to *data*. The weights are rescaled to sum to 1. For a more detailed discussion of moving averages see Kendall and Ord (1990), pp. 28–34.

Another kind of moving average is the so-called *exponentially weighted moving average* or *exponential smoothing*. In exponential smoothing, the weights decrease exponentially and the smoothed series is given by

$$y_t = y_{t-1} + a\,(x_t - y_{t-1}), \text{ for } t = 1, 2, \ldots, n;$$

where a is called the *smoothing constant* and is usually restricted to values between 0 and 1. An initial value of y_0 is required to start the updating. The function

```
ExponentialMovingAverage[data, a]
```

performs exponential smoothing on data with smoothing constant a. The initial value is taken to be the first element in *data*. A different starting value y_0 can be used by prepending y_0 to *data*.

Example 4.9 Plot the time series generated from a random walk model.

An ARIMA(0, 1, 0) model is also called a *random walk model*. Here is the plot of 100 data points generated from a random walk model.

The random number generator is seeded first.

```
In[34]:= SeedRandom[3 845 717];
```

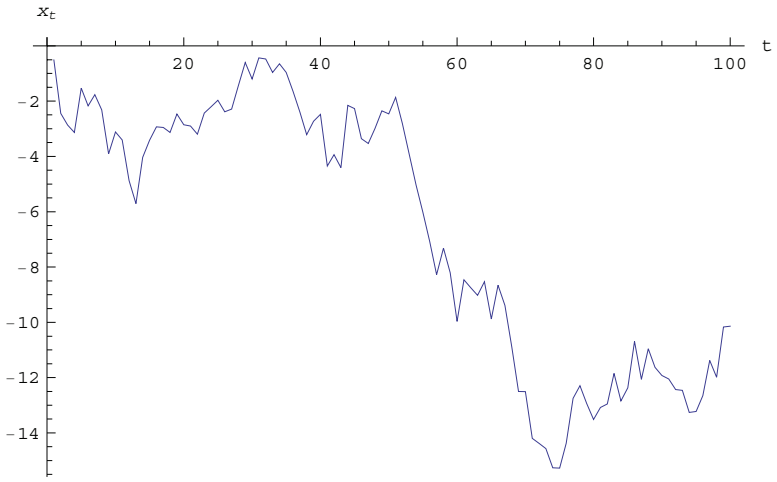
We generate the time series from the random walk model.

```
In[35]:= data = TimeSeries[ARIMAModel[1, {0}, {0}, 1], 100];
```

This is the time plot of the series.

```
In[36]:= ListLinePlot[data, AxesLabel -> {"t", "x_t"}]
```

Out[36]=



Again the graph shows that the data display a tendency to increase with t. We might infer that the process has a nonconstant mean $\mu(t)$. However, we know that the ARIMA model defined in Section 1.3 has mean zero, so this appearance of a nonstationary mean is the result of a single finite realization of a zero-mean nonstationary process. This apparent trend is often termed a *stochastic trend* in contradistinction to the deterministic trend we introduced earlier.

Differencing is an effective way of eliminating this type of trend and rendering the series stationary. Recall that the once-differenced data $\{y_t\}$ is given by $y_t = x_t - x_{t-1} = (1 - B)\, x_t$, and twice-differenced data by $(1 - B)\, y_t = x_t - 2\, x_{t-1} + x_{t-2} = (1 - B)^2\, x_t$. So the ARIMA(0, 1, 0) series after differencing is a stationary series $(1 - B)\, x_t = y_t = z_t$. Similarly, an ARIMA($p$, d, q) series can be transformed into a stationary series by differencing the data d times. To difference the data d times and get $(1 - B)^d\, x_t$ we can use

`ListDifference[`*data*`, `*d*`]`.

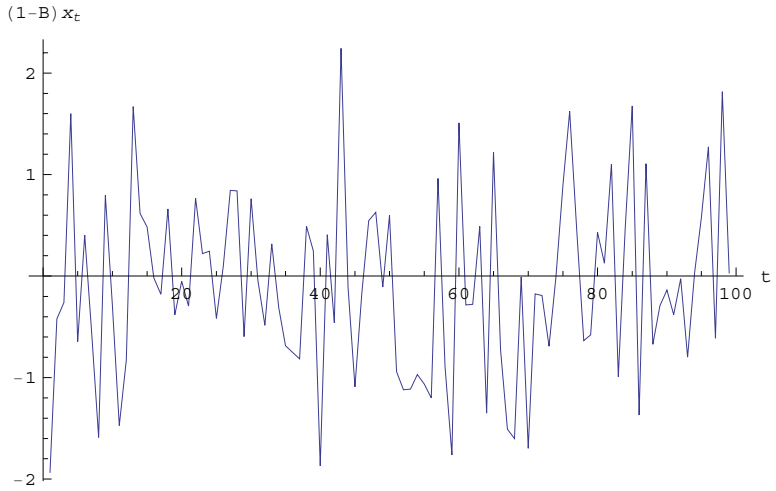
Notice that differencing is really a particular kind of linear filtering and a polynomial trend of degree k can be eliminated by differencing the data $k + 1$ times.

Now we difference the above random walk data and plot the differenced series.

The differenced data plotted here resembles a stationary series in contrast to the previous plot of the original data.

```
In[37]:= ListLinePlot[ListDifference[data, 1], AxesLabel -> {"t", "(1-B)x_t"}]
```

Out[37]=



So far we have dealt with eliminating a nonconstant mean present in the time series. Nonstationarity can also be due to nonconstant variance. For example, in the plot of the airline data (Example 1.2) we see clearly an increase in the variance as t increases. Here we re-display its time plot for examination.

Example 4.10 Transform the airline data into a stationary series.

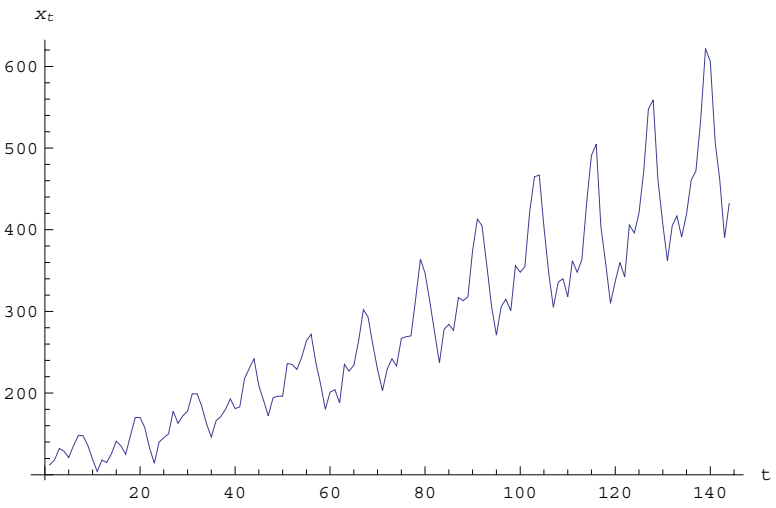
The contents of `airline.dat` are read and put into `aldata`.

```
In[38]:= aldata = ReadList[ToFileName[{"TimeSeries", "Data"}, "airline.dat"], Number];
```

The airline series is plotted here.

```
In[39]:= ListLinePlot[aldata, AxesLabel -> {"t", "x_t"}]
```

Out[39]=

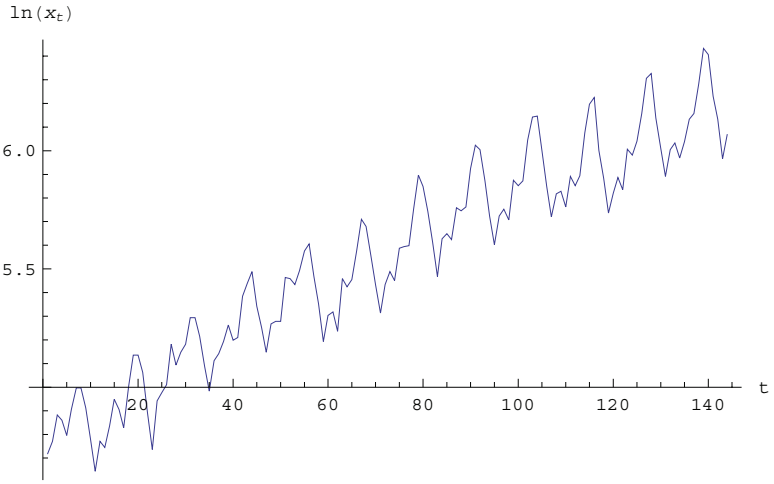


We see that the variance, riding on the trend, is also changing with time. We need to transform the series into a constant variance before modeling it further. To stabilize the variance, a nonlinear transformation such as a logarithmic or square-root transformation is often performed. In this example, we try a natural logarithmic transformation, $y_t = \ln x_t$.

This is the time plot of the airline data after the logarithmic transformation. Note that `Log[aldata]` gives the logarithm of each of the entries of `aldata`.

```
In[40]:= ListLinePlot[Log[aldata], AxesLabel -> {"t", "ln(x_t)"}]
```

Out[40]=



The variance now is stabilized. However, the trend is still present and there is the obvious seasonality of period 12. The method of differencing can also be used to eliminate the seasonal effects. We define the seasonal difference with period s as $x_t - x_{t-s} = (1 - B^s)\, x_t$.

We can difference the data d times with period 1 and D times with the seasonal period s and obtain $(1 - B)^d\, (1 - B^s)^D\, x_t$ using

`ListDifference[`*data*`, {`*d*`, `*D*`}, `*s*`]`.

Here is the airline data after the logarithmic transformation and seasonal differencing with period 12.

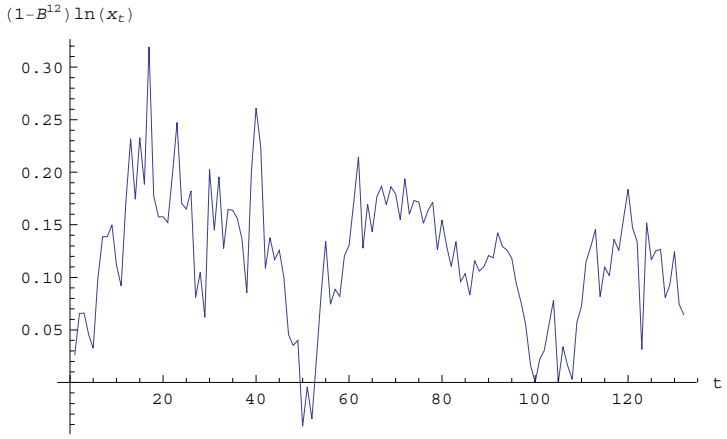
The transformed data is further differenced.

```
In[41]:= data = ListDifference[Log[aldata], {0, 1}, 12];
```

This is the plot of the differenced data.

```
In[42]:= ListLinePlot[data, AxesLabel -> {"t", "(1-B^12)ln(x_t)"}]
```

Out[42]=

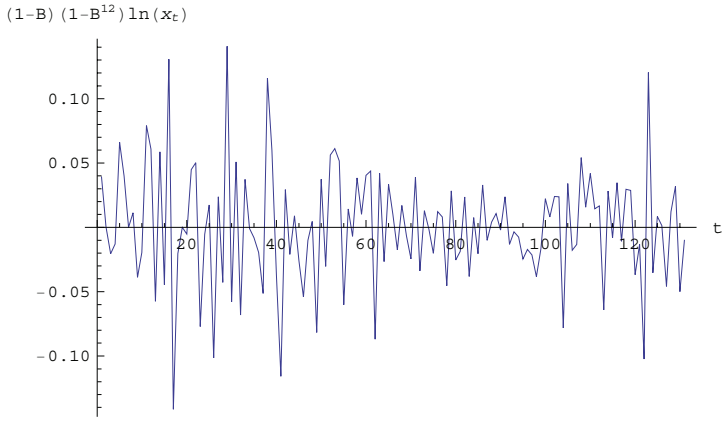


We see that the periodic behavior or seasonal effect has been eliminated. The series after removal of the seasonal component is often referred to as *seasonally adjusted* series or "deseasonalized" series. A further difference gives the following plot.

The transformed data appear stationary.

```
In[43]:= ListLinePlot[ListDifference[data, 1], AxesLabel -> {"t", "(1-B)(1-B^12)ln(x_t)"}]
```

Out[43]=



This series $(1 - B)\left(1 - B^{12}\right)\ln(x_t)$ can be fitted to a stationary model.

In fact, the logarithmic transformation is a special case of a class of transformations called the *Box-Cox transformation*. If we denote the transformed series by $\{y_t\}$ and let λ be a real constant, the Box-Cox transformation is defined by

$$y_t = \left(x_t^{\lambda} - 1\right)/\lambda \text{ for } \lambda \neq 0$$

and

$$y_t = \ln x_t \text{ for } \lambda = 0.$$

Different values of λ yield different transformations. It is trivial to implement this transformation on *Mathematica*. If `data` contains the time series data to be transformed then

$$\left(\texttt{data}^{\lambda} - 1\right) / \lambda$$

or

```
Log[data]
```

gives the transformed series.

Sometimes the nonstationarity can come from models not included in the ARIMA or SARIMA models. For example, the AR polynomial might have unit roots that are not 1. Since the AR coefficients are real, the complex roots on the unit circle will appear in complex conjugate pairs resulting in factors such as $\left(1 - e^{i\alpha} B\right)\left(1 - e^{-i\alpha} B\right) = \left(1 - 2\cos\alpha\, B + B^2\right)$ in our AR polynomial. We can use `ListCorrelate[{1, -2 Cos[`α`], 1}, data]` to remove this kind of nonstationarity.

After the data have been rendered stationary, we are ready to fit an appropriate model to the data. This is the subject of the next two sections.

1.5 Estimation of Correlation Function and Model Identification

As stated in the beginning, given a set of time series data we would like to determine the underlying mechanism that generated the series. In other words, our goal is to identify a model that can "explain" the observed properties of the series. If we assume that after appropriate transformations the series is governed by an ARMA type of model, model identification amounts to selecting the orders of an ARMA model.

In general, selecting a model (model identification), estimating the parameters of the selected model (parameter estimation), and checking the validity of the estimated model (diagnostic checking) are closely related and interdependent steps in modeling a time series. For example, some order selection criteria use the estimated noise variance obtained in the step of parameter estimation, and to estimate model parameters we must first know the model. Other parameter estimation methods combine the order selection and parameter estimation. Often we may need to first choose a preliminary model, and then estimate the parameters and do some diagnostic checks to see if the selected model is in fact appropriate. If not, the model has to be modified and the whole procedure repeated. We may need to iterate a few times to obtain a satisfactory model. None of the criteria and procedures are guaranteed to lead to the "correct" model for finite data sets. Experience and judgment form necessary ingredients in the recipe for time series modeling.

In this section we concentrate on model identification. Since the correlation function is the most telling property of a time series, we first look at how to estimate it and then use the estimated correlation function to deduce the possible models for the series. Other order selection methods will also be introduced. Parameter estimation and diagnostic checking are discussed in Section 1.6.

1.5.1 Estimation of Covariance and Correlation Functions

We are given n observations $\{x_1, x_2, \ldots, x_n\}$ of a process $\{X_t\}$ and we would like to estimate the mean and the covariance function of the process from the given data. Since, as we have mentioned before, the assumption of stationarity is crucial in any kind of statistical inference from a single realization of a process, we will assume in this section that the data have been rendered stationary using the transformations discussed in Section 1.4.

We use the *sample mean* $\overline{X}$ as the estimator of the mean of a stationary time series $\{X_t\}$,

$$\overline{X} = (X_1 + X_2 + \ldots + X_n)/n \tag{5.1}$$

and the *sample covariance function* $\hat{\gamma}$ as the estimator of the covariance function of $\{X_t\}$,

$$\hat{\gamma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}\left(X_{t+k} - \overline{X}\right)\left(X_t - \overline{X}\right). \tag{5.2}$$

That is, the expectation in (2.8) is replaced by an average over the series at different times. It should be borne in mind that $\overline{X}$ and $\hat{\gamma}$ defined in (5.1) and (5.2) are random variables, and for a particular realization of the time series $\{x_1, x_2, \ldots, x_n\}$ they give a particular estimate of the mean and the covariance function. Note that in the definition of the sample covariance, (5.2), the denominator is n although there can be fewer than n terms in the sum. There are other definitions of the sample covariance function that are slightly different from (5.2). For example, one definition uses $n - k$ rather than n in the denominator. For the advantages of using (5.2), see the discussions in Kendall and Ord (1990), Sections 6.2 and 6.3 and in Brockwell and Davis (1987), p. 213.

The *sample correlation function* $\hat{\rho}$ is defined to be the normalized sample covariance function,

$$\hat{\rho}(k) = \hat{\gamma}(k) \big/ \hat{\gamma}(0). \tag{5.3}$$

To calculate the sample mean from the given data we can use the function

`Mean[`*data*`]`

and to calculate the sample covariances and sample correlations up to lag k we can use the functions

`CovarianceFunction[`*data*`, k]` and `CorrelationFunction[`*data*`, k]`.

Note that these are the same functions we used to calculate theoretical covariance and correlation functions from a given model. The difference is in the first argument of these functions. To get the sample covariances or correlations from the given data, the first argument of these functions is the data instead of a model object.

In principle, we can calculate the covariance or correlation up to the maximum lag $n - 1$ where n is the length of the data. However, we should not expect $\hat{\gamma}(k)$ to be very reliable for k comparable to n since in this case there are too few terms contributing to the average in (5.2). However, if you want to calculate the correlation function up to the maximum lag often, you can define a function with the default lag value set to $n - 1$ as follows.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

The lag argument is omitted from the function `mycorrelation` and it is assumed to be $n - 1$.

```
In[2]:= mycorrelation[data1_, data2___] := CorrelationFunction[data1, data2, Length[data1] - 1]
```

Example 5.1 Calculate the sample mean and sample correlation function from the data of length 500 generated from the AR(2) process $X_t = 0.9\, X_{t-1} - 0.8\, X_{t-2} + Z_t$ (see Example 2.6).

We first generate the series according to the AR(2) model `ARModel[{0.9, -0.8}, 1]`.

The random number generator is seeded first.

```
In[3]:= SeedRandom[48376];
```

This generates the time series of length 500 according to the given AR(2) model.

```
In[4]:= data = TimeSeries[ARModel[{0.9, -0.8}, 1], 500];
```

Here is the sample mean of the series.

```
In[5]:= Mean[data]

Out[5]= -0.0042064
```

As we would have expected, the sample mean is close to the true mean 0. Next we calculate the sample correlation $\hat{\rho}(k)$ up to lag $k = 25$ and plot it against k. The plot of $\hat{\rho}(k)$ versus the lag k is often referred to as the *correlogram*.

This calculates the sample correlation function of the series up to lag 25.

```
In[6]:= corr = CorrelationFunction[data, 25];
```

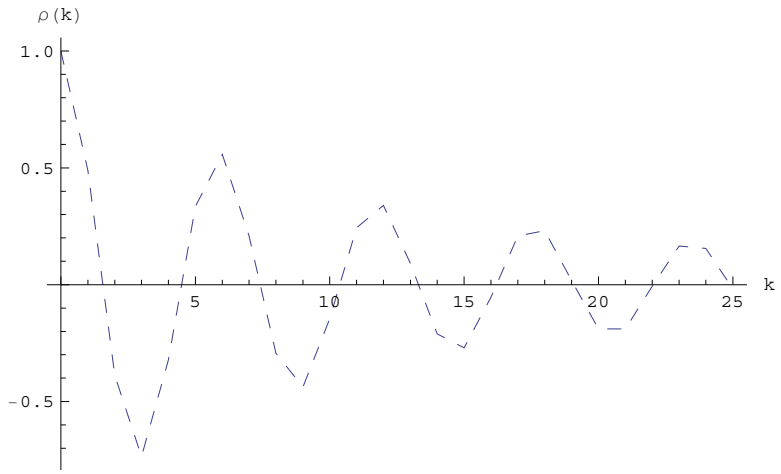
To plot the correlation function, we redefine the function `plotcorr` here.

```
In[7]:= plotcorr[corr_, opts___] :=
         ListPlot[corr, DataRange -> {0, Length[corr] - 1}, PlotRange -> All, opts]
```

Here is the plot of the sample correlation function. We call this plot `g2` for future re-display.

```
In[8]:= g2 = plotcorr[corr, Joined -> True,
           AxesLabel -> {"k", "ρ(k)"}, PlotStyle -> Dashing[{0.02}]]
```

Out[8]=



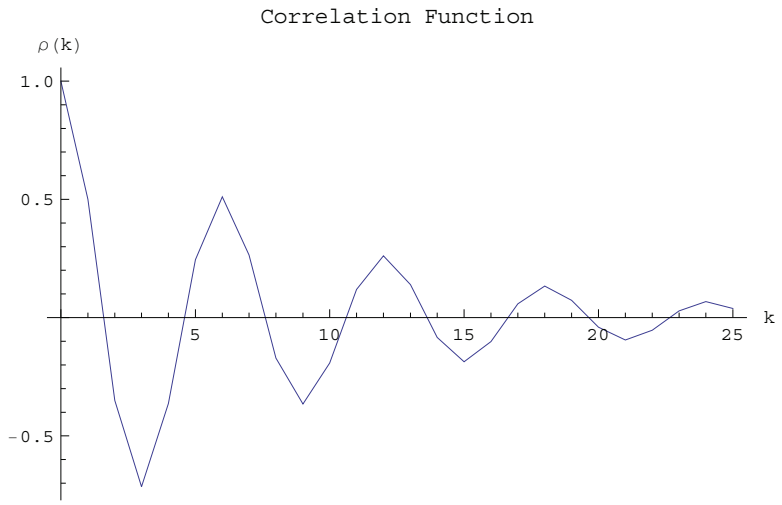
A dashed line is rendered due to the specification of the option `PlotStyle`. The number 0.02 inside `Dashing` specifies the length of the line segments measured as a fraction of the width of the plot.

The theoretical correlation function calculated from the same model was displayed in Example 2.6. Here we display it again for comparison.

This plots the theoretical correlation function of the AR(2) process.

```
In[9]:= g1 = plotcorr[CorrelationFunction[ARModel[{0.9, -0.8}, 1], 25],
          AxesLabel -> {"k", "ρ(k)"}, Joined -> True, PlotLabel -> "Correlation Function"]
```

Out[9]=

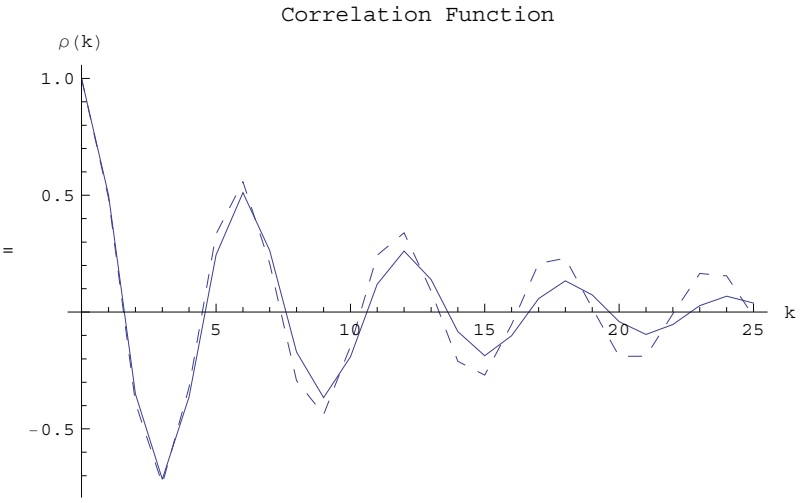


We can see how well the sample correlation function of the AR(2) process actually approximates the true correlation function by juxtaposing the plots of both using the command Show.

The theoretical correlation function (solid line) and the sample correlation function (broken line) are displayed together here using Show.

```
In[10]:= Show[g1, g2]
```

Out[10]=



We see that the sample correlation $\hat{\rho}(k)$ provides a reasonable approximation to the true correlation function $\rho(k)$. Intuitively we also expect, by an application of the central limit theorem, that the larger the n, the better $\hat{\rho}$ approximates ρ. This is indeed the case as we shall see in the next section.

1.5.2 The Asymptotic Distribution of the Sample Correlation Function

Let $\{X_t\}$ be a stationary process with the correlation function ρ. Let $\rho(h)' = (\rho(1), \rho(2), \ldots, \rho(h))$ and $\hat{\rho}(h)' = \left(\hat{\rho}(1), \hat{\rho}(2), \ldots, \hat{\rho}(h)\right)$. It can be shown (see, for example, Brockwell and Davis (1987), p. 214) that under certain general conditions $\hat{\rho}(h)$ has the asymptotic joint normal distribution with mean $\rho(h)$ and variance C/n as $n \to \infty$. The (i, j) element of the matrix C, c_{ij}, is given by

$$c_{ij} = \sum_{k=-\infty}^{\infty} \Big[\rho(k+i)\,\rho(k+j) + \rho(k-i)\,\rho(k+j) + 2\,\rho(i)\,\rho(j)\,\rho^2(k) - 2\,\rho(i)\,\rho(k)\,\rho(k+j) - 2\,\rho(j)\,\rho(k)\,\rho(k+i)\Big]. \tag{5.4}$$

This formula was first derived by Bartlett in 1946 and is called Bartlett's formula. Any stationary ARMA model with $\{Z_t\}$ distributed identically and independently with zero mean and finite variance satisfies the conditions of Bartlett's formula.

Hence for large n, the sample correlation at lag i, $\hat{\rho}(i)$, is approximately normally distributed with mean $\rho(i)$ and variance c_{ii}/n,

$$Var\left(\hat{\rho}(i)\right) = c_{ii}/n = \sum_{k=-\infty}^{\infty} \left[\rho^2(k+i) + \rho(k-i)\,\rho(k+i) + 2\,\rho^2(i)\,\rho^2(k) - 4\,\rho(i)\,\rho(k)\,\rho(k+i)\right]/n. \tag{5.5}$$

Bartlett's formula, (5.4) or (5.5), is extremely useful since it gives us a handle on deciding whether a small value in the sample correlation function is in fact significantly different from zero or is just a result of fluctuations due to the smallness of n. Next we give two examples where Bartlett's formula is used to determine if the sample correlation is zero.

Example 5.2 For a sequence of identically and independently distributed white noise $\{Z_t\}$, $\rho(0) = 1$ and $\rho(i) = 0$ for $i \neq 0$. The above formula, (5.5), reduces to (for $i \neq 0$)

$$Var\left(\hat{\rho}(i)\right) = 1/n.$$

That is, for large n, $\hat{\rho}(i)$ is normally distributed with mean zero and variance $1/n$ for $i = 1, 2, \ldots, h$. This implies that 95 percent of the time the plot of $\hat{\rho}$ should fall within the bounds $\pm 1.96/\sqrt{n}$. In practice, 2 rather than 1.96 is often used in calculating the bounds.

Here we generate a normally distributed random sequence of length 200 with mean 0 and variance 1.5. The sample correlation function is calculated up to 50.

The random number generator is seeded first.

```
In[11]:= SeedRandom[586 351]
```

This generates a random sequence of length 200 with distribution $N(0, 1.5)$.

```
In[12]:= data = RandomSequence[0, 1.5, 200];
```

The sample correlation function up to lag 50 is generated.

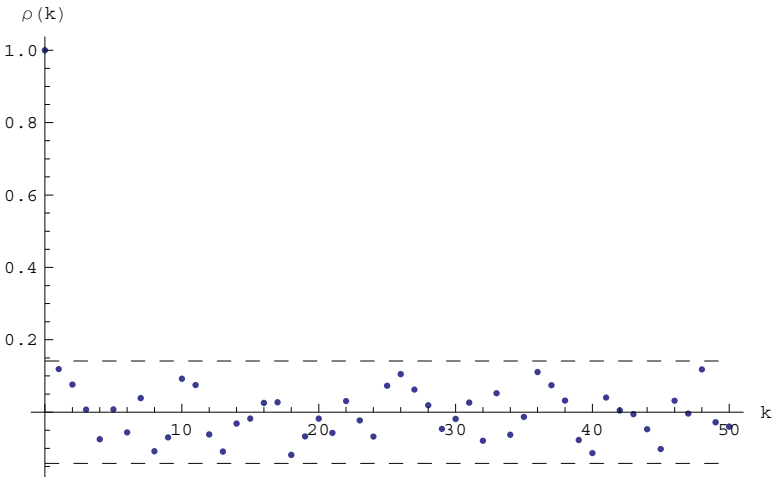
```
In[13]:= corr = CorrelationFunction[data, 50];
```

We can display this sample correlation function along with the bounds $\pm 2/\sqrt{200}$ using `Show`.

The sample correlation function and the bounds are displayed here using `Show`. The function `Plot` is used to plot the two constant functions that form the bounds.

```
In[14]:= Show[plotcorr[corr], Plot[{-2 / Sqrt[200.], 2 / Sqrt[200.]}, {x, 0, 50},
           PlotStyle -> {{Black, Dashing[{0.02}]}}], AxesLabel -> {"k", "ρ(k)"}]
```

Out[14]=



We see that $\hat{\rho}(k)$ falls within the bounds for all $k > 0$. We have no reason to reject the hypothesis that the set of data constitutes a realization of a white noise process.

You can also define your own function that plots the given correlation function and the bounds. For example, you can define the following.

This defines the function that plots the correlation function and the bounds.

```
In[15]:= myplotcorr1[corr_, bound_, opts___] :=
          Show[plotcorr[corr, Joined -> True], Plot[{-bound, bound},
            {x, 0, Length[corr] - 1}, PlotStyle -> {{Black, Dashing[{0.02}]}}], opts]
```

You can also define a function that does the same plot as `myplotcorr1` but uses the data rather than the correlation function as an argument. Note that now the bound is fixed to be $2/\sqrt{n}$ and the correlation is plotted to the maximum lag.

```
In[16]:= myplotcorr2[data_, opts___] := myplotcorr1[
           CorrelationFunction[data, Length[data] - 1], 2 / Sqrt[Length[data]], opts]
```

Example 5.3 For an MA(q) process, $\rho(k) = 0$ for $k > q$. From Bartlett's formula (5.5), it is easy to see that for $i > q$ only the first term in the sum survives. Therefore, for $i > q$ we have

$$Var\left(\hat{\rho}(i)\right) = \left(1 + 2\,\rho^2(1) + 2\,\rho^2(2) + \ldots + 2\,\rho^2(q)\right)\Big/ n. \tag{5.6}$$

If the data of length n (n large) are truly a realization of an MA(q) process, we expect the sample correlation function $\hat{\rho}(i)$ for $i > q$ to fall within the bounds given by $\pm 2\sqrt{Var\left(\hat{\rho}(i)\right)}$ about 95 percent of the time. In practice, the true correlation function ρ is unknown and (5.6) is used with the sample correlation function $\hat{\rho}$ in place of ρ.

Here we are given a set of stationary, zero-mean data of length 200 that is generated from an MA(2) process $X_t = Z_t - 0.4\,Z_{t-1} + 1.1\,Z_{t-2}$. We would like to determine the process that generated the data.

This seeds the random number generator.

```
In[17]:= SeedRandom[8174];
```

This generates a time series of length 200 from the given MA(2) model.

```
In[18]:= data = TimeSeries[MAModel[{-0.4, 1.1}, 1], 200];
```

We first calculate the sample correlation function and plot it along with the bounds for white noise.

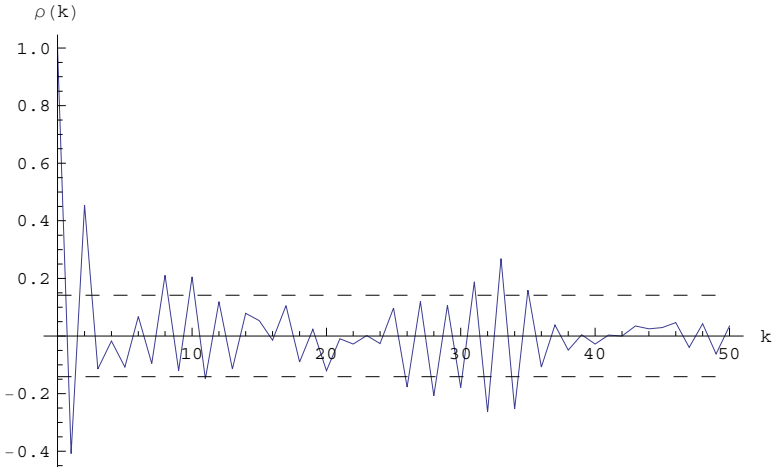
`corr` contains the sample correlation function of the series up to lag 50.

```
In[19]:= corr = CorrelationFunction[data, 50];
```

The sample correlation function and the bounds for a white noise process are displayed together using `myplotcorr1`.

```
In[20]:= myplotcorr1[corr, 2 / Sqrt[200], AxesLabel -> {"k", "ρ(k)"}]
```

Out[20]=



Since the sample correlation function at lags 1 and 2, $\hat{\rho}(1)$ and $\hat{\rho}(2)$, are well beyond the bound, we conclude that they differ significantly from zero and the data are not likely to be random noise. Since the correlations beyond lag 2 are all rather small we may suspect that the data can be modeled by an MA(2) process. The variance of $\hat{\rho}(k)$ for $k > 2$ can be calculated using (5.6), with the sample correlation function replacing the true correlation function, that is, we calculate $\left(1 + 2\,\hat{\rho}^2(1) + 2\,\hat{\rho}^2(2)\right)\Big/ n$.

We first get the sample correlation up to $k = 2$. This is done by extracting the first three elements of `corr` using `Take`.

This extracts the first three elements of `corr`.

```
In[21]:= Take[corr, 3]

Out[21]= {1., -0.408219, 0.454923}
```

We square the list (each individual element will be squared) and then use `Apply` to add all the elements together.

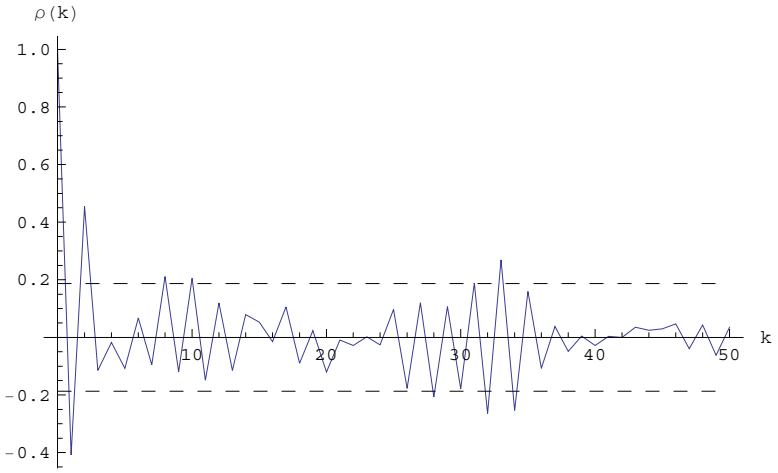
```
In[22]:= (2 Total[%^2] - 1) / 200

Out[22]= 0.00873597
```

We have subtracted 1 to get rid of overcounting $\rho(0)$ (= 1). Now we can display the sample correlation function again with the bounds we just calculated.

The sample correlation function is displayed along with the newly calculated bounds.

```
In[23]:= myplotcorr1[corr, 2 Sqrt[%], AxesLabel -> {"k", "ρ(k)"}]
```

Out[23]=



This affirms the reasonableness of our conclusion that the data are from an MA(2) process.

1.5.3 The Sample Partial Correlation Function

We use the sample partial correlation function $\hat{\phi}_{k,k}$ to estimate the partial correlation function. It is obtained by replacing the true covariance function $\gamma(k)$ used in the Levinson-Durbin algorithm with the sample covariance function $\hat{\gamma}(k)$. To get the sample partial correlation function from given data up to lag h we can use the function

`PartialCorrelationFunction[`*data*`, `*h*`]`.

Again the function name is the same as that used to calculate the partial correlation function of models.

For an AR(p) process, we know from Section 1.2.4 that $\phi_{k,k} = 0$ for $k > p$. The sample partial correlation $\hat{\phi}_{k,k}$ of an AR(p) process for large n and $k > p$ has an asymptotic normal distribution with mean zero and variance given by

$$Var\left(\phi_{k,k}\right) = 1/n. \tag{5.7}$$

We can use $\pm 2/\sqrt{n}$ as a guide to decide if the sample partial correlation can be considered zero.

In the following example, we are given a set of stationary, zero-mean data of length 200 generated from the AR(3) model $X_t = 1.5\,X_{t-1} - 1.0\,X_{t-2} + 0.4\,X_{t-3} + Z_t$.

The random number generator is seeded first.

```
In[24]:= SeedRandom[38 471]
```

This generates a time series of length 200 from the AR(3) model.

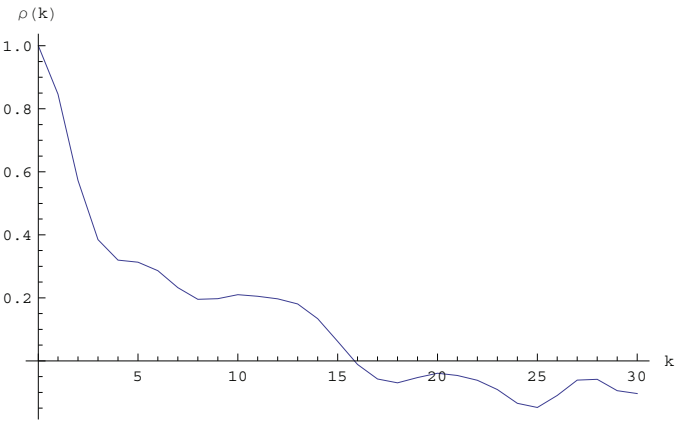
```
In[25]:= data = TimeSeries[ARModel[{1.5, -1.0, 0.4}, 1], 200];
```

Example 5.4 Make a preliminary determination of what process has generated the given data by looking at the sample correlation and the sample partial correlation functions.

We first calculate and plot the sample correlation function of the given data.

```
In[26]:= plotcorr[CorrelationFunction[data, 30], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

Out[26]=



Since there is no sharp cutoff in the plot of the correlation function, it is unlikely that the data are from a pure MA process. Next we calculate and plot the sample partial correlation function. The plot of the sample partial correlation function $\hat{\phi}_{k,k}$ versus the lag k is sometimes referred to as the *partial correlogram*; we display the partial correlogram of `data` together with the bounds $\pm 2/\sqrt{n}$.

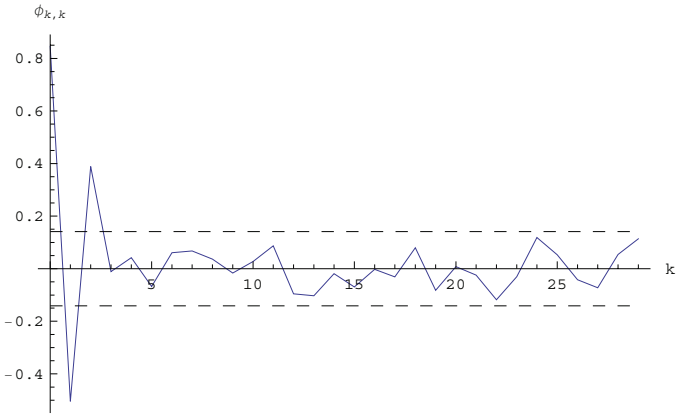
The sample partial correlation function up to lag 30 is calculated from the given series `data` and defined to be `pcf`.

```
In[27]:= pcf = PartialCorrelationFunction[data, 30];
```

The sample partial correlation function is displayed along with the bounds using the function `myplotcorr1` we defined in `In[16]`.

```
In[28]:= myplotcorr1[pcf, 2 / Sqrt[200], AxesLabel -> {"k", "ϕk,k"}]
```

Out[28]=



We see that the sample partial correlation differs from zero significantly only at the first 3 lags. This provides evidence to support the hypothesis that the data are from an AR(3) process.

1.5.4 Model Identification

There are various methods and criteria for selecting the orders of an ARMA model. We have seen in Sections 1.5.2 and 1.5.3 that the correlogram and partial correlogram can provide powerful tools to determine the order of a pure AR or MA process. So after the data have been transformed using the methods suggested in Section 1.4, the first thing to do is to look at the correlogram and partial correlogram. The failure of the sample correlation function to decay to zero sufficiently fast signals that further differencing may be necessary to make the data truly stationary.

Once the data are stationary, we can use our knowledge of theoretical correlation functions (see Section 1.2) of different models to deduce the possible ARMA orders for modeling our data. However, we must keep in mind that sample correlation functions do not always resemble the true correlation functions as well as in Example 5.1, especially in the small n case. Sample correlation at different lags can be highly correlated (see (5.4) for the expression of covariance of $\hat{\rho}(i)$ and $\hat{\rho}(j)$). Also it is easy to show that $\hat{\gamma}$ satisfies $\sum_{k=1}^{n-1} \hat{\gamma}(k) = -1/2$, even though the theoretical correlation function $\gamma(k)$ is strictly positive for all k. (That is, $\hat{\gamma}$ is biased.)

Another type of order selection method is based on the so-called information criteria. The idea is to balance the risks of underfitting (selecting orders smaller than the true orders) and overfitting (selecting orders larger than the true orders). The order is chosen by minimizing a penalty function. The two commonly used functions are

$$\ln \hat{\sigma}^2 + 2\,(p+q)/n \tag{5.8}$$

and

$$\ln \hat{\sigma}^2 + (p+q)\ln n/n. \tag{5.9}$$

Here $\hat{\sigma}^2$ is the estimated noise variance usually obtained from maximum likelihood estimations and n, as usual, is the length of the data. Akaike first suggested that the orders p and q be chosen such that they minimize the value of (5.8). This is called *Akaike's information criterion* (AIC). Similarly, using the minimum of (5.9) to select orders is called using *Bayesian information criterion* (BIC). (Other definitions of AIC use the logarithm of the likelihood function. See, for example, Brockwell and Davis (1987), p. 280.)

The first term in (5.8) and (5.9) can be interpreted as a penalty for underfitting, since the terms that are missing from the parameter fitting will be included in the noise and therefore give rise to an additional contribution to the noise variance. The second term in (5.8) and (5.9) is directly proportional to the number of ARMA parameters and it is clearly the penalty for overfitting. Its effect is to favor a parsimonious model. Choosing a model with the smallest possible number of parameters is an important principle in model selection, and this principle is referred to as the *principle of parsimony*. See Box and Jenkins (1970), Section 1.3.

To get the AIC or BIC value of an estimated model we can simply use

`AIC[`*model*`, `*n*`]` or `BIC[`*model*`, `*n*`]`.

Since the calculation of these values requires estimated noise variance, we will postpone demonstrating the use of these functions until Section 1.6.

A seasonal ARIMA$(p, 0, q)(P, 0, Q)_s$ model can, in principle, be treated as a special ARMA$(p + s\,P, q + s\,Q)$ model in model selection. However, since the number of independent parameters involved is often much smaller than that of a general ARMA$(p + s\,P, q + s\,Q)$ model, the principle of parsimony dictates that we should determine s, P, and Q as well as p and q. This usually presents additional difficulties in order selection. However, if the orders of the seasonal models are small, which often seems to be the case in practice, we can determine the seasonal period s by inspecting the correlogram using the theoretical properties of the correlation function we discussed in Section 1.3. The cutoff in the sample correlation or partial correlation function can suggest possible values of $q + s\,Q$ or $p + s\,P$. From this we can select the orders of regular and seasonal parts.

Example 5.5 A set of seasonal data of length 200 is given below. Determine the orders of the seasonal model that generated the data.

This seeds the random number generator.

```
In[29]:= SeedRandom[205967];
```

This generates a time series of length 200 from the given SARIMA$(1, 0, 0)(1, 0, 0)_4$ model.

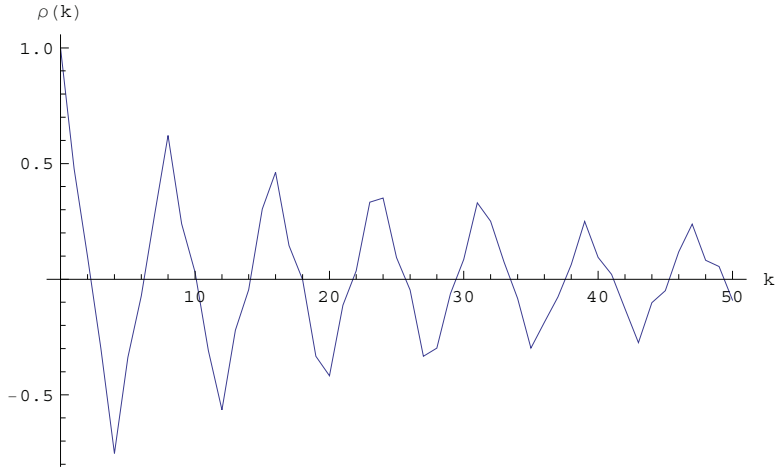
```
In[30]:= data = TimeSeries[SARIMAModel[{0, 0}, 4, {0.6}, {-0.85}, {}, {}, 1], 200];
```

First we look at the correlogram.

This gives the plot of the sample correlation function.

```
In[31]:= plotcorr[CorrelationFunction[data, 50], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

Out[31]=

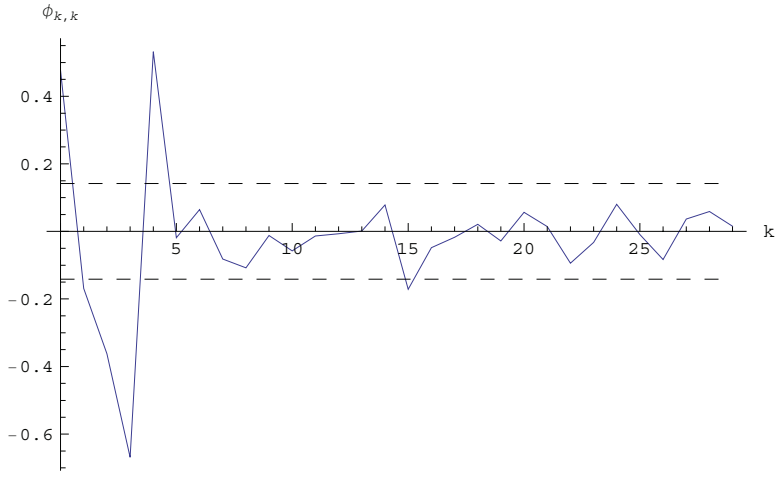


We see that the correlogram has extrema at lags that are multiples of 4. This suggests that $s = 4$. Since the correlogram has no sharp cutoff, the model is unlikely to be a pure MA model. Next we look at the sample partial correlation function. We plot it along with the bounds $\pm 2/\sqrt{200}$.

The sample partial correlation function is displayed along with the bounds $\pm 2/\sqrt{200}$.

```
In[32]:= myplotcorr1[PartialCorrelationFunction[data, 30],
          2 / Sqrt[200], AxesLabel -> {"k", "ϕk,k"}]
```

Out[32]=



The partial correlogram has a cutoff at lag $k = 5$. We conclude that the model is a pure AR model with $p + s\,P = 5$. Given that $s = 4$, we may deduce that $p = 1$ and $P = 1$. This is, of course, an extremely simple example. In practice, model identification can be much more complicated.

For a mixed model we can select the orders using the AIC and BIC defined in (5.8) and (5.9) with $p + q$ replaced by the number of seasonal model parameters $p + P + q + Q$.

1.5.5 Order Selection for Multivariate Series

For an m-variate series, X_t is a column vector of length m. The sample mean is again given by (5.1) and `Mean[data]` gives the estimate of the mean vector. The definition of sample covariance in the multivariate case is given by

$$\hat{\Gamma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}\left(X_{t+k} - \overline{X}\right)\left(X_t - \overline{X}\right)'$$

and

$$\hat{\Gamma}(-k) = \frac{1}{n}\sum_{t=k+1}^{n}\left(X_{t-k} - \overline{X}\right)\left(X_t - \overline{X}\right)'$$

for $0 \le k < n$. The sample correlation function $\hat{R}((k))_{ij} = \hat{\rho}_{ij}(k)$ is defined to be

$$\hat{\rho}_{ij}(k) = \hat{\gamma}_{ij}(k) \Big/ \left(\hat{\gamma}_{ii}(0)\,\hat{\gamma}_{jj}(0)\right)^{1/2},$$

where $\hat{\gamma}_{ij}(k) = \hat{\Gamma}((k))_{ij}$. Note that the above equation reduces to (5.3), the univariate sample correlation, when $i = j$.

In fact, the correlation function is the covariance function of the normalized or standardized series (*i.e.*, a series with zero mean and unit variance), which is obtained by subtracting from each component its mean and dividing by its standard deviation.

> **Example 5.6** Calculate the sample correlation matrices up to lag 3 from the given bivariate data.

The random number generator is seeded first.

```
In[33]:= SeedRandom[479 602]
```

This generates a bivariate time series of length 80 from the given MA(1) model.

```
In[34]:= data = TimeSeries[MAModel[{{{0.5, -0.1}, {0.2, -0.8}}}, {{1, 0}, {0, 1}}], 80];
```

The sample correlation function up to lag 3 is calculated and displayed in a table form.

```
In[35]:= TableForm[Transpose[{Table[rho[i], {i, 0, 3}], CorrelationFunction[data, 3]}]]
```

```
Out[35]//TableForm=
      rho[0]    1.        0.17511
                0.17511   1.
      rho[1]    0.413944  -0.0718686
                0.116806  -0.480424
      rho[2]    0.128807   -0.0246143
                0.0228069  0.0589108
      rho[3]    0.13624     0.125488
                -0.0273056  -0.206116
```

We can get the same result by calculating the sample covariance function Γ of the standardized data. The data can be standardized as follows.

The series generated from the given MA(1) model is standardized.

```
In[36]:= data1 = Transpose[(Transpose[data] - Mean[data]) / StandardDeviation[data]
             Sqrt[Length[data] / (Length[data] - 1)]];
```

Each component of the standardized series `data1` has mean zero.

```
In[37]:= Mean[data1]
```

$$\text{Out[37]= } \left\{-8.76035 \times 10^{-17},\ -3.39626 \times 10^{-18}\right\}$$

The covariance function of the standardized data up to lag 3 is calculated and tabulated.

```
In[38]:= TableForm[Transpose[{Table[gamma[i], {i, 0, 3}], CovarianceFunction[data1, 3]}]]
```

```
Out[38]//TableForm=
      gamma[0]  1.        0.17511
                0.17511   1.
      gamma[1]  0.413944  -0.0718686
                0.116806  -0.480424
      gamma[2]  0.128807   -0.0246143
                0.0228069  0.0589108
      gamma[3]  0.13624     0.125488
                -0.0273056  -0.206116
```

This result is identical to the previous one. The variances of the series 1 and 2 are given by $\Gamma((0))_{11}$ and $\Gamma((0))_{22}$ and they are both 1. Also notice that $\Gamma(k) \approx 0$ for $k > q = 1$.

The methods of order selection employed in the univariate case can also be used for a multivariate series. The fact that an AR(p) process has vanishing partial correlations at lags greater than p and an MA(q) process has vanishing correlations at lags greater than q helps determine the order of an AR or MA process, respectively. For an MA(q) process, the asymptotic variance of the sample correlation $\hat{\rho}_{ij}(k)$ ($k > q$) is given by

$$Var\left(\hat{\rho}_{ij}(k)\right) = \frac{1}{n}\left(1 + 2\sum_{l=1}^{q} \rho_{ii}(l)\,\rho_{jj}(l)\right)$$

for all i and j. If the cross-correlation between two series $\rho_{ij}(k)$ vanishes for all k (*i.e.*, the two series are uncorrelated) and one of the series is a white noise process, the above formula for the variance of $\hat{\rho}_{ij}(k)$ reduces to

$$Var\left(\hat{\rho}_{ij}(k)\right) = 1 \Big/ n.$$

This result can be used to test if two series are independent of each other.

The sample partial correlation for a multivariate case is a direct extension of that for the univariate case. It is calculated using the Levinson-Durbin algorithm. Again, `PartialCorrelationFunction[`*data*`,` *h*`]` gives the sample partial correlation function up to lag h.

The AIC and BIC of an m-variate process use the penalty functions

$$\ln|\hat{\Sigma}| + 2\,m^2(p+q)\Big/n$$

and

$$\ln|\hat{\Sigma}| + 2\,m^2(p+q)\ln n / n,$$

respectively. ($|\hat{\Sigma}|$ denotes the determinant of $\hat{\Sigma}$.) They are again calculated using the functions `AIC[`*model*`,` *n*`]` and `BIC[`*model*`,` *n*`]` and can be used to select the orders of an m-variate ARMA process.

1.6 Parameter Estimation and Diagnostic Checking

In this section different methods of estimating the ARMA parameters are presented: the Yule-Walker method, the Levinson-Durbin algorithm, Burg's algorithm, the innovations algorithm, the long AR method, the Hannan-Rissanen procedure, the maximum likelihood method, and the conditional maximum likelihood method. Functions calculating the logarithm of the likelihood function, the estimated information matrix, and the asymptotic covariance of the maximum likelihood estimators are also given. Residuals and the portmanteau statistic used in the diagnostic checking can also be obtained with functions defined in this section.

1.6.1 Parameter Estimation

We first introduce some commonly used methods of estimating the parameters of the ARMA types of models. Each method has its own advantages and limitations. Apart from the theoretical properties of the estimators (*e.g.*, consistency, efficiency, etc.), practical issues like the speed of computation and the size of the data must also be taken into account in choosing an appropriate method for a given problem. Often, we may want to use one method in conjunction with others to obtain the best result. These estimation methods, in general, require that the data be stationary and zero-mean. Failure to satisfy these requirements may result in nonsensical results or a breakdown of the numerical computation. In the following discussion we give brief descriptions of each estimation method in the time series package; for more details the reader is urged to consult a standard time series text.

Yule-Walker Method

The *Yule-Walker method* can be used to estimate the parameters of an AR(p) model for a given p. If we multiply each side of (2.7) by X_{t-k}, $k = 0, 1, \ldots, p$, and take expectations, we get a set of linear equations called the *Yule-Walker equations*:

$$\gamma(k) = \phi_1 \gamma(k-1) + \phi_2 \gamma(k-2) + \ldots + \phi_p \gamma(k-p) \tag{6.1}$$

for $k = 1, 2, \ldots, p$ and

$$\gamma(0) - \phi_1 \gamma(1) - \phi_2 \gamma(2) - \ldots - \phi_p \gamma(p) = \sigma^{2.} \tag{6.2}$$

The Yule-Walker equations can be solved for the covariance function γ given the AR parameters (in fact, the function `CovarianceFunction` for AR models is obtained by solving the Yule-Walker equations) or they can be solved for the AR coefficients $\{\phi_i\}$ and the noise variance σ^2 if the covariance function is known. In practice, the exact covariance function is unknown and a natural way of getting an estimate of the AR parameters is to use the sample covariance function $\hat{\gamma}$ to replace the corresponding theoretical covariance function γ in (6.1) and (6.2) and solve for $\{\phi_i\}$ and σ^2. The solution so obtained is called the Yule-Walker estimate of the AR parame-

ters. In general, the method of estimating parameters by equating sample moments to theoretical moments is referred to as the *method of moments*.

To estimate the parameters of an AR(p) model fitted to *data* using the Yule-Walker method we can use the function

YuleWalkerEstimate[*data*, *p*].

It gives the estimated model object

ARModel[{$\hat{\phi}_1$, $\hat{\phi}_2$, … , $\hat{\phi}_p$}, $\hat{\sigma}^2$].

Example 6.1 Fit an AR(3) model to the data generated from the model $X_t = 0.9\,X_{t-1} - 0.55\,X_{t-2} + 0.4\,X_{t-3} + Z_t$ using the Yule-Walker method.

We load the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

The random number generator is seeded first.

```
In[2]:= SeedRandom[304 918];
```

A time series of length 120 is generated from the specified AR(3) model.

```
In[3]:= data = TimeSeries[ARModel[{0.9, -0.55, 0.4}, 1], 120];
```

Assuming that the order has been correctly identified we can proceed to estimate the parameters using the Yule-Walker method.

This gives the estimated AR(3) model using the Yule-Walker method.

```
In[4]:= YuleWalkerEstimate[data, 3]

Out[4]= ARModel[{0.763343, -0.242644, 0.204865}, 0.829638]
```

Note that the three numbers in the list are the estimated values of ϕ_1, ϕ_2, and ϕ_3, and the number 0.858176 is the estimated noise variance.

Levinson-Durbin Algorithm

Levinson and Durbin derived an iterative way of solving the Yule-Walker equations. Instead of solving (6.1) and (6.2) directly, which involves inversion of a $p \times p$ matrix, the *Levinson-Durbin algorithm* fits AR models of successively increasing orders AR(1), AR(2), … , AR(k) to the data. Let $\hat{\phi}_{m,j}$ and $\hat{\sigma}_m^2$ be the j^{th} AR coefficient and the noise variance of the fitted AR(m) model, respectively; the recursion formulas of the Levinson-Durbin algorithm are (see Brockwell and Davis (1987), Section 8.2 and Morettin (1984))

$$\hat{\phi}_{m,m} = \left(\hat{\gamma}(m) - \sum_{j=1}^{m-1} \hat{\phi}_{m-1,j}\,\hat{\gamma}(m-j)\right) \Big/ \hat{\sigma}^2_{m-1},$$

$$\hat{\phi}_{m,k} = \hat{\phi}_{m-1,k} - \hat{\phi}_{m,m}\,\hat{\phi}_{m-1,m-k}$$

for $k = 1, 2, \ldots, m-1$, and

$$\hat{\sigma}^2_m = \hat{\sigma}^2_{m-1}\left(1 - \hat{\phi}^2_{m,m}\right)$$

with $\hat{\sigma}^2_0 = \hat{\gamma}(0)$.

The advantage of using the Levinson-Durbin algorithm over a direct solution of the Yule-Walker equations is that it also gives us the partial correlations $\{\hat{\phi}_{1,1}, \hat{\phi}_{2,2}, \ldots\}$. (In fact, `PartialCorrelationFunction` uses the Levinson-Durbin algorithm to compute the partial correlation function.) This allows us to select an appropriate AR model in the process of the parameter estimation. The function

`LevinsonDurbinEstimate[`*data*`, `*k*`]`

fits models AR(1), AR(2), … , AR(k) to *data* and gives a list of estimated model objects.

Example 6.2 Use the Levinson-Durbin algorithm to fit an AR model to the data used in Example 6.1.

This gives six estimated AR models of increasing order.

```
In[5]:= armodels = LevinsonDurbinEstimate[data, 6]

Out[5]= {ARModel[{0.683366}, 0.873062], ARModel[{0.744897, -0.0900402}, 0.865984],
         ARModel[{0.763343, -0.242644, 0.204865}, 0.829638],
         ARModel[{0.735732, -0.209941, 0.101983, 0.134778}, 0.814568],
         ARModel[{0.74695, -0.201451, 0.0845077, 0.196021, -0.0832399}, 0.808924],
         ARModel[{0.748482, -0.205057, 0.0829532, 0.199726, -0.0969802, 0.0183952}, 0.80865]}
```

We can decide on the appropriate order by looking at the last AR(i) coefficient $\phi_{i,i}$ for $i = 1, 2, \ldots, 6$. This is precisely the sample partial correlation function defined earlier. To see this we can explicitly extract the last number in the list of AR coefficients of each model.

The last AR coefficient of each model estimated above is extracted.

```
In[6]:= %[[All, 1, -1]]

Out[6]= {0.683366, -0.0900402, 0.204865, 0.134778, -0.0832399, 0.0183952}
```

Whenever we want to extract an element from the same position in each entry in a list, we can use the `All` part specification. Here we wish to extract the last AR coefficient. The AR coefficients are the first parts of each `ARModel` and the last coefficient is the -1 part of that list.

We get the same result using `PartialCorrelationFunction`.

```
In[7]:= PartialCorrelationFunction[data, 6]

Out[7]= {0.683366, -0.0900402, 0.204865, 0.134778, -0.0832399, 0.0183952}
```

We see that the partial correlations fall within the bound $2/\sqrt{120} = 0.18$ (see (5.7)) for $k > 3$. So we choose to model the data by an AR(3) process and the estimated model is the third entry in `armodels`.

This extracts the third entry in `armodels`.

```
In[8]:= armodels[[3]]

Out[8]= ARModel[{0.763343, -0.242644, 0.204865}, 0.829638]
```

Note that this is exactly the same as the result we got in Example 6.1 by using `YuleWalkerEstimate`.

We draw the attention of the reader to the behavior of the estimated noise variances. To see how they change as the order of the model is varied, we extract and plot them.

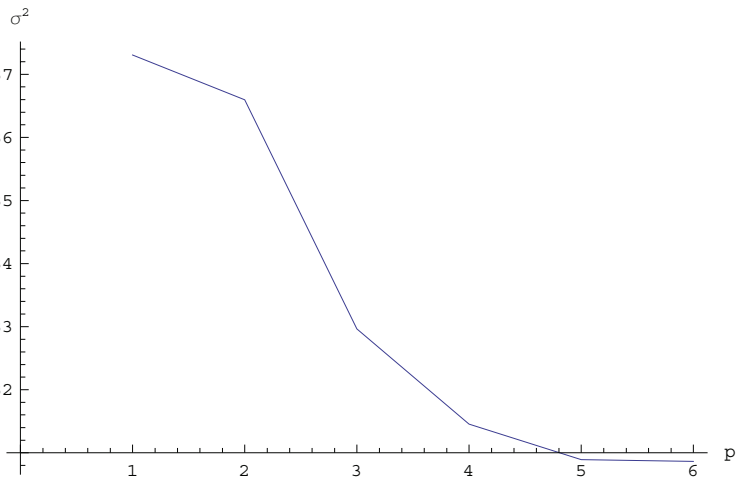
The noise variances are extracted from `armodels`.

```
In[9]:= armodels[[All, -1]]

Out[9]= {0.873062, 0.865984, 0.829638, 0.814568, 0.808924, 0.80865}
```

We plot the noise variance as a function of the AR model order p.

```
In[10]:= ListLinePlot[%, AxesLabel -> {"p", "σ²"}, PlotRange -> All]
```

Out[10]=



We see that initially the variance drops as we increase the order of AR model to be fitted, and after $p = 3$ the variance levels off. This gives an indication that the true order of the model is probably $p = 3$. This is because if the order of the fitted model is smaller than the true order, the noise variance will get an additional contribution from the terms that have been neglected and will be larger.

Burg's Algorithm

Burg's algorithm (also referred to as the *maximum entropy method*) for estimating the AR parameters is also iterative, but it works directly with the data rather than with the sample covariance function. For large samples, Burg's algorithm is asymptotically equivalent to the Yule-Walker estimates although the two differ on small sample properties. For a presentation of Burg's algorithm see Jones (1978). The function

```
BurgEstimate[data, k]
```

fits AR(1), AR(2), … , AR(*k*) models to *data* and gives a list of estimated model objects using Burg's algorithm.

Example 6.3 Use Burg's algorithm to fit an AR model to the data from the AR(3) model studied in Example 6.1.

This gives 5 estimated AR models using Burg's algorithm.

```
In[11]:= BurgEstimate[data, 5]

Out[11]= {ARModel[{0.68477}, 0.872083], ARModel[{0.751946, -0.0980994}, 0.863691],
           ARModel[{0.774545, -0.271324, 0.230368}, 0.817855],
           ARModel[{0.744897, -0.236405, 0.130687, 0.128697}, 0.804309],
           ARModel[{0.755675, -0.225461, 0.110888, 0.191081, -0.0837485}, 0.798668]}
```

This is the estimated AR(3) model. Compare it with the result of the Levinson-Durbin algorithm in Example 6.2.

```
In[12]:= %[[3]]

Out[12]= ARModel[{0.774545, -0.271324, 0.230368}, 0.817855]
```

Innovations Algorithm

For an MA model the method of moments involves solving a set of nonlinear equations and, therefore, it is not as convenient. Instead, other methods are used. The so-called *innovations algorithm* can be used to obtain a preliminary estimate of MA parameters.

Fitting MA(1), MA(2), … models to time series data using the innovations algorithm is analogous to fitting AR(1), AR(2), … models to data using the Levinson-Durbin algorithm. The iterative formulas and the details of the derivation can be found in Brockwell and Davis (1987), pp. 165–166 and Section 8.3.

The function

```
InnovationEstimate[data, k]
```

allows us to use the innovations algorithm to get an estimate of the MA parameters. It gives the estimated models MA(1), MA(2), … , MA(*k*).

Let $\hat{\theta}_{m,j}$ be the j^{th} MA coefficient of the estimated MA(m) model, then the asymptotic variance of $\hat{\theta}_{m,j}$ can be estimated to be

$$\hat{\sigma}_j^2 = Var\,(\hat{\theta}_{m,j}) = \left(1 + \sum_{k=1}^{j-1} \hat{\theta}_{m,k}^2\right) \Big/ n, \tag{6.3}$$

and the order q is estimated to be the lag beyond which $\hat{\theta}_{m,j}$ lies within $\pm 2\,\hat{\sigma}_j$ for $j > q$.

Once the order q is chosen, the estimated model parameters are given by $\hat{\theta}_{m,j}$ for $j = 1, 2, \ldots, q$ where m is sufficiently large that the values of $\hat{\theta}_{m,j}$ have stabilized with respect to m. In contrast, in the Levinson-Durbin algorithm, once the order of the AR process p is chosen, the estimated parameters for that p describe the estimated model. To illustrate how to get the estimate of an MA model, we generate a set of data from an MA(2) process and then try to fit an MA model to it.

This seeds the random number generator.

```
In[13]:= SeedRandom[394 587];
```

A time series of length 120 is generated from the given AR(2) model.

```
In[14]:= data = TimeSeries[MAModel[{-1.5, 0.7}, 1], 120];
```

Example 6.4 Fit an appropriate ARMA model to the given stationary time series `data`.

We first look at the correlogram.

This calculates the sample correlation function up to lag 25.

```
In[15]:= corr = CorrelationFunction[data, 25];
```

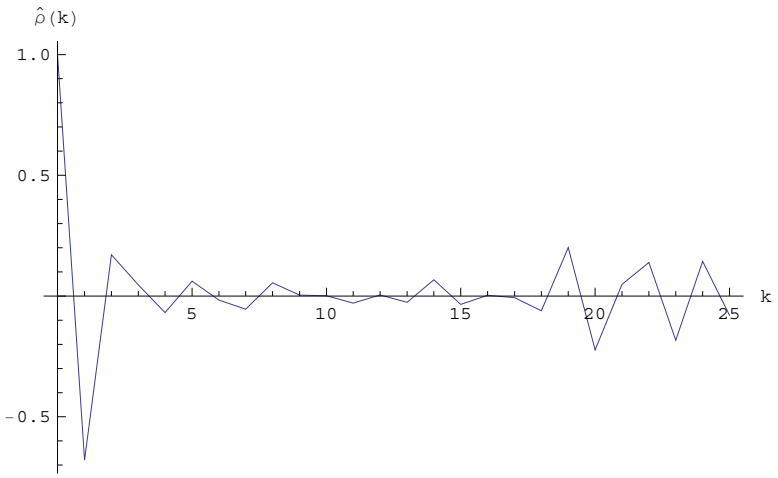
To plot the correlation function, we redefine the function `plotcorr` here.

```
In[16]:= plotcorr[corr_, opts___] :=
          ListPlot[corr, DataRange -> {0, Length[corr] - 1}, PlotRange -> All, opts]
```

The above sample correlation function `corr` is plotted using the function `plotcorr`.

```
In[17]:= plotcorr[corr, Joined -> True, AxesLabel -> {"k", "ρ̂(k)"}]
```

Out[17]=



We see that $|\hat{\rho}(k)|$ is small (< 0.2) except for $k \leq 2$. We can conclude that the data are from an MA model of relatively small order. Next we calculate the bound for sample correlation function assuming $q = 1$ (see (5.6))

The bound for the sample correlation function is calculated under the assumption that $q = 1$.

```
In[18]:= 2 Sqrt[(2 Total[Take[corr, 2]^2] - 1) / 120]
Out[18]= 0.253167
```

The value of $\hat{\rho}(2)$ is within the bound.

```
In[19]:= corr[[3]]
Out[19]= 0.170959
```

So MA(1) is a plausible model.

Now we use the innovations algorithm to determine the MA order and estimate the parameters.

This gives ten estimated MA models of increasing order using the innovations algorithm.

```
In[20]:= mamodels = InnovationEstimate[data, 10]

Out[20]= {MAModel[{-0.679264}, 1.74834], MAModel[{-1.04556, 0.170959}, 1.23994],
          MAModel[{-1.24477, 0.375973, 0.0464327}, 1.07071],
          MAModel[{-1.3602, 0.478456, 0.000269039, -0.068144}, 0.966188],
          MAModel[{-1.42063, 0.570985, 0.0215842, -0.0492007, 0.0613083}, 0.930046],
          MAModel[{-1.42687, 0.647133, 0.0369225, -0.0341669, 0.0927607, -0.0167054},
           0.929079], MAModel[{-1.42865, 0.645337, -0.00497664, -0.0831326, 0.0378468,
            -0.0997403, -0.0544924}, 0.926231], MAModel[{-1.43473, 0.644359, -0.0148594,
            -0.0982001, 0.0313664, -0.115448, -0.031991, 0.0548565}, 0.915075],
          MAModel[{-1.44321, 0.648848, -0.0134027, -0.0893809, 0.0510151,
            -0.098105, 0.0128691, 0.10665, 0.00380608}, 0.909607],
          MAModel[{-1.43844, 0.667823, -0.00788841, -0.0861545, 0.0709004,
            -0.070265, 0.0538966, 0.155801, 0.00864946, 0.00125505}, 0.906144]}
```

In order to compare the MA coefficients of the different models, let us extract and align them using the *Mathematica* function `Column`.

The MA coefficients are extracted and aligned using `Column`.

```
In[21]:= NumberForm[Column[mamodels[[All, 1]]], 2]

Out[21]//NumberForm=
      {-0.68}
      {-1., 0.17}
      {-1.2, 0.38, 0.046}
      {-1.4, 0.48, 0.00027, -0.068}
      {-1.4, 0.57, 0.022, -0.049, 0.061}
      {-1.4, 0.65, 0.037, -0.034, 0.093, -0.017}
      {-1.4, 0.65, -0.005, -0.083, 0.038, -0.1, -0.054}
      {-1.4, 0.64, -0.015, -0.098, 0.031, -0.12, -0.032, 0.055}
      {-1.4, 0.65, -0.013, -0.089, 0.051, -0.098, 0.013, 0.11, 0.0038}
      {-1.4, 0.67, -0.0079, -0.086, 0.071, -0.07, 0.054, 0.16, 0.0086, 0.0013}
```

We see that the MA coefficients $\hat{\theta}_{m,j}$, $j > 2$, lie within the bound $2/\sqrt{120} = 0.18 < 2\,\hat{\sigma}_j$, suggesting that $\theta_j = 0$ for $j > 2$. (Although one number $|\hat{\theta}_{9,8}| = 0.19$ is larger than 0.18, it is within the bound $2\,\hat{\sigma}_2$ given by (6.3).) Since the MA coefficients $\hat{\theta}_{m,j}$, $j \le 2$, are significantly different from zero, we conclude that $q = 2$.

The values of $\hat{\theta}_{m,1}$ and $\hat{\theta}_{m,2}$ stabilize around -1.5 and 0.78, respectively. In fact, for $m \ge 5$ the fluctuations in $\theta_{m,1}$ and $\theta_{m,2}$ are of order $1/\sqrt{n}$. So any $\theta_{m,1}$ and $\theta_{m,2}$ for $m \ge 5$ can be used as our estimated θ_1 and θ_2. We choose $m = 7$ and find the estimated parameters by selecting the first two coefficients of the MA(7) model. This can be done conveniently using *Mathematica*.

This extracts the MA(7) model and keeps only the first two MA coefficients.

```
In[22]:= MapAt[Take[#, 2] &, mamodels[[7]], 1]

Out[22]= MAModel[{-1.42865, 0.645337}, 0.926231]
```

Note that `Map` is used to perform an operation on all arguments while an operation on arguments only at specified locations is accomplished using `MapAt`.

Long AR Method

For a mixed model ARMA(p, q), the above methods are not applicable and we must resort to other techniques. The *long AR method* can be used to estimate the parameters of ARMA(p, q) models as well as AR(p) and MA(q) models. We may recall that an invertible ARMA model can be approximated by an AR(k) model for sufficiently large values of k. The long AR method of estimating ARMA parameters makes use of this fact. It first fits an AR(k) model (k large, thus "long" AR method) to the data $\{x_t\}$,

$$x_t = \phi_1 x_{t-1} + \ldots + \phi_k x_{t-k} + z_t.$$

The estimated AR(k) coefficients $\{\hat{\phi}_i\}$ are obtained using the ordinary least squares regression method. Then using the above equation with the $\{\hat{\phi}_i\}$ and data $\{x_t\}$, we can get the estimated residuals or innovations $\{\hat{z}_t\}$,

$$\hat{z}_t = x_t - \hat{\phi}_1 x_{t-1} - \ldots - \hat{\phi}_k x_{t-k}.$$

Substituting these residuals into the ARMA(p, q) model we have,

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \ldots + \phi_p x_{t-p} + \theta_1 \hat{z}_{t-1} + \theta_2 \hat{z}_{t-2} + \ldots + \theta_q \hat{z}_{t-q} + z_t.$$

Now we can again use the ordinary least squares regression method to estimate the coefficients $\{\phi_i\}$ and $\{\theta_i\}$, and the noise variance is given by the residual sum of squares divided by the number of terms in the sum,

$$\sum_{t=t'+1}^{n} z_t^2 \Big/ (n - t') \text{ where } t' = max\,(q + k, p). \tag{6.4}$$

This procedure of fitting an invertible ARMA(p, q) model to data using the long AR method is encoded in the function

`LongAREstimate[`*data*`, `*k*`, `*p*`, `*q*`]`,

where k is the order of long autoregression. Note that if the data are generated from a non-invertible model, long AR method fails and we may get nonsensical results.

Example 6.5 Fit an MA(2) model to the data in Example 6.4 using the long AR method.

The long AR method is used to obtain the estimated MA(2) model from `data`.

```
In[23]:= LongAREstimate[data, 8, 0, 2]

Out[23]= MAModel[{-1.43634, 0.663807}, 0.994936]
```

Example 6.6 Estimate the parameters of an ARMA(1, 2) model from the data generated below.

The random number generator is seeded first.

```
In[24]:= SeedRandom[31 857];
```

A time series of length 150 is generated from the given ARMA(1, 2) model.

```
In[25]:= data = TimeSeries[ARMAModel[{0.6}, {0.2, -0.5}, 2], 150];
```

This gives the estimated ARMA(1, 2) model using the long AR method. The order of the long autoregression is 10.

```
In[26]:= LongAREstimate[data, 10, 1, 2]

Out[26]= ARMAModel[{0.715687}, {0.0756134, -0.545209}, 1.4078]
```

Hannan-Rissanen Procedure

The idea behind the *Hannan-Rissanen procedure* of estimating ARMA parameters is very similar to that behind the long AR method. That is, an AR(k) model is first fitted to the data in order to obtain the estimates of the noise or innovation z_t; when this estimated noise $\hat{z}_t$ is used in place of the true noise it enables us to estimate ARMA(p, q) parameters using the less expensive method of least squares regression. However, the basic difference between the two methods is that in the long AR method the ARMA model orders to be fitted p and q as well as the order of long autoregression k, are supplied, whereas in the Hannan-Rissanen method the orders are determined within the procedure itself using an information criterion. Specifically, the Hannan-Rissanen procedure consists of following steps.

1. Use the Levinson-Durbin algorithm to fit models AR(i), for $i = 1, 2, \ldots, k_{max}$, to the data.
2. Calculate the AIC values $\ln \hat{\sigma}_i^2 + 2\,i/n$ for $i = 1, 2, \ldots, k_{max}$ of the fitted AR models, and choose the model AR(k) that yields the smallest AIC value.
3. Calculate the residuals $\{\hat{z}_t\}$ from the fitted AR(k) model.
4. Estimate ARMA(p, q) coefficients using the least squares method for $p \le Min\,(p_{max}, k)$ and $q \le q_{max}$; the estimated noise variance is given again by (6.4).
5. Select the model that has the lowest BIC value $(\ln \hat{\sigma}_{p,q}^2 + (p+q) \ln n / n)$.

For more details on the Hannan-Rissanen procedure see Granger and Newbold (1986), pp. 82–83. The function

`HannanRissanenEstimate[`*data*`,` *kmax*`,` *pmax*`,` *qmax*`]`

selects an appropriate ARMA model to fit the data using the Hannan-Rissanen procedure.

Example 6.7 Use Hannan-Rissanen method to fit an ARMA model to the previous data.

The Hannan-Rissanen procedure selects as well as estimates the model. It chooses an MA(1) model for the same data.

```
In[27]:= HannanRissanenEstimate[data, 10, 3, 3]

Out[27]= MAModel[{0.77312}, 1.43075]
```

Instead of getting the model with the lowest BIC value as selected by the Hannan-Rissanen procedure, sometimes it is useful to know the models that have the next lowest BIC values. This can be done by specifying the optional argument in `HannanRissanenEstimate`.

`HannanRissanenEstimate[`*data*`,` *kmax*`,` *pmax*`,` *qmax*`,` *h*`]`

gives *h* estimated models with increasing BIC values. Computationally this does not cost any extra time. It is advisable to compare these *h* different models and decide on one in conjunction with other model identification methods. If you want to get all the models estimated in step 4 of the Hannan-Rissanen procedure, you can give a sufficiently large value of *h*. You can then select a model from these using your own criterion.

Here we use the procedure to print out four models with the lowest BIC values.

```
In[28]:= hrmodels = HannanRissanenEstimate[data, 10, 3, 3, 4]

Out[28]= {MAModel[{0.77312}, 1.43075], ARMAModel[{0.06124}, {0.711995}, 1.42748],
          MAModel[{0.773019, 0.0241686}, 1.43976],
          ARMAModel[{-0.122753, 0.142481}, {0.893641}, 1.41075]}
```

To find the BIC and AIC values of these models we can use `Map`.

We find the BIC values of each of the above four models. These values are in increasing order as they should be.

```
In[29]:= Map[BIC[#, 150] &, %]

Out[29]= {0.391605, 0.422722, 0.431288, 0.444332}
```

The AIC values of the models can be found similarly.

```
In[30]:= Map[AIC[#, 150] &, %%]

Out[30]= {0.371535, 0.38258, 0.391146, 0.384119}
```

We see that while the MA(1) model has the lowest BIC value, the ARMA(1, 2) model has the lowest AIC value. Next we plot the sample correlation.

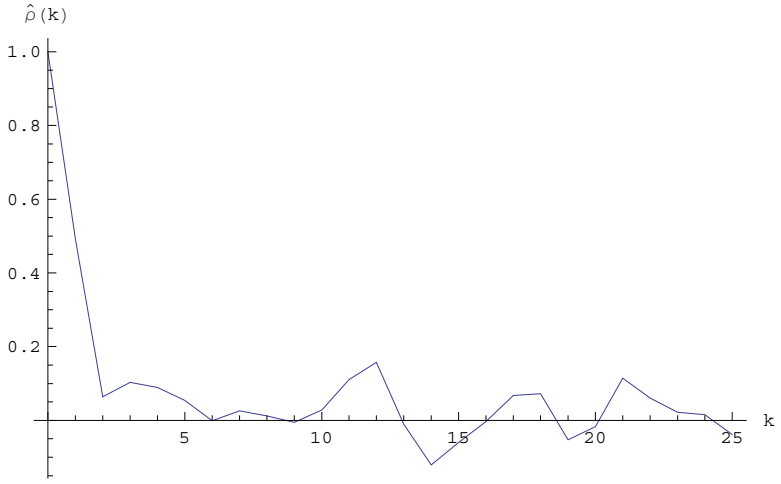
This calculates the sample correlation function up to lag 25.

```
In[31]:= corr = CorrelationFunction[data, 25];
```

The above sample correlation function is plotted here.

```
In[32]:= plotcorr[corr, Joined -> True, AxesLabel -> {"k", "ρ̂(k)"}]
```

Out[32]=



The bound for the sample correlation function is calculated under the assumption that $q = 1$.

```
In[33]:= 2 Sqrt[(1 + 2 corr[[2]]^2) / 150.]

Out[33]= 0.19894
```

We see that although $\hat{\rho}(10)$ is beyond this bound, there is no strong evidence to reject the MA(1) model. So based on the above analysis either MA(1) or ARMA(1, 2) can be tentatively identified as the right model.

Maximum Likelihood Method

When the noise of a zero-mean, stationary ARMA process is normally distributed ($\{Z_t\} \sim N(0, \sigma^2)$), we can get the estimate of ARMA parameters by maximizing the Gaussian likelihood of the process. The parameters so obtained are called the *maximum likelihood estimates* of the parameters. The exact likelihood can be obtained from the prediction error decomposition (see Section 1.7) with the prediction errors being computed using the innovations algorithm. A complete discussion of the innovations algorithm and the calculation of the exact maximum likelihood can be found in Brockwell and Davis (1987), Chapter 5 and Sections 8.3 and 8.4.

The function

`LogLikelihood[`*data*`,` *model*`]`

gives the logarithm of the exact Gaussian likelihood function for the given model and data. In the univariate case, `LogLikelihood` gives what is called the "reduced likelihood" (see Brockwell and Davis (1987), p. 250) and in the multivariate case, it gives the logarithm of the full maximum likelihood (apart from a constant). See Reinsel (1993), Section 5.4.

Example 6.8 Calculate the logarithm of the likelihood of the estimated ARMA(1, 2) model in Example 6.7.

Here is the logarithm of the likelihood of the ARMA(1, 2) model estimated in Example 6.7.

```
In[34]:= LogLikelihood[data, hrmodels[[2]]]

Out[34]= -0.372637
```

The maximum likelihood method of estimating model parameters is often favored because it has the advantage among others that its estimators are more efficient (*i.e.*, have smaller variance) and many large-sample properties are known under rather general conditions.

MLEstimate[*data*, *model*, {ϕ1, {$\phi 1_1$, $\phi 1_2$}}, ...]

fits *model* to *data* using the maximum likelihood method. The parameters to be estimated are given in symbolic form as the arguments to *model*, and two initial numerical values for each parameter are required. Internally the maximization of the exact logarithm of the likelihood is done using the *Mathematica* built-in function FindMinimum and the same options apply. (To find out all the options of FindMinimum use Options[FindMinimum].)

Example 6.9 Use the maximum likelihood method to fit an MA(1) model and an ARMA(1, 2) model to the same data in Example 6.6.

The maximum likelihood estimate of the MA(1) model is obtained using MLEstimate.

```
In[35]:= model1 = MLEstimate[data, MAModel[{θ1}, 1], {θ1, {0.73, 0.74}}]

Out[35]= MAModel[{0.85219}, 1.40225]
```

Similarly, this is the maximum likelihood estimate of the ARMA(1, 2) model.

```
In[36]:= model2 = MLEstimate[data, ARMAModel[{ϕ1}, {θ1, θ2}, 1],
           {ϕ1, {0.1, 0.2}}, {θ1, {0.6, 0.8}}, {θ2, {0.1, 0.3}}]

Out[36]= ARMAModel[{-0.583367}, {1.36826, 0.415562}, 1.38683]
```

This gives the logarithm of the likelihood of the latter model.

```
In[37]:= LogLikelihood[data, %]

Out[37]= -0.336211
```

A couple of points of explanation are in order before we look at the results of our estimation. First, choosing good starting values for the parameters to be estimated is very important in the maximum likelihood estimation since not only can bad initial values slow down the calculation, but also abort the calculation if they reach the parameter region corresponding to nonstationary models. Since they are expected to be close to the values that maximize the likelihood, the estimated parameter values from other estimation methods are usually used as initial values for the maximum likelihood estimation as we have done here. This is why some authors call all the estimates of ARMA parameters that are not based on maximizing the likelihood function preliminary estimates, and use them primarily as initial values in maximizing the likelihood. Second, the alert reader may have noticed that the noise variance is not entered as a symbolic parameter. This is because in the univariate case, the likelihood function is independent of the noise variance and the noise variance is estimated separately at the end. So it does not matter what we input for the noise variance in the argument of the model, it is simply ignored. However, it is important that the noise variance should *not* be included in the parameter search list.

This is because if extra parameters, which do not affect the value of the function being minimized, are included in the search list, `FindMinimum` will fail to find a minimum.

Now we can select our model using AIC or BIC.

The MA(1) model is favored by AIC.

```
In[38]:= {AIC[model1, 150], AIC[model2, 150]}

Out[38]= {0.35141, 0.367018}
```

The MA(1) model is also favored by BIC.

```
In[39]:= {BIC[model1, 150], BIC[model2, 150]}

Out[39]= {0.371481, 0.427231}
```

Both AIC and BIC favor the MA(1) model over the ARMA(1, 2) model. This result may be surprising, after all, we know that the data were generated from an ARMA(1, 2) model. However, when n is small it is possible that other models can be fitted better to the data. More importantly, if the AR and MA polynomials $\phi(x)$ and $\theta(x)$ of an ARMA model have an approximate common factor of degree g, an ARMA$(p-g, q-g)$ model rather than an ARMA(p, q) model is likely to be identified using information criteria.

In our example, the MA polynomial $\theta(x) = 1 + 0.2\,x - 0.5\,x^2$ can be factorized as $\theta(x) = (1 - 0.614143\,x)\,(1 + 0.814143\,x)$.

The MA polynomial $\theta(x) = 1 + 0.2\,x - 0.5\,x^2$ contains a factor close to $(1 - 0.6\,x)$.

```
In[40]:= Expand[(1 - 0.614143 x) (1 + 0.814143 x)]

Out[40]= 1 + 0.2 x - 0.5 x^2
```

We see that $\phi(x) = (1 - 0.6\,x)$ and $\theta(x)$ share an approximate factor. This is why the data behave like an MA(1) model with $\theta_1 \approx 0.81$.

The large discrepancy between the estimated ARMA(1, 2) parameters and the "true" model parameters $\phi_1 = 0.6$, $\theta_1 = 0.2$, and $\theta_2 = -0.5$ can also be explained by this accidental presence of an approximate common factor. Consider a series of data $\{x_t\}$ from an ARMA(p, q) model, $\phi(B)\,x_t = \theta(B)\,z_t$. It also satisfies an ARMA$(p+g, q+g)$ model $c(B)\,\phi(B)\,x_t = c(B)\,\theta(B)\,z_t$, where $c(B)$ is a polynomial of degree g. If we fit an ARMA$(p+g, q+g)$ model to the data, there is no unique solution and the maximum likelihood method can show strong dependence on the initial conditions.

For example, if we change the initial conditions, we get quite different estimates from those in `Out[36]`.

```
In[41]:= MLEstimate[data, ARMAModel[{ϕ1}, {θ1, θ2}, 1],
           {ϕ1, {0.1, 0.2}}, {θ1, {0.7, 0.75}}, {θ2, {-0.1, -0.2}}]

Out[41]= ARMAModel[{-0.583366}, {1.36826, 0.415561}, 1.38683]
```

Conditional Maximum Likelihood Method

It is well known that the exact maximum likelihood estimate can be very time consuming especially when $p + q$ is large. (We warn the user here that the function `MLEstimate` can be rather slow for large n or large number of parameters.) Therefore, often, when the data length n is reasonably large, an approximate likelihood function is used in order to speed up the calculation. The approximation is made by setting the initial q values of the noise $z_p, z_{p-1}, \ldots, z_{p-q+1}$ to $E\,Z_t = 0$ and fixing the first p values of X. The likelihood function so obtained is called the *conditional likelihood function* (conditional on $z_p = z_{p-1} = \ldots = z_{p-q+1} = 0$ and on $X_1, X_2, \ldots, X_p$ being fixed at $x_1, x_2, \ldots, x_p$). Maximizing the conditional likelihood function is equivalent to minimizing the conditional sum of squares function. So the conditional maximum likelihood estimate is the same as the conditional nonlinear least squares estimate, and it is used as an approximation to the exact maximum likelihood estimate. Unlike the case of the least squares estimate, the conditional maximum likelihood estimate of the noise variance is given by the minimum value of the sum of squares divided by the effective number of observations $n - p$. The details of the conditional likelihood estimate are explained in Harvey (1981), Sections 5.2 and 5.3. See also Wilson (1973).

The function

`ConditionalMLEstimate[`*data*`,` *model*`]`

can be used to fit *model* to *data* using the conditional maximum likelihood estimate. It numerically searches for the maximum of the conditional logarithm of the likelihood function using the Levenberg-Marquardt algorithm with initial values of the parameters to be estimated given as the arguments of *model*.

Example 6.10 Fit a model to the data in Example 6.6 using the conditional maximum likelihood method.

We use the result from the Hannan-Rissanen estimate as our initial values. We can input the output of the earlier calculation directly.

The first model in `hrmodels` is used as the initial model to get the conditional maximum likelihood estimate of an MA(1) model.

```
In[42]:= ConditionalMLEstimate[data, hrmodels[[1]]]

Out[42]= MAModel[{0.860552}, 1.40202]
```

The second model in `hrmodels` is used as the initial model to get the conditional maximum likelihood estimate of the ARMA(1, 2) model.

```
In[43]:= arma12 = ConditionalMLEstimate[data, hrmodels[[2]]]

Out[43]= ARMAModel[{-0.0895185}, {0.885883}, 1.40502]
```

For a pure AR model, the conditional maximum likelihood estimate reduces to a linear ordinary least squares estimate and no initial parameter values are necessary. We can if we like simply use

`ConditionalMLEstimate[`*data*`,` *p*`]`

to fit an AR(*p*) model to *data*.

The Asymptotic Properties of the Maximum Likelihood Estimators

Often we would like to know the approximate standard errors associated with our parameter estimates. These errors can be obtained from the asymptotic covariance of the estimators. Let $\beta = (\phi_1, \phi_2, \ldots, \phi_p, \theta_1, \theta_2, \ldots, \theta_q)'$ be the parameters of a stationary and invertible ARMA(p, q) model and $\hat{\beta}$ the maximum likelihood or conditional maximum likelihood estimator of β. Then, it can be shown that as $n \to \infty$,

$$n^{1/2}(\hat{\beta} - \beta) \Rightarrow N(\mathbf{0}, V(\beta)).$$

To calculate V for a univariate ARMA model we can use the function

`AsymptoticCovariance[`*model*`]`.

The function uses the algorithm given in Godolphin and Unwin (1983).

This gives the asymptotic covariance matrix V of the estimators of an MA(3) model.

```
In[44]:= AsymptoticCovariance[MAModel[{θ1, θ2, θ3}, σ^2]];
```

We display the covariance in matrix form.

```
In[45]:= MatrixForm[%]
```

```
Out[45]//MatrixForm=
⎛ 1 - θ3^2          θ1 - θ2 θ3          θ2 - θ1 θ3        ⎞
⎜ θ1 - θ2 θ3        1 + θ1^2 - θ2^2 - θ3^2  θ1 - θ2 θ3    ⎟
⎝ θ2 - θ1 θ3        θ1 - θ2 θ3          1 - θ3^2          ⎠
```

Similarly, the asymptotic covariance matrix V of the estimators of an ARMA(1, 1) model is obtained but not displayed.

```
In[46]:= AsymptoticCovariance[ARMAModel[{ϕ1}, {θ1}, σ^2]];
```

This is the result after simplification.

```
In[47]:= Simplify[%] // MatrixForm
```

```
Out[47]//MatrixForm=
⎛ -((1+θ1 ϕ1)^2 (-1+ϕ1^2))/(θ1+ϕ1)^2     -((-1+θ1^2) (1+θ1 ϕ1) (-1+ϕ1^2))/(θ1+ϕ1)^2 ⎞
⎝ -((-1+θ1^2) (1+θ1 ϕ1) (-1+ϕ1^2))/(θ1+ϕ1)^2   -((-1+θ1^2) (1+θ1 ϕ1)^2)/(θ1+ϕ1)^2   ⎠
```

Although the function `AsymptoticCovariance` works for univariate ARMA models of arbitrary orders, the reader must be warned that with symbolic parameters the expressions can get very complicated with mixed models of even relatively small orders. However, with numerical parameters the covariance matrix is simply a $(p + q) \times (p + q)$ numerical matrix.

In practice, the true parameter values are not known and the asymptotic covariance matrix is estimated by replacing the true model parameters with the estimated parameters. Another way of getting the approximate

asymptotic covariance matrix is to calculate the approximate information matrix whose inverse is the desired covariance matrix. The approximation involves replacing the expectations with sample quantities. The function

InformationMatrix[*data*, *model*]

gives the estimated asymptotic information matrix from *model* and *data*.

Example 6.11 Estimate the asymptotic covariance matrix of the estimated ARMA(1, 2) parameters in Example 6.10 using `AsymptoticCovariance` and inversion of `InformationMatrix`.

This is the estimated asymptotic covariance V using the function `AsymptoticCovariance`.

```
In[48]:= AsymptoticCovariance[arma12]

Out[48]= {{1.32592, -0.309932}, {-0.309932, 0.287658}}
```

We can also estimate V by inverting the information matrix.

```
In[49]:= Inverse[InformationMatrix[data, arma12]]

Out[49]= {{1.32419, -0.307477}, {-0.307477, 0.286746}}
```

The standard errors of $\hat{\phi}_1$, $\hat{\theta}_1$, and $\hat{\theta}_2$ are given by the diagonal elements of $\sqrt{V/n}$.

This gives the standard errors of the estimated parameters.

```
In[50]:= Sqrt[Diagonal[%] / 150]

Out[50]= {0.093957, 0.0437223}
```

Multivariate Case

All of the above parameter estimation methods can be generalized to include multivariate models. The basic ideas are the same although the derivation and notation are considerably more involved. All the functions with the exception of `AsymptoticCovariance` given so far have been coded so that they can be used directly for multivariate cases.

However, for maximum likelihood estimate there is one difference between the univariate and multivariate cases. In the multivariate case, the likelihood function is dependent on the covariance matrix, so the covariance matrix elements should be input as symbolic parameters and included in the parameter search list. The estimation of multivariate ARMA model can potentially involve large number of parameters even for small p and q. This can make the exact maximum likelihood estimation of the parameters computationally very slow. If in some cases one or more parameter values are known, then they should not be input as symbolic parameters, and they should not be included in the parameter search list. For example, in a bivariate AR(1) process, if for some reason $(\phi_1)_{12}$ is fixed to be 0 then `ARModel[{{{p1, 0}, {p3, p4}}, {{s1, s2}, {s2, s4}}]` should be input to the function `MLEstimate` reducing the number of parameters to six.

Example 6.12 Fit a model to the given bivariate data using the Hannan-Rissanen method.

The random number generator is seeded first.

```
In[51]:= SeedRandom[63 771]
```

This generates a bivariate time series of length 200 from the given ARMA(1, 1) model.

```
In[52]:= data1 = TimeSeries[ARMAModel[{{{0.5, 0.2}, {-0.1, 0.4}}},
             {{{0.8, -0.4}, {0.6, 0.3}}}, {{1, 0}, {0, 1}}], 200];
```

The Hannan-Rissanen procedure is used to select three models.

```
In[53]:= hrmodels = HannanRissanenEstimate[%, 5, 5, 5, 3]

Out[53]= {ARMAModel[{{{0.487217, 0.281219}, {-0.059466, 0.423744}}},
           {{{0.778252, -0.425392}, {0.600632, 0.276561}}},
           {{0.958697, 0.0856119}, {0.0856119, 0.837137}}],
          ARMAModel[{{{1.22996, -0.125299}, {0.485357, 0.638093}}, {{-0.779748, 0.346898},
             {-0.723771, -0.0687931}}, {{0.295731, -0.0832667}, {0.318696, -0.0143152}}},
           {{0.888135, 0.046069}, {0.046069, 0.83046}}],
          ARMAModel[{{{0.862036, -0.231722}, {0.316163, 0.136316}},
            {{-0.267936, 0.352856}, {-0.298955, 0.165516}}},
           {{{0.402308, 0.0878135}, {0.223573, 0.569388}}},
           {{0.91445, 0.0637247}, {0.0637247, 0.809764}}]}
```

We obtain the orders of each of the three selected models.

```
In[54]:= If[Head[#] === ARMAModel,
            Head[#][Length[#[[1]]], Length[#[[2]]], Head[#][Length[#[[1]]]] & /@ %

Out[54]= {ARMAModel[1, 1], ARModel[3], ARMAModel[2, 1]}
```

We choose ARMA(1, 1) as our model, and use the conditional maximum likelihood method to further estimate the model parameters.

We obtain the conditional maximum likelihood estimate of the selected model.

```
In[55]:= ConditionalMLEstimate[data1, hrmodels[[1]]]

Out[55]= ARMAModel[{{{0.53379, 0.187977}, {-0.0642834, 0.375942}}},
           {{{0.721899, -0.374849}, {0.626417, 0.375031}}},
           {{0.926471, 0.0453515}, {0.0453515, 0.779338}}]
```

Next we calculate the standard errors of the estimated parameters.

We first compute the asymptotic covariance matrix.

```
In[56]:= Inverse[InformationMatrix[data1, %]];
```

The standard errors of $\{\hat{\Phi}_{1_{11}}, \hat{\Phi}_{1_{12}}, \hat{\Phi}_{1_{21}}, \ldots, \hat{\Theta}_{1_{22}}\}$ are given by the diagonal elements of $\sqrt{V/n}$.

The standard errors are obtained.

```
In[57]:= Sqrt[Diagonal[%] / 200]

Out[57]= {0.0709004, 0.0977617, 0.0636704,
          0.0930485, 0.0688324, 0.0718582, 0.0750902, 0.0916497}
```

1.6.2 Diagnostic Checking

After fitting a model to a given set of data the goodness of fit of the model is usually examined to see if it is indeed an appropriate model. If the model is not satisfactory, modifications are made to the model and the whole process of model selection, parameter estimation, and diagnostic checking must be repeated until a satisfactory model is found.

Residual Testing

There are various ways of checking if a model is satisfactory. The commonly used approach to diagnostic checking is to examine the residuals. There are several alternative definitions of the residuals and here we define the residuals to be the noise calculated from the estimated model. Let $\{\hat{\phi}_i\}$ and $\{\hat{\theta}_i\}$ be estimated ARMA parameters and $\hat{\phi}(x)$ and $\hat{\theta}(x)$ the corresponding AR and MA polynomials, the residual $\hat{z}_t$ is defined by

$$\hat{\phi}(B)\, x_t = \hat{\theta}(B)\, \hat{z}_t. \tag{6.5}$$

Or explicitly,

$$\hat{z}_t = \hat{\theta}^{-1}(B)\, \hat{\phi}(B)\, x_t \text{ for } t = 1, \ldots, n, \tag{6.6}$$

where $x_t = 0$ for $t \le 0$. From (6.5) we see that if the fitted model is the "true" model, the residuals $\{\hat{z}_t\}$ should behave like a white noise process with zero mean and constant variance. So the behavior of the residuals is used to test the adequacy of the fitted model.

The first step in residual testing is to calculate the residuals given the fitted model and observed data. The function

`Residual[`*data*`,` *model*`]`

gives the residuals $\{\hat{z}_t\}$ for $t = 1, 2, \ldots, n$ given the fitted model and observed data. Internally the function iterates (6.6) with the starting values $x_t = 0$ for $t \le 0$ and $\hat{z}_t = 0$ for $t \le 0$.

Example 6.13 Check if the fitted MA(1) model in Example 6.9 is adequate.

The residuals are calculated first.

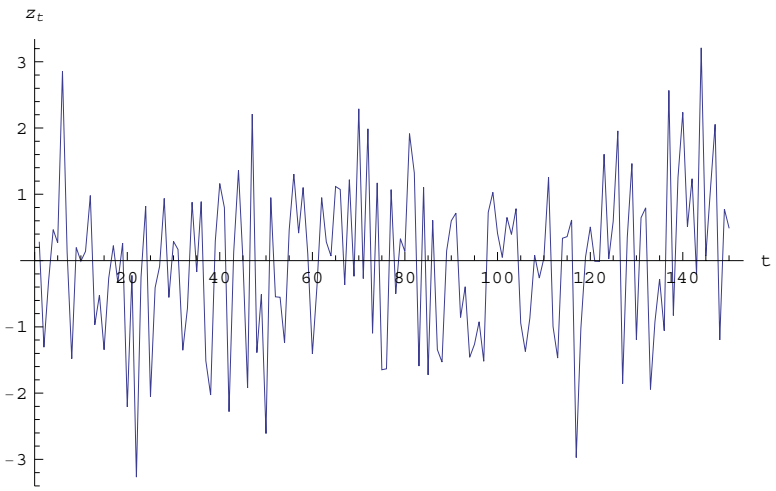
```
In[58]:= res = Residual[data, model1];
```

Since the residuals are also ordered in time, we can treat them as a time series. As in the analysis of the time series itself, the first diagnostic test in examining residuals is to plot them as a function of time to see if it appears to be a stationary random sequence.

This plots the residuals.

```
In[59]:= ListLinePlot[res, AxesLabel -> {"t", "z_t"}]
```

Out[59]=



The graph gives no indication of deviation from stationary random noise. Often we also calculate the sample correlation function of the residuals $\hat{\rho}_z(k)$ and see if it behaves the way it should.

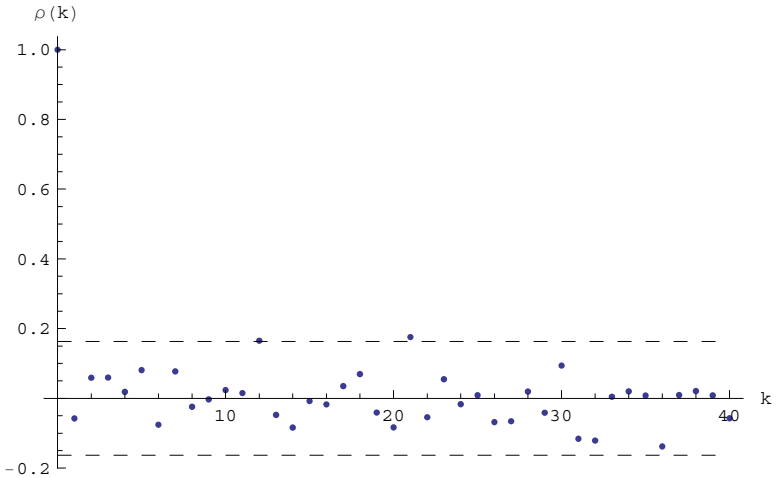
Correlation function of the residuals up to lag 40 is calculated.

```
In[60]:= corr = CorrelationFunction[res, 40];
```

The correlation function is plotted along with the bounds $\pm 1/\sqrt{n}$.

```
In[61]:= Show[plotcorr[corr], Plot[{2 / Sqrt[150], -2 / Sqrt[150]}, {x, 0, 40},
           PlotStyle -> {{Black, Dashing[{0.02}]}}], AxesLabel -> {"k", "ρ(k)"}]
```

Out[61]=



Although the sample correlation of the residuals has smaller asymptotic variance than $1/n$ for small lags (see Brockwell and Davis (1987), pp. 297–300), we nevertheless use $\pm 2/\sqrt{n}$ as a rough guide to see if the correlations are significantly different from zero. In our example, $2/\sqrt{150} = 0.16$, and we see no reason to believe that the residuals are not white noise.

Instead of looking at the correlation function of the residuals $\hat{\rho}_z(k)$ at each k, we can also look at the first h correlation values together and test if the first h correlations are zero (H_0). The *portmanteau test* is based on the statistic

$$Q_h = n(n+2)\sum_{k=1}^{h} \hat{\rho}_z(k)^2 / (n-k),$$

which has an asymptotic χ^2 distribution with $h-p-q$ degrees of freedom. If $Q_h > \chi^2_{1-\alpha}(h-p-q)$, the adequacy of the model is rejected at level α. The function

`PortmanteauStatistic[`*residual*`, h]`

gives the value of Q_h for given residuals and h. Here we calculate the portmanteau statistic using the residuals we obtained above for $h = 35$.

This gives the portmanteau statistic Q_{35}.

```
In[62]:= PortmanteauStatistic[res, 35]

Out[62]= 29.2422
```

This number is compared with $\chi^2_{0.95}(34)$, which can be obtained as follows.

This gives $\chi^2_{0.95}(34)$.

```
In[63]:= Quantile[ChiSquareDistribution[34], 0.95]

Out[63]= 48.6024
```

Since this number exceeds the portmanteau statistic, we accept the fitted model `model1` as adequate.

For an m-variate time series the portmanteau statistic Q_h is (see Granger and Newbold (1986), p. 250 and Reinsel (1993), p. 134)

$$Q_h = n^2 \sum_{k=1}^{h} Tr\left(\hat{\Gamma}'(k)\,\hat{\Gamma}^{-1}(0)\,\hat{\Gamma}(k)\,\hat{\Gamma}^{-1}(0)\right) / (n-k), \tag{6.7}$$

where $\hat{\Gamma}$ is the covariance matrix of residuals. The statistic has an asymptotic χ^2 distribution with $m^2(h-p-q)$ degrees of freedom. The same function `PortmanteauStatistic` can be used to compute (6.7).

There are various other tests for checking the randomness of the residuals. These tests are often easily implemented with *Mathematica* and the reader is urged to try them. Here we show how the turning points test and difference-sign test can be implemented. (See Kendall and Ord (1990), pp. 18–23.)

Turning Points Test

Let $\{x_1, x_2, \ldots, x_n\}$ be a time series. x_i is a turning point if $x_{i-1} < x_i$ and $x_{i+1} < x_i$ or $x_{i-1} > x_i$ and $x_{i+1} > x_i$. In other words, a turning point is a local extremum (a peak or a trough). Let T denote the number of turning points. If the series is a realization of an identically and independently distributed (IID) random process, then T has an asymptotic normal distribution with mean $\mu_T = 2(n-2)/3$ and variance $\sigma_T^2 = Var(T) = (16n - 29)/90$.

Example 6.14 Use the turning points test to see if the above residuals ($n = 150$) are random.

`temp` is a list consisting of the signs of $x_i - x_{i-1}$ for $i = 2, 3, \ldots, n$.

```
In[64]:= temp = Sign[Differences[res]];
```

`Count` counts the number of occurrences of the adjacent elements with opposite signs, that is, the number of turning points.

```
In[65]:= Count[Rest[temp] Most[temp], -1]

Out[65]= 104
```

This calculates $(T - \mu_T)/\sigma_T$.

```
In[66]:= N[(98 - 2 (150 - 2) / 3) / Sqrt[(16 * 150 - 29) / 90]]

Out[66]= -0.129887
```

The result is well within two standard deviations.

The Difference-Sign Test

In this test, the number of times $x_i > x_{i-1}$, or $x_i - x_{i-1}$ is positive is counted. If the series is random we expect this number to be $(n-1)/2$. If the number of times $x_i > x_{i-1}$ is far from $(n-1)/2$, a trend is likely present in the series.

Using `Count` we can easily get the number of times $x_i - x_{i-1}$ is positive.

```
In[67]:= Count[Sign[Differences[res]], 1]

Out[67]= 80
```

The deviation of this count from the mean $(n-1)/2$ can then be compared with the standard deviation $\sigma = ((n+1)/12)^{1/2}$. In this case the deviation is within two σ.

1.7 Forecasting

Now that we have explored methods to estimate the parameters of an appropriately chosen model we turn to one of the main purposes of time series analysis, forecasting or predicting the future values of a series. In this section we discuss some forecasting methods commonly used in time series analysis. We first present the best linear predictor and its derivation in the infinite sample limit. Then we derive the approximate best linear predictor often used to speed up the calculation. We show how to write a program to update the prediction formula when new data are available and also introduce the simple exponential smoothing forecast procedure.

1.7.1 Best Linear Predictor

Suppose that the stationary time series model that is fitted to the data $\{x_1, x_2, \ldots, x_n\}$ is known and we would like to predict the future values of the series $X_{n+1}, X_{n+2}, \ldots, X_{n+h}$ based on the realization of the time series up to time n. The time n is called the *origin* of the forecast and h the *lead time*. A linear predictor is a linear combination of $\{X_1, X_2, \ldots, X_n\}$ for predicting future values; the *best linear predictor* is defined to be the linear predictor with the minimum mean square error. Let $\hat{X}_n(h) = \sum_{i=0}^{n-1} a_i(h) X_{n-i}$ denote the linear predictor for X_{n+h} at lead time h with the origin n and $e_n(h) = X_{n+h} - \hat{X}_n(h)$ the forecast error. Finding the best linear predictor is reduced to finding the coefficients $a_i(h)$, $i = 0, 1, \ldots, n-1$, such that the mean square error $E\, e_n^2(h) = E\left(X_{n+h} - \hat{X}_n(h)\right)^2$ is a minimum.

Although the idea is straightforward, the derivation of the best linear predictor is too involved to be presented here. A detailed derivation of the best linear predictor using the projection theorem and the innovations algorithm is provided in Brockwell and Davis (1987), Chapter 5, pp. 159–177.

The function

`BestLinearPredictor[`*data*`,` *model*`,` *h*`]`

gives the best linear prediction and its mean square error up to h time steps ahead based on the finite sample data and given model. It uses the innovations algorithm to calculate the forecasts and their errors. Here the errors are obtained under the assumption that the model is known exactly. Estimated model parameters can give rise to additional errors. However, they are generally negligible when n is large. See the discussion in Harvey (1981), p. 162.

Example 7.1 In Example 6.9 we have fitted an MA(1) model to the data generated in Example 6.6. Use this information to find the best linear prediction for the next five values in the series.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

This seeds the random number generator.

```
In[2]:= SeedRandom[31 857];
```

We generate the data as in Example 6.6.

```
In[3]:= data = TimeSeries[ARMAModel[{0.6}, {0.2, -0.5}, 2], 150];
```

This is the estimated model from Example 6.9.

```
In[4]:= model1 = MAModel[{0.765687}, 2.13256]

Out[4]= MAModel[{0.765687}, 2.13256]
```

We predict the next 5 values of the series based on the data and the fitted model `model1`.

```
In[5]:= BestLinearPredictor[data, model1, 5]

Out[5]= {{0.474638, 0., 0., 0., 0.}, {2.13256, 3.38283, 3.38283, 3.38283, 3.38283}}
```

The first entry of the above list is the prediction for the next five values of the series; the second entry, mean square errors of the prediction. Note that for an MA(q) model, $\hat{X}_n(h) = 0$ for $h > q$ since after q time steps we lose all previous information and the best we can predict is the mean value of the series which is zero.

1.7.2 Large Sample Approximation to the Best Linear Predictor

It turns out that in the large sample limit ($n \to \infty$) the best linear predictor can be derived and calculated in a much simpler fashion. In the following we derive the best linear predictor in the infinite sample case, and from there derive an approximate formula for calculating the best linear predictor when n is large.

Best Linear Predictor in the Infinite Sample Case

A stationary ARMA(p, q) process at time $n + h$ is given by

$$X_{n+h} = \phi_1 X_{n+h-1} + \phi_2 X_{n+h-2} + \ldots + \phi_p X_{n+h-p} + Z_{n+h} + \theta_1 Z_{n+h-1} + \ldots + \theta_q Z_{n+h-q}. \tag{7.1}$$

Since X_t is a random variable, it is reasonable to use its expectation as our predicted value, taking into account the information we have for X_t at $t = n, n-1, \ldots$. (This subsection is the only place where we assume that we have data extending to the infinite past.) So if $\hat{X}_n(h)$ denotes the forecast h steps ahead, we define $\hat{X}_n(h) = E(X_{n+h} \mid X_n, X_{n-1}, \ldots)$, the conditional expectation of X_{n+h} given $\{X_n, X_{n-1}, \ldots\}$. On taking the conditional expectation on both sides of (7.1) and letting $\hat{Z}_n(t) = E(Z_{n+t} \mid X_n, X_{n-1}, \ldots, X_1, \ldots)$, we obtain,

$$\hat{X}_n(h) = \phi_1 \hat{X}_n(h-1) + \phi_2 \hat{X}_n(h-2) + \ldots + \hat{Z}_n(h) + \theta_1 \hat{Z}_n(h-1) + \ldots + \theta_q \hat{Z}_n(h-q). \tag{7.2}$$

Equation (7.2) can be used recursively to obtain the forecast values of X_{n+h} for $h = 1, 2, \ldots$ once we know the right-hand side of (7.2). It is easy to see that for $h \le 0$, $\hat{X}_n(h)$ and $\hat{Z}_n(h)$ are simply the realized values of X_{n+h} and Z_{n+h}, respectively,

$$\hat{X}_n(h) = X_{n+h}, \tag{7.3}$$

and

$$\hat{Z}_n(h) = Z_{n+h}; \tag{7.4}$$

and for $h > 0$, since the future values of the noise are independent of X_t ($t \le n$), we have

$$\hat{Z}_n(h) = 0. \tag{7.5}$$

$\hat{X}_n(h)$ obtained from (7.2) using (7.3) to (7.5) is, in fact, the best linear predictor. To see this we show that the mean square forecast error of $\hat{X}_n(h)$ is minimum. Consider an arbitrary predictor $\tilde{X}_n(h)$ which is linear in X_i, $i \le n$. It can be rewritten in terms of $\{Z_t\}$ as $\tilde{X}_n(h) = \sum_{i=h}^{\infty} \tilde{\psi}_i Z_{n+h-i}$. The sum starts at $i = h$ because future noise has no influence on our prediction. Now consider its mean square error

$$E\left(X_{n+h} - \tilde{X}_n(h)\right)^2 = \sigma^2 \left(\sum_{j=0}^{h-1} \psi_j^2 + \sum_{j=h}^{\infty} \left(\psi_j - \tilde{\psi}_j\right)^2 \right), \tag{7.6}$$

where we have used the expansion $X_{n+h} = \sum_{j=0}^{\infty} \psi_j Z_{n+h-j}$ (see (2.9)). The mean square error in (7.6) achieves its minimum value if $\psi_j = \tilde{\psi}_j$. But this is exactly the case for the expansion of $\hat{X}_n(h)$ in terms of $\{Z_t\}$ since (7.2) has the same form as the ARMA equation governing X_{n+h}, (7.1). Therefore, $\hat{X}_n(h)$ is the desired best linear predictor. Its forecast error is given by

$$e_n(h) = \sum_{j=0}^{h-1} \psi_j Z_{n+h-j} \tag{7.7}$$

and its mean square forecast error is given by

$$E\left(X_{n+h} - \hat{X}_n(h)\right)^2 = \sigma^2 \sum_{j=0}^{h-1} \psi_j^2. \tag{7.8}$$

Approximate Best Linear Predictor

Where does the assumption of an infinite sample enter in the above derivation? It is used when we replace $E(Z_t \mid X_n, X_{n-1}, \dots)$ by Z_t for $t \le n$ (see (7.4)). This is true only if we know the series all the way back to the infinite past (*i.e.*, we have an infinite sample) since knowing a finite number of data points $X_n, \dots, X_1$ does not determine Z_t completely. To see this we recall that an invertible ARMA model can be written as $Z_t = \theta^{-1}(B)\,\phi(B)\,X_t = \sum_{i=0}^{\infty} \pi_i X_{t-i}$. So only if we have infinite data points can we replace the conditional expectation by Z_t. Although in practice we invariably have a finite number of observations, the above derivation of the best linear predictor in the infinite sample limit nevertheless enables us to develop a way of calculating the approximate best linear predictor when n is large.

Let $\hat{X}_n(h) = E(X_{n+h} \mid X_n, X_{n-1}, \dots, X_1)$ and $\hat{Z}_n(h) = E(Z_{n+h} \mid X_n, X_{n-1}, \dots, X_1)$. For an invertible model, the π weights decrease exponentially, and for large n it is a good approximation to truncate the infinite sum and write,

$$Z_{n+h} = \sum_{i=0}^{n+h-1} \pi_i X_{n+h-i}. \tag{7.9}$$

Note that Z_{n+h} in (7.9) is just the residual defined in (6.6) since truncating the infinite sum is the same as setting $X_t = 0$ for $t \le 0$. Under this approximation we again arrive at $\hat{Z}_n(h) = Z_{n+h}$ for $h \le 0$, the same result as in (7.4). With (7.3) to (7.5) and (7.9), (7.2) provides a recursive way of computing the predicted values of X_{n+h} for $h = 1, 2, \dots$. This is often used as an approximate best linear predictor in the finite but large sample case to speed up the calculation. However, we must keep in mind that only when n is sufficiently large and the model is invertible is the approximation good.

Although (7.9) is used to get the approximate predictor for the finite sample case, the mean square error of the best linear predictor in the infinite sample case, (7.8), is used to approximate that in the finite sample case. This can underestimate the real error corresponding to the given predictor, but it makes little difference when the model is invertible and n is large. To get the approximate best linear predictor and its mean square error defined by (7.2) to (7.5), (7.9), and (7.8), we can simply use the same function for getting the exact best linear predictor `BestLinearPredictor` and set its option `Exact` to `False`.

In the rest of the section we give some examples of using `BestLinearPredictor` to get both exact and approximate best linear predictions.

Example 7.2 For an AR(1) process $X_{t+1} = \phi_1 X_t + Z_t$, (7.2) gives $\hat{X}_n(1) = \phi_1 X_n$ and $\hat{X}_n(h) = \phi_1^h X_n$. The mean square error is $\sigma^2 \sum_{i=0}^{h-1} \phi_1^{2i}$, which is obtained by first noting that $\psi_j = \phi_1^j$ and by using (7.8).

These are the predicted values and their mean square errors.

```
In[6]:= BestLinearPredictor[{x1, x2, x3}, ARModel[{ϕ1}, σ^2], 4]

Out[6]= {{x3 ϕ1, x3 ϕ1^2, x3 ϕ1^3, x3 ϕ1^4}, {σ^2, σ^2 (1 + ϕ1^2), σ^2 (1 + ϕ1^2 + ϕ1^4), σ^2 (1 + ϕ1^2 + ϕ1^4 + ϕ1^6)}}
```

We see that for a stationary time series the predicted value converges to the mean of the series (=0) and the mean square error of the forecast converges to the variance of the series ($=\sigma^2$) for large h. Also when the time

series is an AR process, Z_{n+h} for $h \le 0$ does not appear in (7.2), so the assumption of infinite sample does not come in and no approximation is made. So even if we set `Exact -> False`, (7.2) to (7.5) and (7.8) give the exact finite sample best linear predictor and its mean square error. However, when the MA part is present, the approximation can make a difference as in the following example.

Example 7.3 Given the first 15 data points of a time series generated from the MA(2) process $X_t = Z_t + 0.5\, Z_{t-1} - 1.2\, Z_{t-2}$ ($\sigma^2 = 1$), find the predictions for the next four data points and their mean square errors.

We first define the model as `ma2` to avoid repeated typing.

```
In[7]:= ma2 = MAModel[{0.5, -1.2}, 1];
```

The random number generator is seeded first.

```
In[8]:= SeedRandom[1093]
```

This generates a time series of length 15.

```
In[9]:= madata = TimeSeries[ma2, 15];
```

These are the exact best linear predictions and their mean square errors.

```
In[10]:= BestLinearPredictor[madata, ma2, 4]

Out[10]= {{-0.0558164, 0.548794, 0., 0.}, {1.89171, 1.92927, 2.69, 2.69}}
```

However, if we try to perform the same calculation approximately, we obtain different results.

This gives the approximate best linear predictions.

```
In[11]:= BestLinearPredictor[madata, ma2, 4, Exact -> False]

Out[11]= {{123.22, -107.172, 0., 0.}, {1, 1.25, 2.69, 2.69}}
```

The reason that we get totally different forecasts is that the model is not invertible.

The model is not invertible.

```
In[12]:= InvertibleQ[ma2]

Out[12]= False
```

So the approximation (7.9) is not valid.

On the other hand, for an invertible model and a large data set, the approximation can be very good and it is often used to speed up calculations. The following example is for an invertible MA(2) model with 100 data points.

Example 7.4 Calculate the prediction for the next four data points given the first 100 data points generated from the model $X_t = Z_t - 0.5\,Z_{t-1} + 0.9\,Z_{t-2}$. The noise variance is 1.

We define the model to be `model`.

```
In[13]:= model = MAModel[{-0.5, 0.9}, 1];
```

The random number generator is seeded first.

```
In[14]:= SeedRandom[1093]
```

This generates the time series.

```
In[15]:= data = TimeSeries[model, 100];
```

The exact best linear prediction is obtained.

```
In[16]:= BestLinearPredictor[data, model, 4]

Out[16]= {{1.82978, -0.855409, 0., 0.}, {1., 1.25, 2.06, 2.06}}
```

This yields the approximate best linear prediction.

```
In[17]:= BestLinearPredictor[data, model, 4, Exact -> False]

Out[17]= {{1.83237, -0.851662, 0., 0.}, {1, 1.25, 2.06, 2.06}}
```

Again $\hat{X}_n(h) = 0$ for $h > q$. (See the remark after Example 7.1.) Also for an MA model $\psi_j = \theta_j$ ($\theta_0 \equiv 1$) the mean square error given by (7.8) is $\sigma^2 \sum_{i=0}^{q} \theta_i^2$ for $h > q$.

A natural question is how large n has to be in order to get a very good approximation to the best linear prediction. It depends on how close to the unit circle the zeros of the MA polynomial are. The closer the zeros are to the unit circle, the slower the π weights decrease and the larger the n required to ensure the validity of (7.9). We can explicitly find the absolute values of the roots of $\theta(x) = 0$ in the above example.

Here are the absolute values of the roots.

```
In[18]:= Abs[x /. Solve[1 - 0.5 x + 0.9 x^2 == 0, x]]

Out[18]= {1.05409, 1.05409}
```

It appears that $n = 100$ gives an adequate approximation.

Example 7.5 For the invertible ARMA(1, 2) process $X_t - 0.9\,X_{t-1} = Z_t - 0.4\,Z_{t-1} + 0.8\,Z_{t-2}$, the two roots of $\theta(x) = 0$ have an absolute value around 1.12. We expect that the approximation will be good even for moderate n. This is indeed the case as we see below.

This defines the model.

```
In[19]:= model = ARMAModel[{0.9}, {-0.4, 0.8}, 1];
```

The random number generator is seeded.

```
In[20]:= SeedRandom[9387]
```

This generates a time series of length 50.

```
In[21]:= data = TimeSeries[model, 50];
```

This yields the exact best linear prediction.

```
In[22]:= BestLinearPredictor[data, model, 4]

Out[22]= {{3.79618, 3.76456, 3.3881, 3.04929}, {1., 1.25001, 2.8125, 4.07813}}
```

Here is the approximate best linear prediction.

```
In[23]:= BestLinearPredictor[data, model, 4, Exact -> False]

Out[23]= {{3.79627, 3.76543, 3.38889, 3.05}, {1, 1.25, 2.8125, 4.07813}}
```

We see that in this case for $n = 50$ the approximate prediction is in good agreement with the exact best linear prediction.

1.7.3 Updating the Forecast

Another advantage of using the approximate forecast formula is that the forecast can be easily updated when new observations are available. From (7.7) the forecast error for the prediction $\hat{X}_n(h)$ is given by $e_n(h) = X_{n+h} - \hat{X}_n(h) = \sum_{j=0}^{h-1} \psi_j Z_{n+h-j}$. And the difference $e_n(h+1) - e_{n+1}(h)$ is $\hat{X}_{n+1}(h) - \hat{X}_n(h+1) = \psi_h Z_{n+1}$. Setting $h = 1$, we get $Z_{n+1} = e_n(1) = X_{n+1} - \hat{X}_n(1)$. So we have

$$\hat{X}_{n+1}(h) = \hat{X}_n(h+1) + \psi_h\big(X_{n+1} - \hat{X}_n(1)\big). \tag{7.10}$$

This is the desired formula for updating the forecast. That is, once X_{n+1} is known, we know the actual forecast error $e_n(1)$, and the forecast value for X_{n+h+1} is modified by a quantity proportional to $e_n(1)$. Equation (7.10) allows us to get the forecasts based on $n+1$ observations from the forecasts based on n observations. Note that the updating formula is for the approximate best linear predictor only.

As an exercise, let us try to write a program implementing (7.10). We want to find the forecasts $\{\hat{X}_{n+1}(1), \ldots, \hat{X}_{n+1}(h)\}$ of the time series given the model, the old forecast values $\{\hat{X}_n(1), \ldots, \hat{X}_n(h+1)\}$, and the new available value X_{n+1}. The ψ weights can be obtained by extracting the first argument of `ToMAModel`. A simple function that performs the updating can be written as follows.

This defines a function to do the updating.

```
In[24]:= newforecast[oldcast_?VectorQ, model_, newvalue_] :=
           Rest[oldcast] + (newvalue - oldcast[[1]]) ToMAModel[model, Length[oldcast] - 1][[1]]
```

Example 7.6 Here is an example using the function `newforecast` to update the forecast of the ARMA(1, 1) model $X_t = 0.5\, X_t - 1 + Z_t - 0.7\, Z_{t-1}$.

We first define the ARMA(1, 1) model to be `model`.

```
In[25]:= model = ARMAModel[{0.5}, {-0.7}, 1];
```

This seeds the random number generator.

```
In[26]:= SeedRandom[94 387]
```

This generates the time series data.

```
In[27]:= data = TimeSeries[model, 50];
```

Suppose that only the first 40 data points are available and we predict the next four points.

Here is the prediction for the next four points based on 40 data points.

```
In[28]:= BestLinearPredictor[Take[data, 40], model, 4, Exact -> False]

Out[28]= {{0.0655937, 0.0327968, 0.0163984, 0.00819921}, {1, 1.04, 1.05, 1.0525}}
```

When the 41st data point is given, the prediction based on the 41 data points can be obtained from the old prediction using `newforecast`.

Here is the prediction for X_{42}, X_{43}, and X_{44}.

```
In[29]:= newforecast[%[[1]], model, data[[41]]]

Out[29]= {0.0035147, 0.00175735, 0.000878675}
```

Note that the new forecast has a different lead time as well as forecast origin. Note also that the mean square forecast errors are not updated. They are given by (7.8) and remain the same independent of the forecast origin since they are the mean square forecast errors in the infinite sample case.

These updated forecast values are indeed the same as those had we done the forecast with 41 data points in the first place.

This gives the prediction for the next three points based on 41 data points.

```
In[30]:= BestLinearPredictor[Take[data, 41], model, 3, Exact -> False]

Out[30]= {{0.0035147, 0.00175735, 0.000878675}, {1, 1.04, 1.05}}
```

Example 7.7 Suppose more than one new observation is made available, we can recursively update the forecast using the *Mathematica* internal function `Fold` and `newforecast` defined above. Suppose initially we make a forecast of a lead time 10 based on $n = 40$ data points and later 5 new data points are obtained. We can update our forecast based on the new available information as follows. (The same data and model as in Example 7.6 are used to illustrate this.)

We denote the first 40 data points `data0`.

```
In[31]:= data0 = Take[data, 40];
```

We denote the next 5 data points `newdata`.

```
In[32]:= newdata = Take[data, {41, 45}];
```

This is the prediction.

```
In[33]:= BestLinearPredictor[data0, model, 10, Exact -> False]

Out[33]= {{0.0655937, 0.0327968, 0.0163984, 0.00819921, 0.00409961,
           0.0020498, 0.0010249, 0.000512451, 0.000256225, 0.000128113},
          {1, 1.04, 1.05, 1.0525, 1.05313, 1.05328, 1.05332, 1.05333, 1.05333, 1.05333}}
```

This updates the above prediction.

```
In[34]:= Fold[newforecast[#1, model, #2] &, %[[1]], newdata]

Out[34]= {0.00671743, 0.00335871, 0.00167936, 0.000839678, 0.000419839}
```

We get the same result using the forecast origin $n = 45$.

```
In[35]:= BestLinearPredictor[Take[data, 45], model, 5, Exact -> False]

Out[35]= {{0.00671743, 0.00335871, 0.00167936, 0.000839678, 0.000419839},
          {1, 1.04, 1.05, 1.0525, 1.05313}}
```

1.7.4 Forecasting for ARIMA and Seasonal Models

The above forecast formulas can be generalized to the special nonstationary case of an ARIMA process. An ARIMA(p, d, q) process after differencing d times is an ARMA(p, q) process, and if it is stationary, we can use the formulas outlined above to forecast future values and then transform back ("integrate") to get the forecast for the ARIMA model. For the derivation of the best linear predictor for ARIMA models, see Brockwell and Davis (1987), p. 304. For SARIMA models, we can expand its seasonal and ordinary polynomials and write it as an ordinary ARIMA model, and the prediction methods above can be used similarly.

Example 7.8 Forecast a SARIMA$(1, 2, 0)\,(2, 2, 2)_3$ model
$(1 - B)\left(1 - B^3\right)(1 - 0.5\,B)\left(1 - 0.4\,B^3 + 0.8\,B^6\right)X_t = \left(1 - 0.3\,B + 0.7\,B^2\right)Z_t$.

We define the model.

```
In[36]:= model = SARIMAModel[{1, 1}, 3, {0.5}, {0.4, -0.8}, {-0.3, 0.7}, {}, 1];
```

This seeds the random number generator.

```
In[37]:= SeedRandom[37 561]
```

This generates the time series.

```
In[38]:= TimeSeries[model, 100];
```

We obtain the prediction for the next five values.

```
In[39]:= BestLinearPredictor[%, model, 5]

Out[39]= {{203.722, 207.506, 204.199, 197.286, 202.619}, {1., 2.44, 6.44, 20.88, 39.1984}}
```

Since the model is not stationary, the mean square forecast error in fact diverges as the lead time $h \to \infty$.

There is another point worth noting. So far we have assumed that there is no constant term present in our definition of an ARMA model (see (2.3)), and any constant term can be transformed away simply by subtracting the mean from a stationary process. However, in a nonstationary model like ARIMA or SARIMA model, the constant term cannot be transformed away directly and we have to first difference the process to get a stationary process $Y_t = (1 - B^s)^D (1 - B)^d X_t$. Now $\{Y\}$ is stationary and any constant can be transformed away as before.

Predicting the future values of an ARIMA or SARIMA process with a constant term involves a further complication. We must first difference the data to get $\{y_t = (1 - B^s)^D (1 - B)^d x_t\}$, then subtract the sample mean $\overline{y}$ of $\{y_t\}$ from the differenced series. After getting the predicted $y - \overline{y}$ values, we need to add the sample mean back to get $\hat{y}$. To get the prediction for x, we can use the function

$$\texttt{IntegratedPredictor[}\mathit{data}\texttt{, \{}d\texttt{, }D\texttt{\}, }s\texttt{, }\hat{y}\texttt{]}.$$

Here *data* are the original data and *yhat* is the predicted values for the process $\{Y_t\}$. The prediction errors are given as if the constant were not there. (See Example 9.2 for an illustrative use of `IntegratedPredictor`.)

Example 7.9 Forecast of the ARIMA(0, 1, 1) model $X_t = X_{t-1} + Z_t + \theta_1 Z_{t-1}$.

The approximate best linear predictor gives (see (7.2)) $\hat{X}_n(1) = X_n + \theta_1 Z_n$ and $\hat{X}_n(h) = \hat{X}_n(h-1)$ for $h = 2, 3, \ldots$. So the forecast is a constant independent of h. Z_n is calculated from (7.9) and the forecast can be written as

$$\hat{X}_n(h) = (1 + \theta_1) \sum_{i=0}^{n-1} (-\theta_1)^i X_{n-i} = (1 + \theta_1) X_n - \theta_1 \hat{X}_{n-1}(1). \tag{7.11}$$

That is, the forecast for an ARIMA(0, 1, 1) process is an exponentially weighted average of past observations.

This seeds the random number generator.

```
In[40]:= SeedRandom[305 908]
```

This generates the time series from the ARIMA(0, 1, 1) model.

```
In[41]:= data = TimeSeries[ARIMAModel[1, {0}, {-0.3}, 1], 20];
```

Note that the prediction is a constant independent of the lead time.

```
In[42]:= BestLinearPredictor[data, ARIMAModel[1, {0}, {-0.3}, 1], 3, Exact -> False]

Out[42]= {{2.38559, 2.38559, 2.38559}, {1, 1.49, 1.98}}
```

1.7.5 Exponential Smoothing

There are other ways of forecasting future values. One commonly used forecast method is called *exponential smoothing*. In Example 7.9, we showed that the approximate best linear predictor for an ARIMA(0, 1, 1) process is given by an exponentially weighted average of past observations. The exponential smoothing method of forecasting is given by the same formula, that is, the forecast is the weighted average of past observations with more weight given to the more recent past values and exponentially decreasing weights for earlier values. If we define the *smoothing constant* $a = 1 + \theta_1$, (7.11) becomes

$$\hat{X}_n(h) = a\, X_n + (1 - a)\, \hat{X}_{n-1}(1) = \hat{X}_{n-1}(1) + a\big(X_n - \hat{X}_{n-1}(1)\big). \tag{7.12}$$

This is the forecast formula using exponential smoothing. The function

ExponentialMovingAverage[*data*, *a*]

gives the $\hat{X}_i(1)$ for $i = 1, \ldots, n$ using (7.12) with smoothing constant a and starting value x_1, the first entry in *data*. We see that the approximate best linear predictor for ARIMA(0, 1, 1) model corresponds to exponential smoothing with a smoothing constant $a = 1 + \theta_1$ and $x_0 = 0$.

This is the predicted value for X_{21} using the exponential smoothing method for the data in Example 7.9.

```
In[43]:= ExponentialMovingAverage[Join[{0}, data], 0.7][[-1]]

Out[43]= 2.38559
```

1.7.6 Forecasting for Multivariate Time Series

The above forecast formulas and functions for univariate time series can be generalized directly to the multivariate case. Here the forecast $\hat{X}_n(h)$ is a vector, and the mean square errors are now error covariance matrices and (7.8) becomes $\sum_{j=0}^{h-1} \Psi_j \Sigma \Psi_j'$. The function `newforecast` we wrote earlier for updating the forecast should be modified to incorporate the matrix nature of the Σ and Ψ weights.

This defines `newforecast` for the multivariate case.

```
In[44]:= newforecast[oldcast_?MatrixQ, model_, newvalue_] :=
          ToMAModel[model, Length[oldcast] - 1][[1]].(newvalue - oldcast[[1]]) + Rest[oldcast]
```

Example 7.10 Forecast a bivariate ARMA(1, 1) model with forecast origin 30 and lead time 3.

This defines the model.

```
In[45]:= model = ARMAModel[{{{0.3, -0.1}, {0.8, -0.5}}},
           {{{0.2, -0.5}, {0.2, 0.9}}}, {{1, 0}, {0, 1}}];
```

The random number generator is seeded.

```
In[46]:= SeedRandom[34 069]
```

This gives the prediction for the next three values of the series and the error covariance matrices.

```
In[47]:= BestLinearPredictor[TimeSeries[model, 30], model, 3]
```

$$\text{Out[47]=}\ \left\{\{\{0.360291, -0.120058\}, \{0.120093, 0.348262\}, \{0.00120175, -0.0780564\}\},\ \left\{\left\{\left\{1., -3.64316 \times 10^{-10}\right\}, \left\{-3.64316 \times 10^{-10}, 1.\right\}\right\},\ \{\{1.61, 0.26\}, \{0.26, 2.16\}\}, \{\{1.6609, 0.4046\}, \{0.4046, 2.6324\}\}\right\}\right\}$$

1.8 Spectral Analysis

We have so far studied stationary time series in terms of quantities that are functions of time. For example, the covariance function and correlation function are functions of the time lag. This approach is termed time series analysis in the *time domain*. Another approach is to analyze the time series in Fourier space or in the *frequency domain*. Although theoretically it provides a different representation of the same information, this approach can yield both powerful numerical methods of analysis and new insights. The techniques used in the frequency domain fall under the general rubric of spectral analysis and the fundamental tool is the Fourier transform. In this section we study time series in the frequency domain. First we introduce the concept of power spectrum and illustrate how to obtain the spectrum of a given ARMA process. Then we discuss how to get the estimated spectrum from time series data. Smoothing of spectra in both the time and frequency domains using various windows is also demonstrated.

1.8.1 Power Spectral Density Function

In the time domain we have investigated the covariance or correlation function of a stationary time series. Alternatively, we can study the *(power) spectral density function* or simply the *(power) spectrum* as a function of the frequency ω. The spectrum of a stationary time series $f(\omega)$ is the counterpart of a covariance function in frequency domain. That is, it is the Fourier transform of the covariance function $\gamma(k)$ and vice versa:

$$f(\omega) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} \gamma(k)\, e^{-ik\omega} \tag{8.1}$$

and

$$\gamma(k) = \int_{-\pi}^{\pi} f(\omega)\, e^{ik\omega}\, d\omega. \tag{8.2}$$

Here the covariance function $\gamma(k)$ is assumed to satisfy $\sum_{-\infty}^{\infty} |\gamma(k)| < \infty$ (*i.e.*, $\gamma(k)$ is *absolutely summable*). Since $\gamma(k) = \gamma(-k)$, (8.1) can also be written as

$$f(\omega) = \frac{1}{2\pi}\left(\gamma(0) + 2\sum_{k=1}^{\infty} \gamma(k)\cos(k\omega)\right).$$

We can immediately identify the following properties of the spectrum $f(\omega)$: (a) $f(\omega)$ is 2π-periodic, that is, $f(\omega) = f(\omega + 2\pi j)$ (j integer), and (b) $f(\omega)$ is real and even ($f(\omega) = f(-\omega)$). These properties of $f(\omega)$ make it sufficient to consider the spectrum in the range $0 \le \omega \le \pi$.

Setting $k = 0$ in (8.2), we have $\sigma^2 = \int_{-\pi}^{\pi} f(\omega)\, d\omega$. We see that the total variance of the process can be "decomposed" into contributions from different frequencies, and $f(\omega)\, d\omega$ represents the contribution to the total variance of the components in the frequency range $(\omega, \omega + d\omega)$.

Again we caution the reader that conventions differ. Some authors define the spectrum with different factors. Others use the correlation function instead of covariance function in (8.1) to define the spectrum. We call this the normalized spectrum and use (8.1) as our definition of the spectrum.

Example 8.1 Use (8.1) to find the spectrum of a white noise.

The covariance function of a white noise process is given by $\gamma(0) = \sigma^2$, $\gamma(k) = 0$ for $k \neq 0$. From (8.1) its spectrum is $f(\omega) = \sigma^2 / (2\pi)$. It is independent of ω and its plot against frequency is "flat". This means that each frequency contributes equally to the variance and thus the name white noise.

1.8.2 The Spectra of Linear Filters and of ARMA Models

In this section we will derive the spectrum of a filtered process and of an ARMA process from the definitions given in Section 1.8.1. We will assume that the appropriate transformations have been made to render the process under consideration zero-mean and stationary.

Spectrum of a Linear Filtered Process

We have introduced the concept of a time-invariant linear filter in Section 1.4. Now we want to look at the effect of filtering on the spectrum of a process. Let $\{Y_t\}$ be a stationary, zero-mean process with spectrum $f_Y(\omega)$ and $X_t = \sum_{j=-\infty}^{\infty} \psi_j Y_{t-j}$. The (constant) filter weights $\{\psi_j\}$ satisfy $\sum_{j=-\infty}^{\infty} |\psi_j| < \infty$. Using the definitions of covariance function and (8.1), it is straightforward to show that the spectrum of the filtered process $\{X_t\}$ is given by

$$f_X(\omega) = T(\omega) f_Y(\omega) = \left|\psi\left(e^{-iw}\right)\right|^2 f_Y(\omega). \tag{8.3}$$

So if $\{X_t\}$ is the process obtained from application of a linear filter of weights $\{\psi_j\}$ on $\{Y_t\}$, the spectrum of $\{X_t\}$ is simply the spectrum of $\{Y_t\}$ multiplied by the weight function $\left|\psi\left(e^{-i\omega}\right)\right|^2$, where $\psi\left(e^{-i\omega}\right) = \sum_{j=-\infty}^{\infty} \psi_j e^{-i\omega j}$ is called the *transfer function* of the filter and the squared modulus $\left|\psi\left(e^{-i\omega}\right)\right|^2 = T(\omega)$ the *power transfer function* of the filter. Note $\psi\left(e^{-i\omega}\right)$ is also called the frequency response function, and some authors refer to $\left|\psi\left(e^{-i\omega}\right)\right|^2$ as the transfer function.

As an exercise let us write a little program to calculate the power transfer function $T(\omega)$ of a given linear filter. We assume that only a finite number of filter weights are nonzero (*i.e.*, $\psi_j = 0$ for $|j|$ larger than a positive number M). With this assumption, as we have pointed out before, we can always treat a linear filter as a causal (or one-sided) one (*i.e.*, $\psi_j = 0$ for $j < 0$) since doing so only amounts to a "time shifting" in time domain or an extra phase factor in the transfer function, and it leaves the power transfer function invariant. Therefore, the filter weights can be input in the form *weight* = $\{\psi_0, \psi_1, \psi_2, \ldots, \psi_M\}$.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

The transfer function for a list of weights `weight` can be defined as

```
Table[E^(-I ω j), {j, 0, Length[weight] - 1}].weight
```

Calling the expression for the transfer function `tf`, the power transfer function can be defined as

```
tf * (tf /. a_Complex -> Conjugate[a])
```

A complex number $a = x + iy$ is represented in *Mathematica* as `Complex[x, y]`. The complex conjugate of `tf` is obtained by replacing each complex number in `tf` by its conjugate.

Using the above two lines, we can define a *Mathematica* function called `powertransferfunction`, which computes $\left|\psi\left(e^{-i\omega}\right)\right|^2$ given the filter weights $\{\psi_j\}$ and the frequency variable ω as arguments.

We define a function to calculate the power transfer function.

```
In[2]:= powertransferfunction[weight_List, ω_] := Module[{tf},
          tf = Table[E^(-I ω j), {j, 0, Length[weight] - 1}].weight;
          Simplify[ExpToTrig[Expand[tf (tf /. a_Complex -> Conjugate[a])]]]]
```

Example 8.2 Find the power transfer function of a simple moving average.

A simple moving average of order M is a linear filter of weights $\psi_j = 1/M$ for $j = 0, 1, \ldots, M-1$ and $\psi_j = 0$ otherwise. We use the function defined above to find the power transfer function of the simple moving average filter of order 5.

This is the power transfer function of a moving average of order 5.

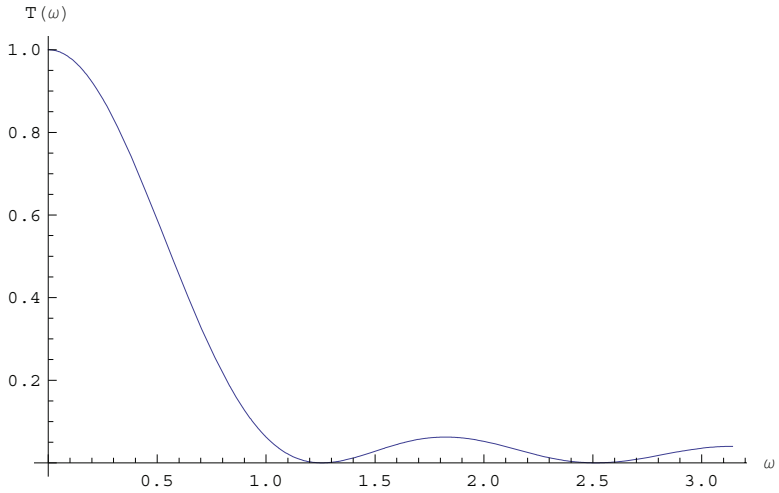
```
In[3]:= powertransferfunction[Table[1 / 5, {5}], ω]
```

$$\text{Out[3]} = \frac{1}{25}\left(1 + 2\cos[\omega] + 2\cos[2\omega]\right)^2$$

We plot the above power transfer function.

```
In[4]:= Plot[%, {ω, 0, Pi}, PlotRange -> All, AxesLabel -> {"ω", "T(ω)"}]
```

Out[4]=



The power transfer function of the filter has a maximum at $\omega = 0$ and is rather small at "high" frequencies. Since the spectrum of the filtered process is the original spectrum multiplied by this function, we see that the effect of the moving average is to suppress the high frequency (short time) part of the spectrum in the original process and retain the low frequency (long time) spectrum. The kind of filter is called a *low-pass filter*. This is why we can use the moving average to eliminate short time fluctuations and, thus, estimate long-term trends.

Spectrum of an MA(q) Process

An MA(q) process can be thought of as a white noise process $\{Z_t\}$ passed through a linear filter of weights $\{1, \theta_1, \theta_2, \ldots, \theta_q\}$. From (8.3) an MA($q$) process has the spectrum

$$f_X(\omega) = \left|\theta\left(e^{-i\omega}\right)\right|^2 f_Z(\omega) = \frac{\sigma^2}{2\pi}\left|\theta\left(e^{-i\omega}\right)\right|^2,$$

where $f_Z(\omega) = \sigma^2/(2\pi)$ is the white noise spectrum, and $\theta(x)$ is the usual MA polynomial $\left(\theta\left(e^{-i\omega}\right) = 1 + \theta_1 e^{-i\omega} + \theta_2 e^{-2i\omega} + \ldots + \theta_q e^{-qi\omega}\right)$. To get the spectrum of an MA process we can use the function `Spectrum`.

`Spectrum[`*model*`, ω]`

gives the spectrum of an ARMA type of model as a function of frequency ω and its code resembles the above demonstration program. (We can also obtain the same MA spectrum by finding the power transfer function and multiplying it by the white noise spectrum.)

Example 8.3 Find the power spectrum of an MA(1) process $X_t = Z_t + \theta_1 Z_{t-1}$.

This gives the spectrum of an MA(1) process.

```
In[5]:= Spectrum[MAModel[{θ1}, σ^2], ω]
```

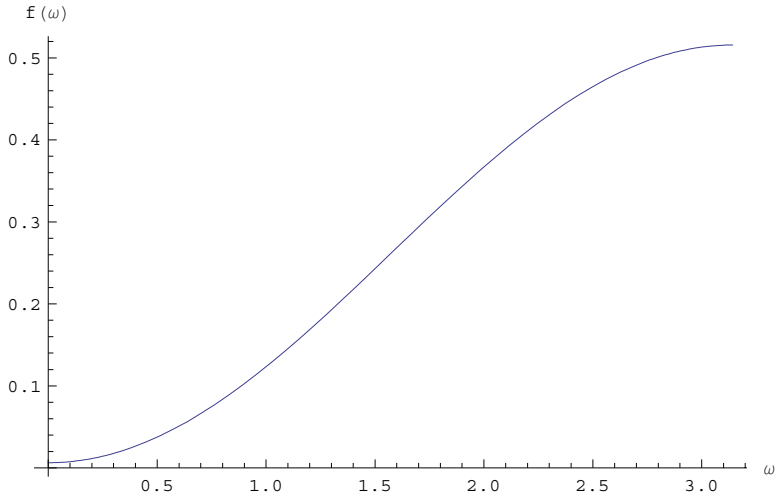
$$\text{Out[5]} = \frac{\sigma^2 \left(1 + 2 \text{Cos}[\omega]\, \theta_1 + \theta_1^2\right)}{2\pi}$$

We see that the spectrum of an MA(1) process is the otherwise flat white noise spectrum multiplied by a weight function (power transfer function), which depends on ω. Since $\cos\omega$ is monotonically decreasing for ω in $[0, \pi]$, as the frequency ω increases $f(\omega)$ decreases for positive θ_1 and increases for negative θ_1. Also since the covariance function at lag 1, $\gamma(1)$, is θ_1, a negative θ_1 can cause the series to fluctuate rapidly and we expect large contributions to the variance at high frequencies. This is indeed the case as we see in the following graph.

Here is the MA(1) spectrum for $\theta_1 = -0.8$ and $\sigma^2 = 1$. `% /. {σ -> 1, θ1 -> -0.8}` is used to replace `s` and `t1` in the above expression with the corresponding values.

```
In[6]:= Plot[% /. {σ -> 1, θ1 -> -0.8}, {ω, 0, π}, AxesLabel -> {"ω", "f(ω)"}]
```

Out[6] =



As an exercise we can plot spectra of different MA models just to develop some intuition. For example, let us find and plot the spectrum for an MA(2) model with $\theta_1 = -0.5$ and $\theta_2 = 0.9$.

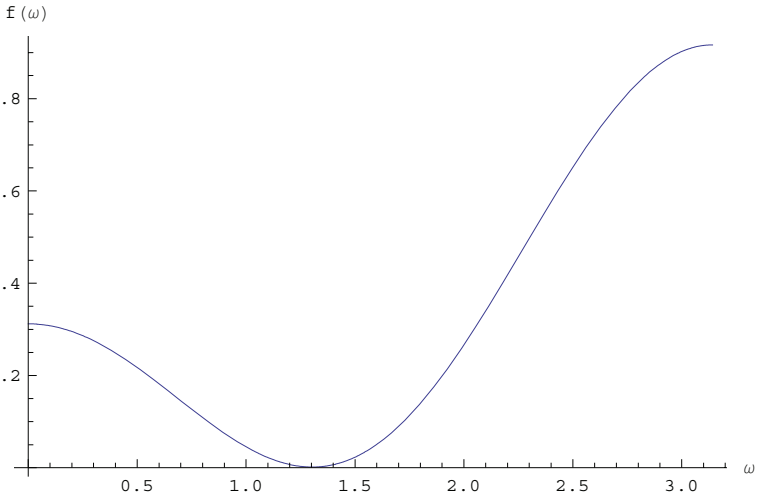
This calculates the spectrum of the MA(2) model.

```
In[7]:= Spectrum[MAModel[{-0.5, 0.9}, 1], ω];
```

Here is the plot of the spectrum. `Evaluate` is used inside the `Plot` to speed up generation of the plot.

```
In[8]:= Plot[Evaluate[%], {ω, 0, π}, AxesLabel -> {"ω", "f(ω)"}]
```

Out[8]=



Spectra of AR(p) and ARMA(p, q) Processes

Similarly, using (8.3) and writing $Z_t = \phi(B)\, X_t$, the spectrum of an AR(p) process is simply given by $f_Z(\omega) = \left|\phi\left(e^{-i\,\omega}\right)\right|^2 f_X(\omega)$ or

$$f_X(\omega) = \frac{\sigma^2}{2\,\pi\left|\phi\left(e^{-i\,\omega}\right)\right|^2},$$

where $\phi(x)$ is the usual AR polynomial. $\left(1 \Big/ \left|\phi\left(e^{-i\,\omega}\right)\right|^2\right.$ is the power transfer function of the linear filter of weights $\{\psi_j\}$ where $X_t = \sum_{j=-\infty}^{\infty} \psi_j\, Z_{t-j}$.)

Example 8.4 Find the spectrum of an AR(1) process $X_t - \phi_1\, X_{t-1} = Z_t$.

This is the spectrum of an AR(1) model.

```
In[9]:= Spectrum[ARModel[{ϕ1}, σ^2], ω]
```

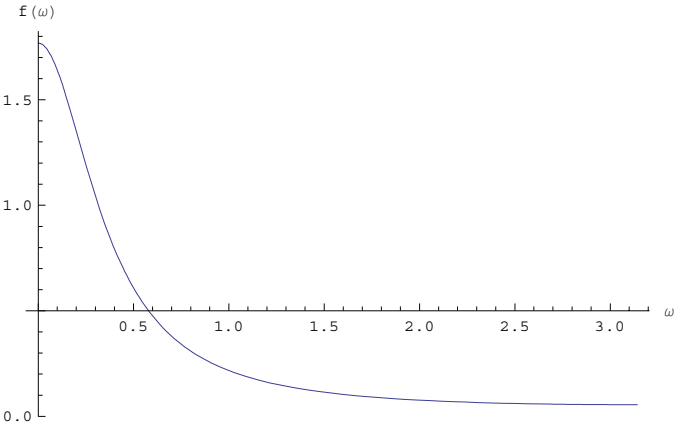
$$\text{Out[9]} = \frac{\sigma^2}{2\,\pi\left(1 - 2\,\text{Cos}[\omega]\,\phi_1 + \phi_1^2\right)}$$

Again the power spectrum is monotonic in ω and, depending on the sign of ϕ_1, the power is concentrated at either low or high frequencies. For $\phi_1 = 0.7$ and $\sigma^2 = 1$ we plot the spectrum.

Here is the spectrum of the AR(1) model with $\phi_1 = 0.7$.

```
In[10]:= Plot[% /. {σ -> 1, ϕ1 -> 0.7}, {ω, 0, π},
          AxesLabel -> {"ω", "f(ω)"}, PlotRange -> All]
```

Out[10]=



In general the spectrum of an AR(p) process is not monotonic and depending on the signs of the AR coefficients the spectrum can assume different shapes.

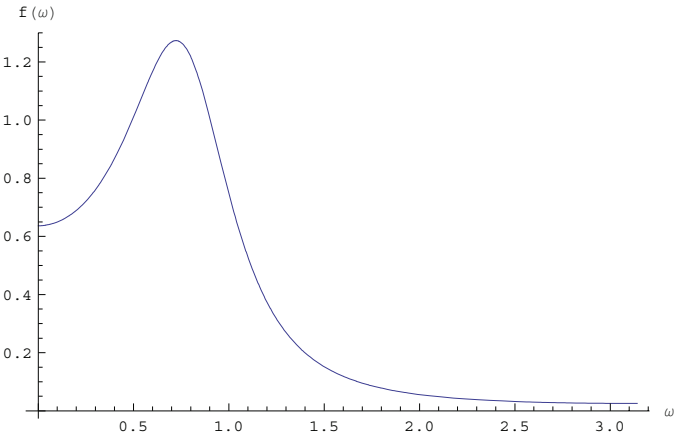
We calculate the spectrum of an AR(2) process with $\phi_1 = 1$ and $\phi_2 = -0.5$.

```
In[11]:= Spectrum[ARModel[{1, -0.5}, 1], ω];
```

Here is the plot of the above spectrum.

```
In[12]:= g1 = Plot[%, {ω, 0, π}, AxesLabel -> {"ω", "f(ω)"}]
```

Out[12]=



The spectrum of an ARMA process $f_X(\omega)$ can also be derived using (8.3). Let $Y_t = \phi(B)\, X_t = \theta(B)\, Z_t$, then we have $f_Y(\omega) = \left|\phi\left(e^{-i\,\omega}\right)\right|^2 f_X(\omega) = \left|\theta\left(e^{-i\,\omega}\right)\right|^2 f_Z(\omega)$ where $f_Z(\omega) = \sigma^2 \big/ 2\,\pi$ is the white noise spectrum. Therefore,

$$f_X(\omega) = \frac{\sigma^2}{2\,\pi}\,\frac{\left|\theta\left(e^{-i\,\omega}\right)\right|^2}{\left|\phi\left(e^{-i\,\omega}\right)\right|^{2.}} \tag{8.4}$$

This is often called a *rational spectrum*. AR and MA spectra are special cases of this spectrum when $\theta(x) = 1$ and $\phi(x) = 1$, respectively.

Example 8.5 Calculate the spectrum of an ARMA(1, 1) process.

This gives the spectrum of an ARMA(1, 1) process.

```
In[13]:= Spectrum[ARMAModel[{ϕ1}, {Θ1}, 1], ω]
```

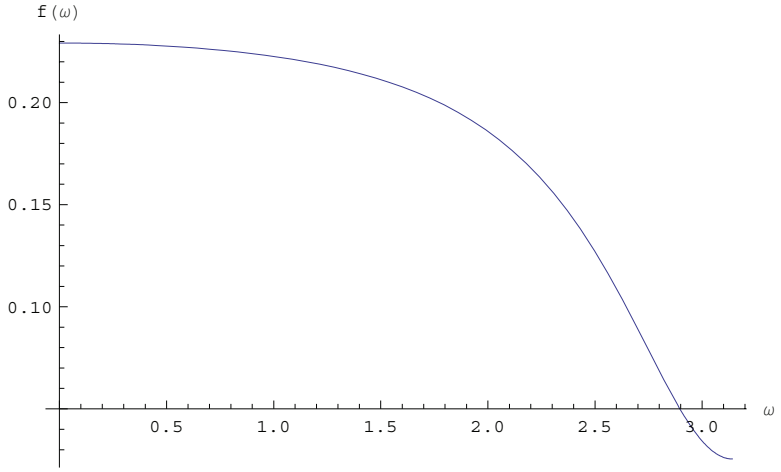
$$\text{Out[13]} = \frac{1 + 2\,\text{Cos}[\omega]\,\Theta_1 + \Theta_1^2}{2\,\pi\,\left(1 - 2\,\text{Cos}[\omega]\,\phi_1 + \phi_1^2\right)}$$

Note that the noise variance has been set to 1. The values of ϕ_1 and θ_1 determine the shape of the spectrum.

Here is the plot of the spectrum with $\phi_1 = -0.5$ and $\theta_1 = 0.8$.

```
In[14]:= Plot[% /. {ϕ1 -> -0.5, Θ1 -> 0.8}, {ω, 0, π}, AxesLabel -> {"ω", "f(ω)"}]
```

Out[14]=



Example 8.6 Here we give an example of spectrum of a seasonal model.

Consider the model $(1 - \phi_1 B)\left(1 - \Phi_1 B^6\right) X_t = Z_t$. It has the following spectrum.

This is the spectrum of the given seasonal model.

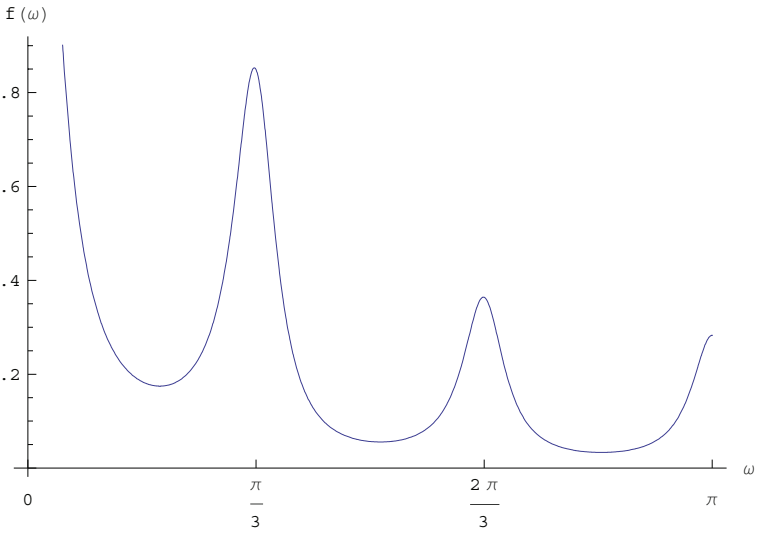
```
In[15]:= Spectrum[SARIMAModel[{0, 0}, 6, {ϕ1}, {Φ1}, {}, {}, σ^2], ω]
```

$$\text{Out[15]} = \frac{\sigma^2}{2\,\pi\,\left(1 - 2\,\text{Cos}[\omega]\,\phi_1 + \phi_1^2\right)\,\left(1 - 2\,\text{Cos}[6\,\omega]\,\Phi_1 + \Phi_1^2\right)}$$

This is the plot of the spectrum.

```
In[16]:= Plot[% /. {σ -> 1, ϕ1 -> 0.5, Φ1 -> 0.5}, {ω, 0, π},
          AxesLabel -> {"ω", "f(ω)"}, Ticks -> {{0, π/3, 2π/3, π}, Automatic}]
```

Out[16]=



We have used the option `Ticks -> {{0, π/3, 2π/3, π}, Automatic}`, which yields tick marks at specified points along the abscissa and puts them automatically along the ordinate.

This multiplicative seasonal process can be thought of as two successive linear filters being applied to a white noise process, and the spectrum is the product of the two corresponding power transfer functions and the white noise spectrum. The seasonal part of the spectrum has an ω dependence through $\cos 6\omega$. Since $\cos 6\omega = 1$ for $\omega = 0, \pi/3, 2\pi/3$, and π, the seasonal part of the spectrum reaches its maximum value at these frequencies for $\Phi_1 > 0$. This spectrum is again weighted by the monotonically decreasing ordinary AR spectrum resulting in the final shape of the spectrum we see above.

1.8.3 Estimation of the Spectrum

The spectrum of a time series can be obtained from previously stated formulation as long as we know the model satisfied by the time series. Often in practice we only have a finite set of time series data and we would like to estimate from it the spectrum of the process.

There are two commonly used approaches to estimating the spectrum. The first is called the ARMA method of spectrum estimation. In this way of estimating the spectrum, we first fit an appropriate ARMA type of model to the data and then get the estimated spectrum by replacing the true model parameters in (8.4) with the estimated ARMA parameters. The function `Spectrum[`*model*`, w]` gives the estimated spectrum as a function of ω with *model* here being the estimated model from the given data set.

Another approach to estimating spectrum is nonparametric, that is, it uses the time series data directly and no model is assumed *a priori*. A "natural" way of getting the estimate of $f(\omega)$, $\hat{f}(\omega)$, is to replace the covariance function $\gamma(k)$ in (8.1) by the sample covariance function $\hat{\gamma}(k)$,

$$\hat{f}(\omega) = \frac{1}{2\pi} \sum_{k=-(n-1)}^{n-1} \hat{\gamma}(k)\, e^{-i k \omega}. \tag{8.5}$$

Here ω takes on continuous values in the range $[-\pi, \pi]$, and we call $\hat{f}(\omega)$ the *continuous sample spectrum*. Note that the sum in (8.5) is restricted to $|k| < n$, since for a time series of length n the sample covariance function can be calculated up to at most a lag of $n - 1$. It is straightforward to write a one-line program that gives the continuous sample spectrum given the sample covariance function `cov` $= \{\hat{\gamma}(0), \hat{\gamma}(1), \ldots, \hat{\gamma}(n-1)\}$ and the frequency variable ω. For example, write (8.5) as $\hat{f}(\omega) = \hat{\gamma}(0)/(2\pi) + \sum_{k=1}^{n-1} \hat{\gamma}(k) \cos(k\omega)/\pi$. The continuous sample spectrum can be obtained using

```
cov[[1]]/(2Pi) + Rest[cov].Table[Cos[k ω], {k, 1, Length[cov]-1}]/Pi.
```

In practice, it is often convenient to only consider $\hat{f}(\omega)$ defined on a set of discrete *Fourier frequencies* $\omega_j = 2\pi j/n$ for $j = 0, 1, \ldots, n-1$. Let $\omega = \omega_j$ in (8.5) and we call $\hat{f}(\omega_j)$ for $j = 0, 1, \ldots, n-1$ the *discrete sample spectrum*. It turns out that computationally it is very convenient to calculate the discrete sample spectrum, since using the definition of the sample covariance function, (8.5) can be written at $\omega = \omega_j$ as

$$\hat{f}(\omega_j) = \frac{1}{2\pi n} \left| \sum_{t=1}^{n} x_t\, e^{i \omega_j t} \right|^2 = \frac{1}{2\pi} \left| x(\omega_j) \right|^2, \tag{8.6}$$

where $x(\omega_j) = \frac{1}{\sqrt{n}} \sum_{t=1}^{n} x_t\, e^{i \omega_j t}$ is the discrete Fourier transform of x_t.

So the discrete sample spectrum is simply proportional to the squared modulus of the Fourier transform of the data. In the rest of the discussion we will drop the adjective "discrete" and use sample spectrum to mean $\hat{f}(\omega_j)$. *Periodogram* is another name for essentially the same quantity. To get the sample spectrum of a given data $\{x_1, x_2, \ldots, x_n\}$ use

```
Spectrum[data].
```

It gives $\hat{f}(\omega_j)$ for $j = 0, 1, \ldots, n-1$. Internally, it simply uses the function `Fourier`. Although the definition of the Fourier transform here differs from that of `Fourier` by a phase factor, this phase factor will not enter in (8.6) due to the squared modulus. You can also define a function to estimate the normalized spectrum defined by (8.5) with $\hat{\rho}(k)$ in place of $\hat{\gamma}(k)$ as follows.

This defines a function that gives the estimated normalized spectrum.

```
In[17]:= normalizedspectrum[data_] := Spectrum[(data - Mean[data]) / StandardDeviation[data]];

In[18]:= normalizedspectrum[data1_, data2_] :=
          Spectrum[(data1 - Mean[data1]) / StandardDeviation[data1],
           (data2 - Mean[data2]) / StandardDeviation[data2]]
```

Example 8.7 We have calculated the spectrum of the AR(2) model $X_t - X_{t-1} + 0.5\, X_{t-2} = Z_t$ in Example 8.4. We now calculate the sample spectrum from the data generated from this model.

This seeds the random number generator.

```
In[19]:= SeedRandom[294 857]
```

The time series of length 150 is generated from the given AR(2) model.

```
In[20]:= data = TimeSeries[ARModel[{1.0, -0.5}, 1], 150];
```

This gives the sample spectrum.

```
In[21]:= spec = Spectrum[data];
```

If we just want to get some idea of what the sample spectrum looks like, we can simply do `ListLinePlot[spec]`. However, as we have mentioned before, `ListLinePlot[spec]` plots the sample spectrum $\{\hat{f}(\omega_0), \hat{f}(\omega_1), \ldots, \hat{f}(\omega_{n-1})\}$ against $\omega = 1, 2, \ldots, n$ not $\omega = \omega_j$ for $j = 0, 1, \ldots, n-1$. A careful plot of the sample spectrum should use `ListLinePlot` to plot points $\left\{\left\{\omega_0, \hat{f}(\omega_0)\right\}, \left\{\omega_1, \hat{f}(\omega_1)\right\}, \ldots\right\}$. To avoid repeated typing we can define a function `plotspectrum` to plot sample spectrum.

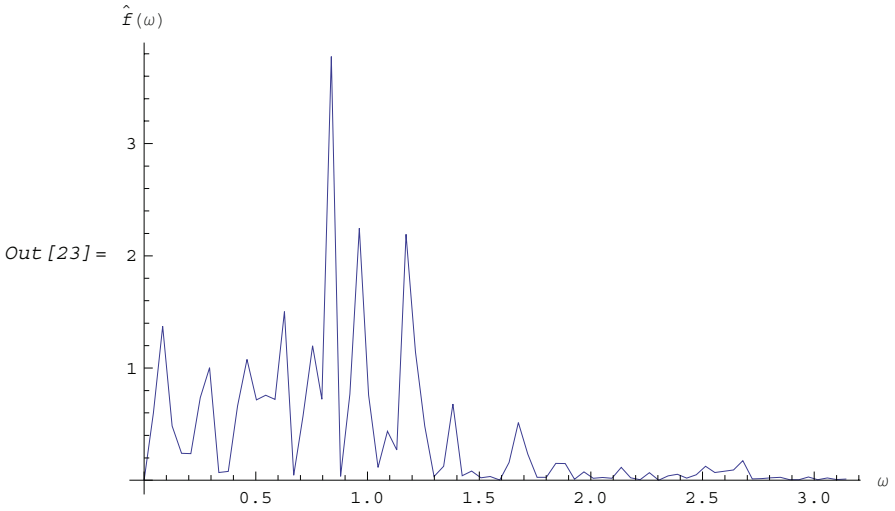
This defines the function `plotspectrum`.

```
In[22]:= plotspectrum[spec_List, opts___] :=
          Module[{w = Table[i, {i, 0, Pi, 2 Pi / Length[spec]}]},
           ListLinePlot[Transpose[{w, Take[spec, Length[w]]}], opts]]
```

Note that we only need the spectrum in the range $[0, \pi]$ and the spectrum in $(\pi, 2\pi)$ has been dropped.

Here is the plot of the sample spectrum.

```
In[23]:= plotspectrum[spec, PlotRange -> All, AxesLabel -> {"ω", "f̂(ω)"}]
```



1.8.4 Smoothing the Spectrum

In general, the sample spectrum can fluctuate a lot and its variance can be large, as can be seen in the last example. In fact, the variance of the sample spectrum does not go to zero as the length of the time series $n \to \infty$. In other words, $\hat{f}(\omega)$ is not a consistent estimator of $f(\omega)$. In order to reduce the fluctuations in the sample spectrum, we often "smooth" the sample spectrum using weighted averages. There are two commonly used approaches to spectrum smoothing; they correspond to performing a weighted average in the frequency domain and in the time domain, respectively. In the following, we will show how to smooth a spectrum using both of the approaches and discuss the relationship between the two methods.

Smoothing in the Frequency Domain

Let $\{W_n(k)\}$ $(k = -M, -(M-1), \ldots, (M-1), M)$ be a set of weights satisfying

$$W_n(k) = W_n(-k), \text{ and } W_n(k) = 0 \text{ for } |k| > M,$$

$$\sum_{|k| \le M} W_n(k) = 1.$$

In the following we will omit the subscript n in $W_n(k)$, *i.e.*, the n dependence of the weights is understood. Given a discrete sample spectrum $\hat{f}(\omega_j)$ we define its smoothed spectrum by

$$\hat{f}_S(\omega_j) = \sum_{k=-M}^{M} W(k)\hat{f}(\omega_{j-k}). \tag{8.7}$$

That is, the smoothed spectrum at frequency $w_j = 2\pi j/n$ is the weighted average of the spectrum in the neighborhood $[\omega_{j-M}, \omega_{j+M}]$ of ω_j. Since only frequencies in this range are "seen" in the averaging process, the set of weights $\{W(k)\}$ is referred to as a *spectral window*. (The subscript S in $\hat{f}_S(\omega)$ stands for spectral window.) The function

`SmoothedSpectrumS[`*spectrum*`,` *window*`]`

smooths the given sample spectrum using the supplied spectral window. This function is very much like a weighted moving average since $\hat{f}_S$ is obtained by an application of the filter $\{W(k)\}$ on the sample spectrum. However, it differs in two respects: (a) the periodic nature of the sample spectrum (*i.e.*, $\hat{f}(\omega) = \hat{f}(\omega + 2\pi j)$) is taken into account in implementing (8.7), so the output of `SmoothedSpectrumS` (smoothed spectrum using spectral window) has the same length as the input spectrum; and (b) since $W(k) = W(-k)$, we only need to input the spectral window as $\{W(0), W(1), \ldots, W(M)\}$.

Designing a window includes choosing an M and the appropriate weights. See Priestley (1981), Section 7.5 for details on how to choose a window. An often used spectral window is the Daniell window defined by $W(k) = 1/(2M+1)$ for $|k| \le M$ and 0 otherwise (*i.e.*, it is a rectangular window). Using the Daniell window to smooth the sample spectrum is the same as doing a simple moving average.

Example 8.8 Here we use the Daniell window ($M = 6$) to smooth the sample spectrum in Example 8.7.

When $M = 6$ the Daniell window is $W(k) = 1/(2M+1) = 1/13$. We can generate the list of $M + 1 = 7$ identical weights using `Table[1/13, {7}]`.

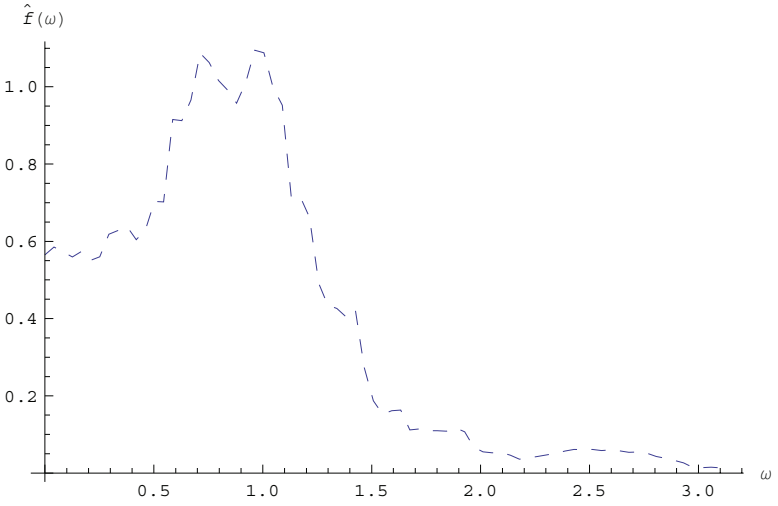
This gives the smoothed spectrum.

```
In[24]:= specs = SmoothedSpectrumS[spec, Table[1 / 13, {7}]];
```

Here is the plot of the smoothed spectrum.

```
In[25]:= plotspectrum[specs, PlotStyle -> Dashing[{0.02}], AxesLabel -> {"ω", "f̂(ω)"}]
```

Out[25]=

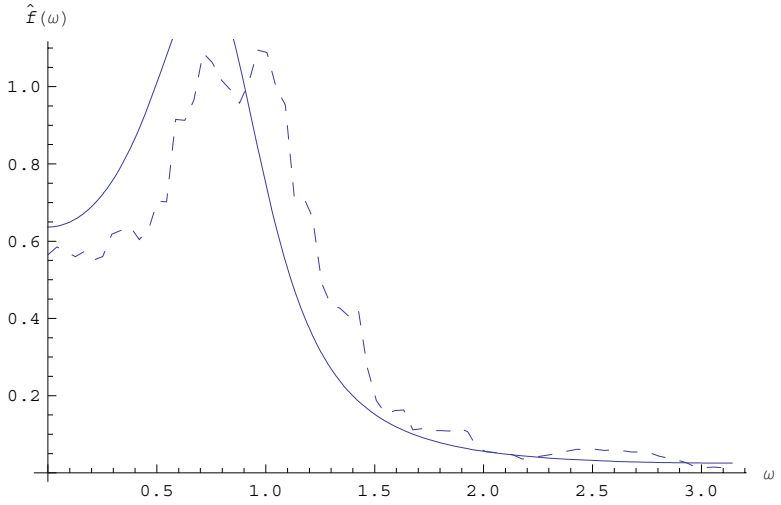


This spectrum closely resembles the theoretical spectrum calculated earlier for the same model (see `g1` in Example 8.4). We display them together to see the resemblance using `Show`.

The smoothed spectrum and the theoretical spectrum are displayed together.

```
In[26]:= Show[%, g1]
```

Out[26]=



Changing windows or using different M for the Daniell window can greatly affect the shape of the smoothed spectrum. The reader is encouraged to try different windows and see their effect on smoothing.

Smoothing in the Time Domain

Another approach to smoothing a spectrum is to perform averaging in the time domain. Instead of averaging the sample spectrum, we assign weights to the sample covariance function in (8.5) such that the contributions from covariance at large lags, which are generally not reliable, will be small or zero. Let $\{\lambda(k)\}$ $(k = -M, -(M-1), \ldots, M-1, M)$ be a set of weights satisfying

$$\lambda(-k) = \lambda(k), \text{ and } \lambda(k) = 0 \text{ for } |k| > M,$$

$$\lambda(0) = 1 \text{ and } |\lambda(k)| \leq 1;$$

then $\{\lambda(k)\}$ constitutes a *lag window* and M is called the *truncation point*. A smoothed spectrum using a lag window is defined by

$$\hat{f}_L(\omega) = \frac{1}{2\pi} \sum_{k=-M}^{M} \lambda(k)\, \hat{\gamma}(k)\, e^{-ik\omega}. \tag{8.8}$$

The subscript L stands for lag window. Note that (8.8) has defined a continuous spectrum for $\omega = [-\pi, \pi]$, not just for discrete Fourier frequencies. This is not too bad from a computational point of view since the truncation point M is generally small compared to n.

The function

SmoothedSpectrumL[*cov*, *window*, ω]

gives the smoothed spectrum $\hat{f}_L(\omega)$ defined in (8.8). The argument *cov* is the sample covariance function calculated from the given time series data; the lag window specified by *window* is $\{\lambda(0), \lambda(1), \ldots, \lambda(M)\}$; and w is the frequency variable. Note that the truncation point M should not be greater than the largest lag in the covariance function *cov*. The advantage of using covariance function rather than the time series data directly as the input to SmoothedSpectrumL is that it allows us to try different windows or truncation points without having to recalculate the covariance function each time. Observe also the correspondence of the arguments in SmoothedSpectrumS, where the sample spectrum is weighted, and in SmoothedSpectrumL, where the sample covariance function is weighted.

We now list some commonly used lag windows.

1. *Rectangular* or *Truncated window*: $\lambda(k) = 1$ for $|k| \le M$, and $\lambda(k) = 0$ for $|k| > M$.
2. *Bartlett window*: $\lambda(k) = 1 - |k|/M$ for $|k| \le M$ and 0 otherwise.
3. *Blackman-Tukey window*: $\lambda(k) = 1 - 2a + 2a\cos(\pi k/M)$ for $|k| \le M$ and 0 otherwise. Here a is a constant in the range (0, 0.25]. When $a = 0.23$, the window is called a *Hamming window*, and when $a = 0.25$, a *Hanning window*.
4. *Parzen window*: $\lambda(k) = 1 - 6(k/M)^2 + 6(|k|/M)^3$ for $|k| \le M/2$, $\lambda(k) = 2(1 - |k|/M)^3$ for $M/2 < |k| \le M$, and 0 otherwise.

Example 8.9 Estimate the spectrum from the data in Example 8.7 using the Hanning window ($a = 0.25$) with $M = 12$.

The covariance function is first calculated.

```
In[27]:= cov = CovarianceFunction[data, 30];
```

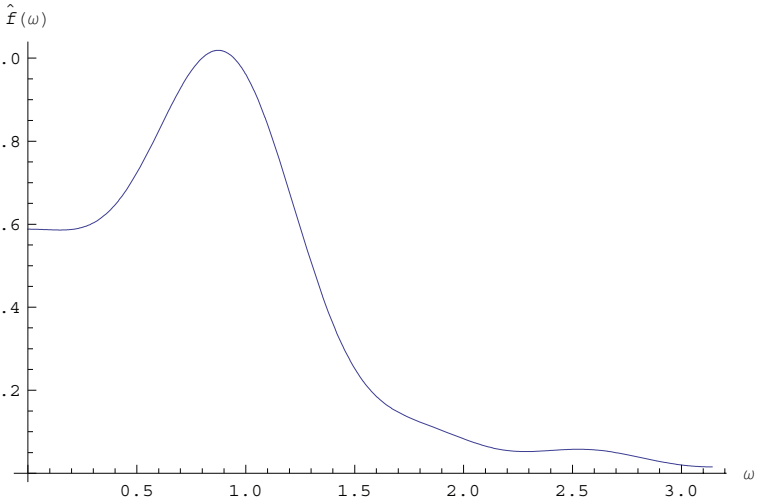
This gives the smoothed spectrum using the Hanning window.

```
In[28]:= spec = SmoothedSpectrumL[cov, Table[0.5 + 0.5 Cos[π k / 12], {k, 0, 12}], ω];
```

This is the plot of the smoothed spectrum.

```
In[29]:= Plot[Evaluate[spec], {ω, 0, π}, AxesLabel -> {"ω", "f̂(ω)"}]
```

Out[29]=



It closely resembles the theoretical spectrum we plotted earlier (see `g1` in Example 8.4). Again, different windows or truncation points can give different smoothed spectra.

The Equivalence of the Two Approaches

While they appear to be quite different, the two approaches to smoothing spectrum are actually intimately related. If we define a spectral window with a continuous weight function $W(\omega)$ as the Fourier transform of the lag window $\{\lambda(k)\}$,

$$W(\omega) = \frac{1}{2\pi} \sum_{k=-M}^{M} \lambda(k)\, e^{-ik\omega}, \tag{8.9}$$

then

$$\lambda(k) = \int_{-\pi}^{\pi} W(\omega)\, e^{ik\omega}\, d\omega. \tag{8.10}$$

It is easy to show that

$$\hat{f}_L(\omega) = \int_{-\pi}^{\pi} \hat{f}(\omega - \alpha)\, W(\alpha)\, d\alpha \sim \sum_{|j| \le n/2} \hat{f}\left(\omega - \omega_j\right) W(j), \tag{8.11}$$

where $\hat{f}(\omega)$ is the continuous sample spectrum defined in (8.5), $W(j) = \frac{2\pi}{n} W(\omega_j)$, and $\omega_j = 2\pi j/n$. (See, for example, Brockwell and Davis (1987), pp. 348–349.) So smoothing in the time domain using the lag window $\{\lambda(k)\}$ is the same as smoothing in the frequency domain using the spectral window $W(\omega)$ and vice versa where $W(\omega)$ and $\{\lambda(k)\}$ are related through (8.9) and (8.10). At the Fourier frequencies the right-hand side of (8.11) is precisely $\hat{f}_S$ defined in (8.7). Depending on the window, it may be easier to do the smoothing in one domain

than in the other. For example, a lag window with small M (narrow width) translates into a spectral window with no cutoff and vice versa. Here we give a couple of spectral windows that correspond to the lag windows listed earlier.

The corresponding spectral window, or *kernel*, is obtained by evaluating the summation in (8.9).

This is the corresponding rectangular window in the frequency domain.

```
In[30]:= (1 + 2 Sum[Cos[k ω], {k, 1, M}]) / (2 π)
```

$$\text{Out[30]} = \frac{1 + \text{Csc}\left[\frac{\omega}{2}\right]\left(-\text{Sin}\left[\frac{\omega}{2}\right] + \text{Sin}\left[\frac{\omega}{2} + \text{M}\,\omega\right]\right)}{2\,\pi}$$

This spectral window denoted by $D_M(\omega)$ is often called the *Dirichlet kernel* of order M. We define it to be `Di[M, ω]`.

This defines `Di` to be the Dirichlet kernel.

```
In[31]:= Di[M_, ω_] = %;
```

Whenever we want to use the Dirichlet kernel we can simply use `Di[M, ω]`. Note that if ω is an integer multiple of 2π the limit with respect to ω can be used.

This is the Bartlett window in the frequency domain.

```
In[32]:= Simplify[(1 + 2 Sum[(1 - k / M) Cos[k ω], {k, 1, M}]) / (2 π)]
```

$$\text{Out[32]} = \frac{\text{Csc}\left[\frac{\omega}{2}\right]^2 \text{Sin}\left[\frac{\text{M}\,\omega}{2}\right]^2}{2\,\text{M}\,\pi}$$

This is often referred to as *Fejér kernel* of order M, written as $F_M(\omega)$, and we define it to be `F[M, ω]`. As with the Dirichlet kernel, the limit with respect to ω can be used for integer multiples of 2π.

This defines `F` to be the Fejér kernel.

```
In[33]:= F[M_, ω_] = %;
```

Similarly we can get the spectral window corresponding to Blackman-Tukey window. The result is a linear combination of Dirichlet kernels:

$$a\,D_M(\omega - \pi/M) + (1 - 2\,a)\,D_M(\omega) + a\,D_M(\omega + \pi/M).$$

For the Parzen window, we need to evaluate the sum separately for M even and for M odd. When M is odd, use $M' - 1$ in place of M (M' even) then substitute M back once the summation is done.

To demonstrate the equivalence of the two approaches we generate a time series of length 50 and calculate the smoothed spectrum using the Bartlett window ($M = 5$) in both the time and frequency domains.

This seeds the random number generator.

```
In[34]:= SeedRandom[239 450]
```

This generates a time series of length 50 from the AR(1) model.

```
In[35]:= data = TimeSeries[ARModel[{0.5}, 1], 50];
```

We first smooth the spectrum in the frequency domain using Fejér kernel. The weights $\{W(j) = W(\omega_j) * 2\pi/n = F[5, 2\pi j/50] * 2\pi/50\}$ are generated using `Table`. $F[5, 0] = 5/(2\pi)$ is added separately using `Prepend` to avoid taking the $\omega = 0$ limit.

The smoothing is performed in the frequency domain.

```
In[36]:= sps = SmoothedSpectrumS[Spectrum[data],
             N[Prepend[Table[F[5, ω], {ω, 2π/50, π, 2π/50}], 5/(2π)] * 2π/50]];
```

The smoothing is done in the time domain.

```
In[37]:= SmoothedSpectrumL[CovarianceFunction[data, 5], Table[1 - k/5, {k, 0, 5}], ω];
```

At the frequencies $\omega_j = 2\pi j/50$, $j = 0, 1, \ldots, 49$, the difference between the two results should be small.

This gives the spectrum at frequencies $\omega_j = 2\pi j/50, j = 0, 1, \ldots, 49$.

```
In[38]:= spl = % /. ω -> Table[N[2π/50 j], {j, 0, 49}];
```

The difference is indeed negligible.

```
In[39]:= Max[Abs[sps - spl]]

Out[39]= 0.0753793
```

1.8.5 Spectrum for Multivariate Time Series

The definition of the spectrum of multivariate time series is similar to that in the univariate case except now we deal with a spectrum matrix (spectral density matrix) $f(\omega)$ given by (see (8.1))

$$f(\omega) = \frac{1}{2\pi}\sum_{-\infty}^{\infty}\Gamma(k)\,e^{-ik\omega}, \tag{8.12}$$

where $\Gamma(k)$ is the covariance matrix defined in Section 1.2.5 (see (2.11)). The i^{th} diagonal element of $f(\omega)$, $f_{ii}(\omega)$, is the (auto)spectrum of the time series i and the off-diagonal element $f_{ij}(\omega) = 1/(2\pi)\sum_{k=-\infty}^{\infty}\gamma_{ij}(k)\,e^{-ik\omega}$ $(i \neq j)$ is called the *cross spectrum* (cross spectral density function) of time series i and j. Note that as in the univariate case, $f(\omega)$ is 2π-periodic. However, since $\gamma_{ij}(k)$ and $\gamma_{ij}(-k)$ need not be the same the cross spectrum $f_{ij}(\omega)$ is in general complex and it can be written as

$$f_{ij}(\omega) = c_{ij}(\omega) - i\,q_{ij}(\omega) = \alpha_{ij}(\omega)\,e^{i\phi_{ij}(\omega)}. \tag{8.13}$$

Various spectra relating to the cross spectrum can be defined from (8.13). The real part of the cross spectrum, $c_{ij}(\omega)$, is called the *cospectrum*, and the negative imaginary part, $q_{ij}(\omega)$, the *quadrature spectrum*; the amplitude of the cross spectrum, $\alpha_{ij}(\omega) = \left|f_{ij}(\omega)\right|$ is called the *amplitude spectrum* and the argument, $\phi_{ij}(\omega)$, the *phase spectrum*. Two other useful functions are the *squared coherency spectrum* (some authors call it coherence) and *gain function* of time series i and j, and they are defined by $K_{ij}^2(\omega) = \left|f_{ij}(\omega)\right|^2 \Big/ \left(f_{ii}(\omega) f_{jj}(\omega)\right)$ and $G_{ij}(\omega) = \left|f_{ij}(\omega)\right| \Big/ f_i(\omega)$, respectively. Although these functions are defined separately due to their different physical interpretations, all the information is contained in our basic spectrum (8.12), and we will see that it is a simple matter to extract these quantities once the spectrum $f(\omega)$ is known.

Spectra of Linear Filters and of ARMA Models

The transfer function of a linear filter in multivariate case is defined similarly except that now the filter weights $\{\Psi_j\}$ are matrices. Equation (8.3) is generalized to

$$f_X(\omega) = \Psi\left(e^{-i\omega}\right) f_Y(\omega)\, \Psi'\left(e^{i\omega}\right) \tag{8.14}$$

and a function implementing (8.14) can be easily written to get the spectrum of a filtered process. This will be left as an exercise for the diligent reader.

The ARMA spectrum can be derived as in the univariate case using (8.14); it is given by

$$f(\omega) = \frac{1}{2\pi} \Phi^{-1}\left(e^{-i\omega}\right) \Theta\left(e^{-i\omega}\right) \Sigma \Theta'\left(e^{i\omega}\right) \Phi'^{-1}\left(e^{i\omega}\right),$$

and the same function `Spectrum[`*model*`, ω]` applies to the multivariate case.

Example 8.10 Here is an example of the spectrum of a multivariate ARMA(1, 1) process.

This defines a bivariate ARMA(1, 1) model.

```
In[40]:= model = ARMAModel[{{{0.6, -0.2}, {0.8, -0.9}}},
            {{{1.1, -0.5}, {0.7, -1.}}}, {{1, 0}, {0, 1}}];
```

This gives the spectrum.

```
In[41]:= spec = Spectrum[model, ω]
```

```
Out[41]= {{(-16.5132 - 20.3947 Cos[ω] - 4.47368 Cos[2 ω]) /
     (2 π (-3.24842 - 0.978947 Cos[ω] + 2. Cos[2 ω])),
    (-11.7395 - 0.552632 Cos[2 ω] + Cos[ω] (-8.56579 - 3.73684 i Sin[ω]) - 3.43421 i Sin[ω]) /
     (2 π (-3.24842 - 0.978947 Cos[ω] + 2. Cos[2 ω]))},
   {(-11.7395 - 0.552632 Cos[2 ω] + Cos[ω] (-8.56579 + 3.73684 i Sin[ω]) + 3.43421 i Sin[ω]) /
     (2 π (-3.24842 - 0.978947 Cos[ω] + 2. Cos[2 ω])),
    (-15.9516 + 6.47368 Cos[ω] - 1.05263 Cos[2 ω]) /
     (2 π (-3.24842 - 0.978947 Cos[ω] + 2. Cos[2 ω]))}}
```

From this spectrum matrix we can easily find other spectra defined in the beginning of this section. Using the fact that $f_{ij}(\omega) = f_{ji}^{*}(\omega)$ (here * means complex conjugate), the cospectrum of $f_{ij}(\omega)$ is simply given by $c_{ij}(\omega) = \left(f_{ij}(\omega) + f_{ji}(\omega)\right)/2$, the quadrature spectrum by $q_{ij}(\omega) = i\left(f_{ij}(\omega) - f_{ji}(\omega)\right)/2$, etc.

Here is the cospectrum of the above ARMA(1, 1) model.

```
In[42]:= Chop[Together[(spec[[1, 2]] + spec[[2, 1]]) / 2]]
```

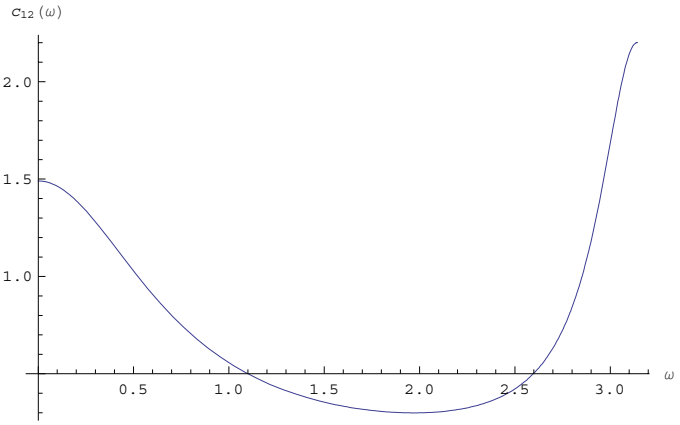
$$\text{Out[42]} = \frac{0.0795775\ (-23.4789 - 17.1316\ \text{Cos}[\omega] - 1.10526\ \text{Cos}[2\,\omega])}{-3.24842 - 0.978947\ \text{Cos}[\omega] + 2.\ \text{Cos}[2\,\omega]}$$

If we only want to see the plot of the spectrum and do not need the explicit function form, we can simply use the function `Re` to extract the real part of $f_{ij}(\omega)$.

This is the plot of the cospectrum.

```
In[43]:= plot0 = Plot[Evaluate[Re[spec[[1, 2]]]], {ω, 0, π}, AxesLabel -> {"ω", "c12 (ω)"}]
```

Out[43] =

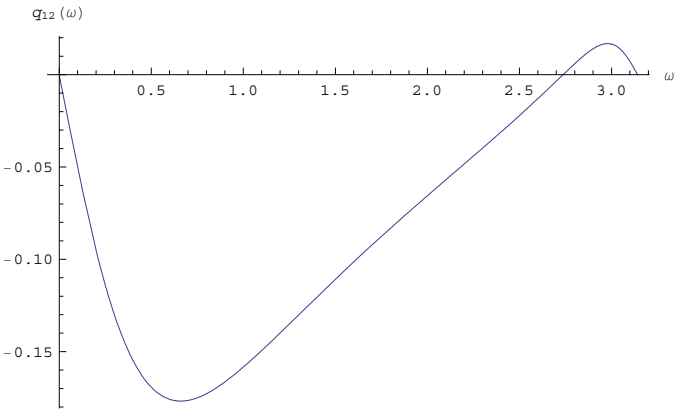


Similarly we can get other spectra.

Here is the plot of the quadrature spectrum.

```
In[44]:= Plot[Evaluate[-Im[spec[[1, 2]]]], {ω, 0, π}, AxesLabel -> {"ω", "q12 (ω)"}]
```

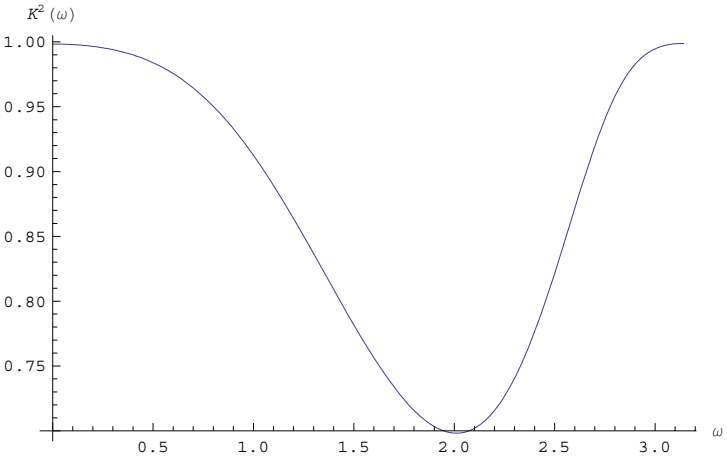
Out[44] =



Here is the plot of the squared coherency.

```
In[45]:= Plot[Evaluate[Abs[spec[[1, 2]]]^2 / (spec[[1, 1]] spec[[2, 2]])],
           {ω, 0, π}, AxesLabel -> {"ω", "K^2 (ω)"}]
```

Out[45]=



Various other spectra can be plotted using the built-in *Mathematica* functions `Abs`, `Arg`, etc.

Estimation of the Spectrum

We can estimate the spectrum of an ARMA process by fitting an m-variate ARMA model to m-variate data and using `Spectrum[`*model*`, ω]` to get the estimated spectra. The sample spectrum of a multivariate process is again given by (8.5) with the sample covariance matrix $\hat{\Gamma}(k)$ in place of univariate sample covariance function $\hat{\gamma}(k)$. As in the univariate case, the function `Spectrum[data]` gives the sample spectrum of m-variate *data* at frequencies $\omega_j = 2\pi j/n$ for $j = 0, 1, \ldots, n-1$, and its output is $\{\hat{f}(\omega_0), \hat{f}(\omega_1), \ldots, \hat{f}(\omega_{n-1})\}$ where $\hat{f}(\omega_j)$ is an $m \times m$ matrix. Again, the function `normalizedspectrum` defined earlier (see Example 8.7) can be used to estimate the normalized spectrum in the multivariate case.

Example 8.11 Here is the sample spectrum of a bivariate time series data of length 100 generated from the ARMA(1, 1) model used in Example 8.10.

The random number generator is seeded.

```
In[46]:= SeedRandom[902 587]
```

This generates the time series.

```
In[47]:= tseries = TimeSeries[model, 100];
```

This calculates the sample spectrum.

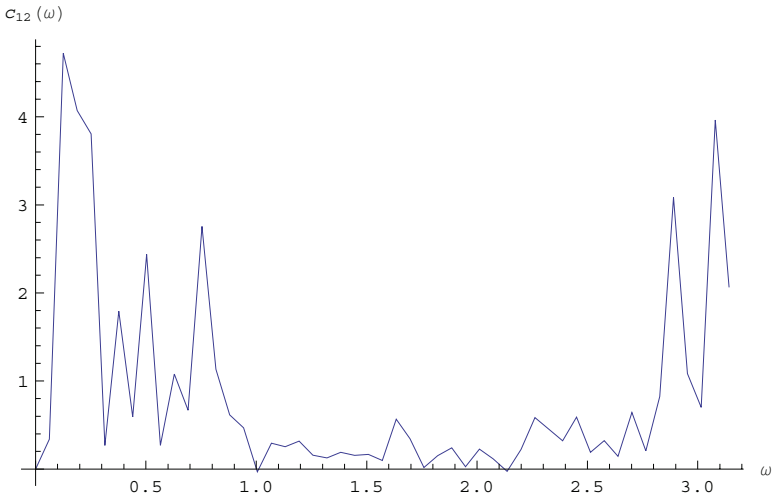
```
In[48]:= spec = Spectrum[tseries];
```

And from this sample spectrum, we can obtain all the other spectra. For example, the cospectrum can be obtained by extracting the (1, 2) component of the sample spectrum matrix and taking its real part.

Here we plot the sample cospectrum.

```
In[49]:= plotspectrum[Re[spec[[All, 1, 2]]], AxesLabel -> {"ω", "c12(ω)"}, PlotRange -> All]
```

Out[49]=



To smooth the spectrum we proceed as in the univariate case. `SmoothedSpectrumL[cov, window, ω]` and `SmoothedSpectrumS[spec, window]` are used to get the smoothed spectrum matrix $\hat{f}_L(\omega)$ and matrices $\hat{f}_S(\omega_j)$ for $j = 0, 1, \ldots, n-1$, respectively. They are defined in (8.7) and (8.8) with $\hat{f}$ and $\hat{\gamma}$ now replaced by the sample spectrum matrix and sample covariance matrix. Matrix weights can also be used. For example, W and Λ in $window = \{W(0), W(1), \ldots W(M)\}$ and $window = \{\Lambda(0), \Lambda(1), \ldots \Lambda(M)\}$ can be matrices so that different components do not have to be smoothed with the same window. The (i, j) component of the spectrum will be smoothed according to

$$\hat{f}_S((\omega_k))_{ij} = \sum_{l=-M}^{M} W((l))_{ij} \hat{f}_{ij}(\omega_{k-l})$$

or

$$\hat{f}_L((\omega))_{ij} = \frac{1}{2\pi} \sum_{k=-M}^{M} \Lambda((k))_{ij} \hat{\Gamma}((k))_{ij} e^{-ik\omega},$$

where $W(-k) = W'(k)$ and $\Lambda(-k) = \Lambda'(k)$ are assumed throughout. If scalar weights are entered, all the components will be smoothed using the same given weights.

Example 8.12 Smooth the spectrum of Example 8.11 using the Bartlett window.

We first calculate the sample covariance function.

```
In[50]:= cov = CovarianceFunction[tseries, 30];
```

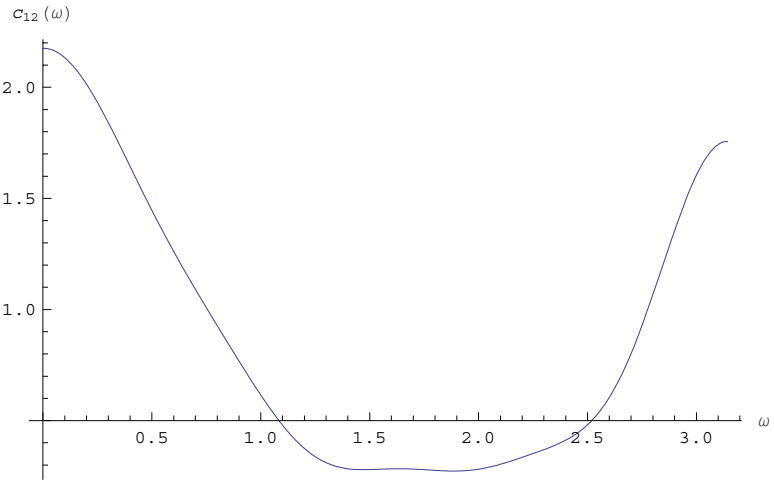
This gives the smoothed spectrum using the Bartlett window.

```
In[51]:= spec1 = SmoothedSpectrumL[cov, Table[1 - k/10, {k, 0, 10}], ω];
```

This is the plot of the smoothed cospectrum.

```
In[52]:= plot1 = Plot[Evaluate[Re[spec1[[1, 2]]]], {ω, 0, Pi}, AxesLabel -> {"ω", "c12 (ω)"}]
```

Out[52]=



Next we look at the smoothed cospectrum using the Daniell window.

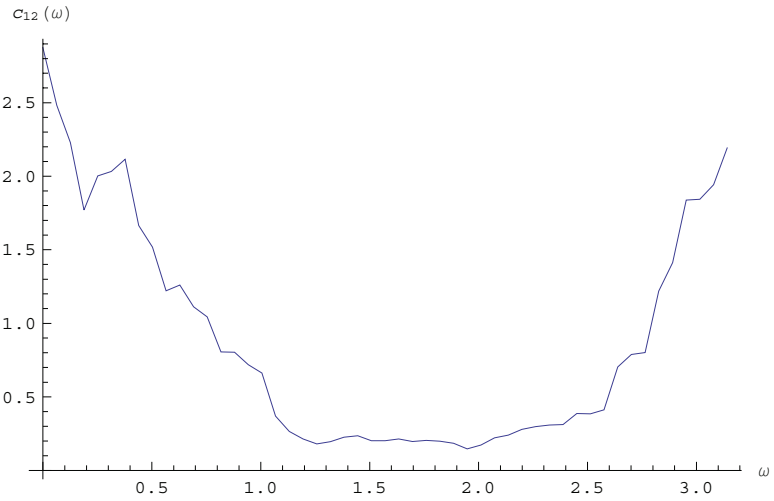
This gives the smoothed spectrum using the Daniell window.

```
In[53]:= spec2 = SmoothedSpectrumS[spec, Table[1 / 9, {5}]];
```

This is the plot of the cospectrum.

```
In[54]:= plot2 = plotspectrum[Re[spec2[[All, 1, 2]]], AxesLabel -> {"ω", "c12 (ω)"}]
```

Out[54]=

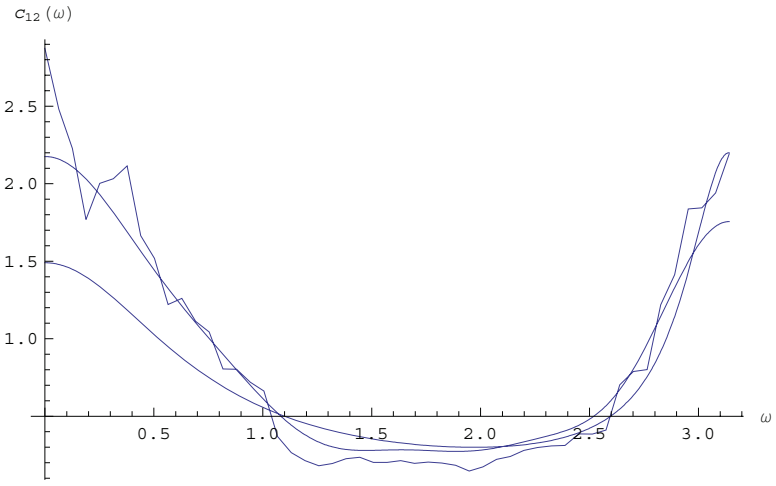


For comparison we display the two smoothed cospectra together with the theoretical cospectrum (see `plot0` in Example 8.10).

This displays the three cospectra together.

```
In[55]:= Show[plot0, plot1, plot2, PlotRange -> All]
```

Out[55]=



It is a simple matter to get estimates for other spectra once $\hat{f}(\omega)$ is known. For example, as we have demonstrated, the cospectrum is simply the real part of the appropriate component of the spectrum and it is trivial to extract it using the `Re` and `Map` functions. You can also define your own functions to extract different spectra. For example, you can define the following function to get the squared coherency between series i and j.

Note here that `spec` should be a smoothed discrete spectrum so that the denominator will not be zero at $\omega = 0$.

```
In[56]:= squaredcoherency[spec_, i_, j_] :=
          Chop[#[[i, j]] Conjugate[#[[i, j]]]] / (#[[i, i]] #[[j, j]]) & /@ spec
```

1.9 Structural Models and the Kalman Filter

1.9.1 Structural Models

So far we have discussed ARMA type of models where the trend and often the seasonal components of a given time series are removed before detailed analysis. A structural time series model, on the other hand, is designed to model trend and seasonal components explicitly. For example, the basic structural model is defined by

$$y_t = \mu_t + \gamma_t + \epsilon_t \tag{9.1}$$

where μ represents the trend and γ is the cyclic or seasonal component, while ϵ is the irregular or noise part. Both μ and γ evolve according to their own equations. For example, we can have a linear trend with

$$\begin{aligned} \mu_t &= \mu_{t-1} + \beta_{t-1} + \eta_t \\ \beta_t &= \beta_{t-1} + \zeta_t \end{aligned} \tag{9.2}$$

(where $\beta_0 \neq 0$) and the seasonality, γ_t, modeled as,

$$\gamma_t = -\sum_{j=1}^{s-1} \gamma_{t-j} + \omega_t, \tag{9.3}$$

where $\{\epsilon_t\}$, $\{\eta_t\}$, $\{\zeta_t\}$, and $\{\omega_t\}$ are all white noise variables with zero mean; they are assumed to be independent of each other.

When the seasonal component is absent, we can define a simpler version of the basic structural model called the *local linear trend model* by

$$\begin{aligned} y_t &= \mu_t + \epsilon_t \\ \mu_t &= \mu_{t-1} + \beta_{t-1} + \eta_t \\ \beta_t &= \beta_{t-1} + \zeta_t \end{aligned} \tag{9.4}$$

or an even simpler *local level model* by

$$\begin{aligned} y_t &= \mu_t + \epsilon_t \\ \mu_t &= \mu_{t-1} + \eta_t. \end{aligned} \tag{9.5}$$

In the above models, the "trend" μ_t (or seasonality γ_t) is not observable. Often the task is to infer the behavior of the trend or some other quantities from the time series $\{y_t\}$ we observe. This class of problems is most conveniently treated by casting the models into a state-space form and using the Kalman filter technique.

1.9.2 State-Space Form and the Kalman Filter

In general, a state-space representation of a discrete time series model has the form:

$$Y_t = G_t X_t + d_t + W_t \tag{9.6}$$

$$X_t = F_t X_{t-1} + c_t + V_t, \tag{9.7}$$

where $\{Y_t\}$, which can be either a scalar or a vector, is the process we observe and (9.6) is called the "observation equation". $\{Y_t\}$ depends on another process $\{X_t\}$ that cannot be observed directly. X_t, which is a vector in general, is referred to as the "state" of the system at time t, and it evolves according to the "state equation" (9.7). F, G, c, and d are known matrices or vectors that may be dependent on time. $\{W_t\}$ and $\{V_t\}$ are independent Gaussian white noise variables with zero mean and covariance matrices given by $E\, V_t\, V_s' = \delta_{t\,s}\, Q_t$ and $E\, W_t\, W_s' = \delta_{t\,s}\, R_t$, respectively, where the prime denotes transpose.

Often, we would like to obtain an estimate of the unobservable state vector X based on the information available at time t, I_t, where I_t contains the observations of Y up to Y_t. The Kalman filter provides a recursive procedure for calculating the estimate of the state vector. Let $\hat{X}_{t|s}$ denote the best linear estimator of X_t based on the information up to and including time s and $P_{t|s}$, its mean square error $E\left(X_t - \hat{X}_{t|s}\right)\left(X_t - \hat{X}_{t|s}\right)'$. The following equations constitute the Kalman filter:

$$\hat{X}_{t|t} = \hat{X}_{t|t-1} + K_t\left(Y_t - G_t\, \hat{X}_{t|t-1} - d_t\right) \tag{9.8}$$

$$\hat{X}_{t+1|t} = F_{t+1}\, \hat{X}_{t|t} + c_{t+1} \tag{9.9}$$

$$P_{t|t} = P_{t|t-1} - K_t\, G_t\, P_{t|t-1} \tag{9.10}$$

$$P_{t+1|t} = Q_{t+1} + F_{t+1}\, P_{t|t}\, F_{t+1}', \tag{9.11}$$

where

$$K_t = P_{t|t-1}\, G_t'\left(G_t\, P_{t|t-1}\, G_t' + R_t\right)^{-1}$$

is called the Kalman gain. The above equations can be used to calculate the estimate $\hat{X}_{t|t}$ and its mean square error $P_{t|t}$, recursively. Clearly, like all recursive procedures this needs initial values, $\left\{\hat{X}_{m+1|m},\, P_{m+1|m}\right\}$, to start the recursion. We will first present the function in *Mathematica* that implements these equations given the initial values and discuss ways of calculating the initial values in Section 1.9.3.

The *Mathematica* function that performs Kalman filtering ((9.8) to (9.11)) given the initial values $\left\{\hat{X}_{m+1|m},\, P_{m+1|m}\right\}$ and the data Y_{m+1} is

$$\texttt{KalmanFilter[}Y_{m+1}\texttt{,}\left\{\hat{X}_{m+1|m},\, P_{m+1|m}\right\}\texttt{,}\, F_{m+2}\texttt{,}\, G_{m+1}\texttt{,}\, Q_{m+2}\texttt{,}\, R_{m+1}\texttt{,}\, c_{m+2}\texttt{,}\, d_{m+1}\texttt{]}.$$

It yields $\{\hat{X}_{m+2|m+1}, P_{m+2|m+1}, \hat{X}_{m+1|m+1}, P_{m+1|m+1}\}$. If $c_t = 0$ and $d_t = 0$, the last two arguments can be omitted. When all the known matrices and vectors are independent of time,

$$\texttt{KalmanFilter}[\{Y_{m+1}, Y_{m+2}, \ldots, Y_T\}, \{\hat{X}_{m+1|m}, P_{m+1|m}\}, F, G, Q, R, c, d]$$

gives

$$\{\{\hat{X}_{m+2|m+1}, P_{m+2|m+1}, \hat{X}_{m+1|m+1}, P_{m+1|m+1}\}, \{\hat{X}_{m+3|m+2}, P_{m+3|m+2}, \hat{X}_{m+2|m+2}, P_{m+2|m+2}\}, \ldots, \{\hat{X}_{T+1|T}, P_{T+1|T}, \hat{X}_{T|T}, P_{T|T}\}\}.$$

However, if any one of F, G, Q, R, c, d is time dependent, the above arguments to `KalmanFilter`, F, G, Q, R, c, d, should be replaced by $\{F_{m+2}, F_{m+3}, \ldots, F_{T+1}\}$, $\{G_{m+1}, G_{m+2}, \ldots, G_T\}$, $\{Q_{m+2}, Q_{m+3}, \ldots, Q_{T+1}\}$, $\{R_{m+1}, R_{m+2}, \ldots, R_T\}$, $\{c_{m+2}, c_{m+3}, \ldots, c_{T+1}\}$, and $\{d_{m+1}, d_{m+2}, \ldots, d_T\}$, respectively.

The Kalman filter gives the estimate of the state variable X at time t given the information up to t, $\hat{X}_{t|t}$. As more and more information is accumulated, i.e., $\{Y_{t+i}\}$ $(i = 1, 2, \ldots, s)$ are known, the estimate of X_t can be improved by making use of the extra available information. Kalman smoothing is a way of getting the estimate of X_t, given the information I_T, where $T > t$:

$$\hat{X}_{t|T} = \hat{X}_{t|t} + J_t\left(\hat{X}_{t+1|T} - \hat{X}_{t+1|t}\right) \tag{9.12}$$

and

$$P_{t|T} = P_{t|t} + J_t(P_{t+1|T} - P_{t+1|t})\, J_t', \tag{9.13}$$

where $J_t \equiv P_{t|t}\, F_{t+1}'\, P_{t+1|t}^{-1}$. The two equations given above are often referred to as the Kalman fixed-point smoother.

To obtain $\hat{X}_{t|T}$ and $P_{t|T}$, we first use the Kalman filter to get $\hat{X}_{t|t}$ and $P_{t|t}$ for t up to T. Then using (9.12) and (9.13), $\hat{X}_{T-1|T}, \hat{X}_{T-2|T}, \ldots, \hat{X}_{t|T}$ and $P_{T-1|T}, P_{T-2|T}, \ldots, P_{t|T}$ can be calculated recursively. The *Mathematica* function

$$\texttt{KalmanSmoothing}[\textit{filterresult}, F]$$

gives $\{\{\hat{X}_{m+1|T}, \hat{X}_{m+2|T}, \ldots, \hat{X}_{T|T}\}, \{P_{m+1|T}, P_{m+2|T}, \ldots, P_{T|T}\}\}$, where *filterresult* is the output of `KalmanFilter`, i.e., $\{\{\hat{X}_{m+2|m+1}, P_{m+2|m+1}, \hat{X}_{m+1|m+1}, P_{m+1|m+1}\}, \ldots, \{\hat{X}_{T+1|T}, P_{T+1|T}, \hat{X}_{T|T}, P_{T|T}\}\}$, and F is the transition matrix in state equation (9.7). Note that if F is time dependent, the second argument of `KalmanSmoothing` should be $\{F_{m+2}, F_{m+3}, \ldots, F_{T+1}\}$.

The Kalman prediction is the calculation of the estimate of the future values X_{t+h} $(h > 0)$ based on the current information I_t, i.e., the calculation of $\hat{X}_{t+h|t}$. It is easy to see that

$$\hat{X}_{t+h|t} = F_{t+h}\, \hat{X}_{t+h-1|t} + c_{t+h}$$

and

$$P_{t+h|t} = F_{t+h}\, P_{t+h-1|t}\, F_{t+h}' + Q_{t+h}.$$

So starting from $\{\hat{X}_{t+1|t}, P_{t+1|t}\}$ obtained from the Kalman filtering, the above equations can be iterated to get $\{\hat{X}_{t+h|t}, P_{t+h|t}\}$. It is straightforward to see that the prediction for Y is

$$\hat{Y}_{t+h|t} = G_{t+h}\,\hat{X}_{t+h|t} + d_{t+h},$$

and the corresponding mean square error $f_{t+h|t}$ is

$$f_{t+h|t} = G_{t+h}\,P_{t+h|t}\,G'_{t+h} + R_{t+h}.$$

The function

$$\texttt{KalmanPredictor}[\{\hat{X}_{t+1|t},\ P_{t+1|t}\},\ F,\ Q,\ c,\ h],$$

when the known matrices and vectors are time independent, or

$$\texttt{KalmanPredictor}[\{\hat{X}_{t+1|t},\ P_{t+1|t}\},\ \{F_{t+2},\ \ldots, F_{t+h}\},\ \{Q_{t+2},\ \ldots,\ Q_{t+h}\},\ \{c_{t+2}, \ldots, c_{t+h}\}],$$

when the known matrices and vectors are time dependent, gives the next h predicted values and their mean square errors $\{\{\hat{X}_{t+1|t}, \hat{X}_{t+2|t}, \ldots, \hat{X}_{t+h|t}\}, \{P_{t+1|t}, P_{t+2|t}, \ldots, P_{t+h|t}\}\}$. Again, the argument c can be omitted if it is always 0.

1.9.3 Applications of the Kalman Filter

The Kalman filtering technique can be used conveniently in the analysis of certain time series, once we write the time series model in a state-space form. In the following we will mention a few applications of the Kalman filter and illustrate some of them by examples. For a detailed treatment of the Kalman filter see, for example, Harvey (1989), Chapter 3.

The simple local level model (9.5) is in fact in the state-space form already with $X_t = \mu_t$, $G_t = 1$, $F_t = 1$, $c_t = 0$, and $d_t = 0$. It is easy to see that the local linear trend model (9.4) can be written in the state-space form as

$$y_t = (1,\ 0)\begin{pmatrix}\mu_t\\ \beta_t\end{pmatrix} + \epsilon_t$$

and

$$\begin{pmatrix}\mu_{t+1}\\ \beta_{t+1}\end{pmatrix} = \begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}\begin{pmatrix}\mu_t\\ \beta_t\end{pmatrix} + \begin{pmatrix}\eta_{t+1}\\ \zeta_{t+1}\end{pmatrix}$$

with

$$R = \sigma_\epsilon^2 \text{ and } Q = \begin{pmatrix}\sigma_\eta^2 & 0\\ 0 & \sigma_\zeta^2\end{pmatrix}.$$

Similarly, the basic structural model ((9.1) to (9.3)) when $s = 4$ is equivalent to

$$y_t = (1, 0, 1, 0, 0)\,\alpha_t + \epsilon_t$$

where

$$\alpha_t = \begin{pmatrix} \mu_t \\ \beta_t \\ \gamma_t \\ \gamma_{t-1} \\ \gamma_{t-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \mu_{t-1} \\ \beta_{t-1} \\ \gamma_{t-1} \\ \gamma_{t-2} \\ \gamma_{t-3} \end{pmatrix} + \begin{pmatrix} \eta_t \\ \zeta_t \\ \omega_t \\ 0 \\ 0 \end{pmatrix}.$$

ARMA models can also be cast into a state-space form, which in general is nonunique. For example, one particular representation of the AR(2) model in state-space form is (for the state-space form of an ARMA(p, q) model see Harvey (1989), Chapter 3)

$$y_t = (1, 0)\,x_t$$

and

$$x_t = \begin{pmatrix} y_t \\ \phi_2\, y_{t-1} \end{pmatrix} = \begin{pmatrix} \phi_1 & 1 \\ \phi_2 & 0 \end{pmatrix} \begin{pmatrix} y_{t-1} \\ \phi_2\, y_{t-2} \end{pmatrix} + \begin{pmatrix} z_t \\ 0 \end{pmatrix}.$$

An alternative representation for the state vector is

$$x_t = \begin{pmatrix} y_t \\ y_{t-1} \end{pmatrix} = \begin{pmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} y_{t-1} \\ y_{t-2} \end{pmatrix} + \begin{pmatrix} z_t \\ 0 \end{pmatrix}.$$

Initializing the Kalman Filter

The initial values of $\hat{X}_{t|t-1}$ and $P_{t|t-1}$ have to be given in order to start the Kalman filter. If the coefficients in (9.7) are all time independent and the eigenvalues of the transition matrix F are inside the unit circle, the state vector is stationary and the initial value $\hat{X}_{1|0}$ can be solved as

$$\hat{X}_{1|0} = (I - F)^{-1}\, c$$

and $P_{1|0}$ obtained from

$$P_{1|0} = P = F\,P\,F' + Q.$$

The solution to the above equation is

$$vec\,(P) = (I - F \otimes F)^{-1}\, vec\,(Q) \tag{9.14}$$

where $vec(P)$ is the vector obtained by stacking the columns of P one below the other.

For example, the AR(2) model with $\phi_1 = 0.9$, $\phi_2 = -0.5$, and $\sigma^2 = 1$ is stationary and the initial values are $\hat{X}_{1|0} = (0, 0)'$ and P can be obtained using (9.14). The following *Mathematica* program solves for P.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

`getP` solves for the initial value $P_{1|0}$ for a stationary state equation. Note that Q is always symmetric.

```
In[2]:= getP[F_, Q_] :=
         Partition[Inverse[IdentityMatrix[#^2] - Flatten[Map[Flatten, Transpose /@
                  Outer[Times, F, F], {2}], 1]].Flatten[Q], #] &[Length[F]]
```

This gives $P_{1|0}$ which is used to start the Kalman filter recursion.

```
In[3]:= getP[{{0.9, 1}, {-0.5, 0}}, {{1, 0}, {0, 0}}]

Out[3]= {{2.08333, -0.625}, {-0.625, 0.520833}}
```

Consider the case when the state equation is not stationary, and we do not have any prior information about the distribution of the state vector. In general, if the dimension of X is m, we can use the first m values of Y (assuming that Y is a scalar) to calculate $\hat{X}_{m+1|m}$ and $P_{m+1|m}$ and start the recursion from there. To obtain $\hat{X}_{m+1|m}$ and $P_{m+1|m}$, we can use the Kalman filter ((9.8) to (9.11)) with $E(X_1 \mid I_0) = E(X_1) = \mathbf{0}$ and the so-called diffuse prior $P_{1|0} = \kappa I$, where I is an identity matrix and κ is eventually set to infinity. For example, the local level model is not stationary. In the absence of any prior information, we use $\hat{X}_{1|0} = E(X_1) = 0$ and $P_{1|0} = k$ and $\hat{X}_{2|1}$ and $P_{2|1}$ can be obtained as follows.

This gives $\{\hat{X}_{2|1}, P_{2|1}, \hat{X}_{1|1}, P_{1|1}\}$. Note that the result depends on the first data point y_1.

```
In[4]:= KalmanFilter[y1, {0, k}, 1, 1, Q, R]

Out[4]= {k y1/(k + R), k + Q - k^2/(k + R), k y1/(k + R), k - k^2/(k + R)}
```

Now we take the limit $k \to \infty$. Note that we get a finite result, and the first two elements will be used as the initial values to start the recursion.

```
In[5]:= Limit[%, k -> Infinity]

Out[5]= {y1, Q + R, y1, R}
```

For the local linear trend model, X is a two-dimensional vector ($m = 2$), and the calculation of the initial values using a diffuse prior, in general, needs the first two data points y_1 and y_2.

This gives $\{\{\hat{X}_{2|1}, P_{2|1}, \hat{X}_{1|1}, P_{1|1}\}, \{\hat{X}_{3|2}, P_{3|2}, \hat{X}_{2|2}, P_{2|2}\}\}$.

```
In[6]:= KalmanFilter[{y1, y2}, {{0, 0}, k {{1, 0}, {0, 1}}},
          {{1, 1}, {0, 1}}, {{1, 0}}, {{q1, 0}, {0, q2}}, r];
```

Note that $P_{2|1}$ is infinite, whereas $P_{3|2}$ is finite.

```
In[7]:= Limit[Take[%, All, 2], k -> Infinity]

Out[7]= {{{y1, 0}, {{∞, ∞}, {∞, ∞}}},
         {{-y1 + 2 y2, -y1 + y2}, {{2 q1 + q2 + 5 r, q1 + q2 + 3 r}, {q1 + q2 + 3 r, q1 + 2 (q2 + r)}}}}
```

Here $\{\hat{X}_{3|2}, P_{3|2}\}$ is extracted, and it can be used as the initial value to start the Kalman filter recursion.

```
In[8]:= %[[-1]]

Out[8]= {{-y1 + 2 y2, -y1 + y2}, {{2 q1 + q2 + 5 r, q1 + q2 + 3 r}, {q1 + q2 + 3 r, q1 + 2 (q2 + r)}}}
```

Although in principle `KalmanFilter` can be used to calculate $\{\hat{X}_{m+1|m}, P_{m+1|m}\}$ using the first m data points as illustrated above, it is infeasible when m is large because of the symbolic parameter k involved in the calculation. Alternatively, we can obtain the initial values by writing X_m in terms of first m data points of Y, and therefore, solve for $\hat{X}_{m|m}$. The following is a program that gives $\{\hat{X}_{m+1|m}, P_{m+1|m}\}$. Note that the program is only for the case where Y is a scalar and the coefficient matrices are independent of time. For other ways of calculating starting values, see Harvey (1989), Chapter 3.

This yields $\{\hat{X}_{m+1|m}, P_{m+1|m}\}$ given the known matrices and vectors of a state model and the first m values of data. Note that the first argument contains the first m values of the data, F and Q are $m \times m$ matrices, and G is a $1 \times m$ matrix. When $m = 1$, every input argument is a scalar.

```
In[9]:= getInit[y_?AtomQ, F_, G_, Q_, R_, c_:0, d_:0] := {# * (y - d) + c, #^2 * R + Q} &[F / G];

In[10]:= getInit[y_?VectorQ, F_, G_, Q_, R_, c_:0, d_:0] :=
          Module[{m = Length[y], invF = Inverse[F], temp, zero, finvtemp, tempzero},
           zero = Table[0, {m}];
           temp = Flatten[NestList[#.invF &, G, m - 1], 1];
           finvtemp = F.Inverse[Reverse[temp]];
           tempzero = Append[Reverse[Rest[temp]], zero];
           {finvtemp.(y - d + If[c === 0, 0, Reverse[Accumulate[Rest[temp]]].c]) + c,
            finvtemp.(Outer[Total[MapThread[Dot, {#1, #2}]] &,
                 NestList[Append[Rest[#], zero] &, #.Q & /@ tempzero, m - 1],
                 NestList[Append[Rest[#], zero] &, tempzero, m - 1], 1] +
                R IdentityMatrix[m]).Transpose[finvtemp] + Q}]
```

For example, this gives the same initial values for the local linear trend model as above but is much faster.

```
In[11]:= getInit[{y1, y2}, {{1, 1}, {0, 1}}, {{1, 0}}, {{q1, 0}, {0, q2}}, r]

Out[11]= {{-y1 + 2 y2, -y1 + y2}, {{2 q1 + q2 + 5 r, q1 + q2 + 3 r}, {q1 + q2 + 3 r, q1 + 2 q2 + 2 r}}}
```

Kalman Filtering and Kalman Smoothing

Often, the state variable X represents some quantity we are interested in knowing. The Kalman filter enables us to estimate the state variable X from the observations of Y. For example, in the local level model, μ is the trend and the estimation of μ_t, given $y_1, y_2, \ldots, y_t$, is given by the Kalman filter (9.8) to (9.11).

Example 9.1 To illustrate the use of the Kalman filter, we generate a series according to the local level model (9.5) with $R = E\,\epsilon^2 = 0.5$ and $Q = E\,\eta^2 = 1$.

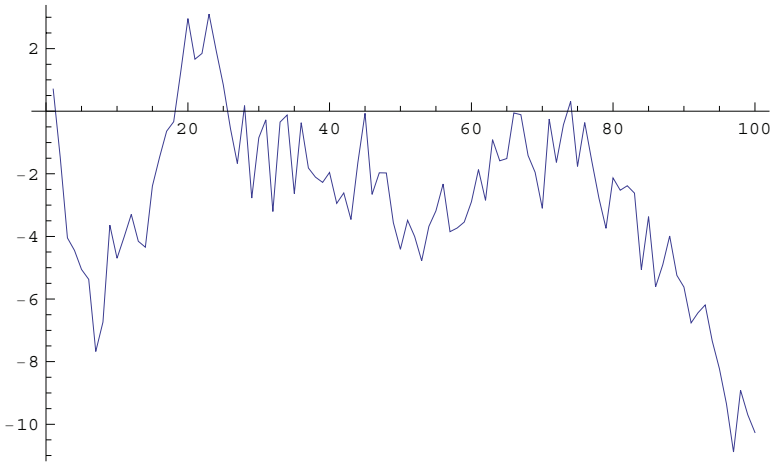
This generates a time series of length 100 according to the local level model (9.5).

```
In[12]:= (SeedRandom[34 561];
          eta = RandomSequence[0, 1, 100];
          mu = Accumulate[eta];
          data1 = RandomSequence[0, 0.5, 100] + mu;)
```

This plots the series.

```
In[13]:= ListLinePlot[data1]
```

Out[13]=



As illustrated earlier, these are $\hat{X}_{2|1}$ and $P_{2|1}$ and they are used as the starting values for `KalmanFilter`.

```
In[14]:= init = Limit[Take[KalmanFilter[data1[[1]], {0, k}, 1, 1, 1, 0.5], 2], k -> Infinity]

Out[14]= {0.708558, 1.5}
```

Using `getInit` we get the same result.

```
In[15]:= getInit[data1[[1]], 1, 1, 1, 0.5]

Out[15]= {0.708558, 1.5}
```

`kf` contains the result of the Kalman filtering. (Note that we have suppressed the output.) Note that y_1 is used to obtain the initial values, so the data now starts from y_2.

```
In[16]:= kf = KalmanFilter[Rest[data1], init, 1, 1, 1, 0.5];
```

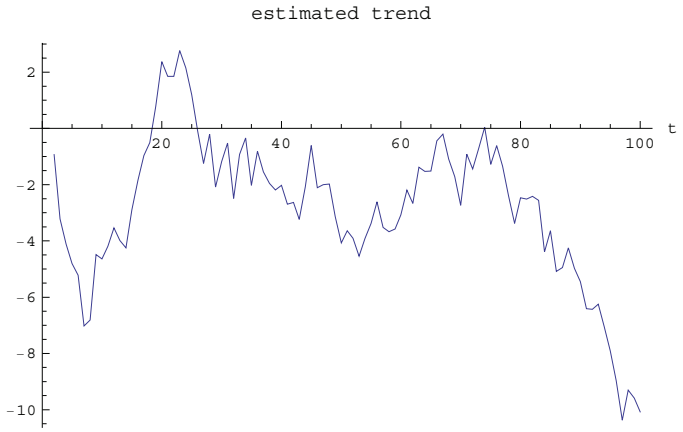
From the Kalman filter result `kf`, we extract $\hat{\mu}_{t|t}$, the estimated values of μ.

```
In[17]:= kf[[All, 3]];
```

Here the estimate of μ_t given $\{y_1, y_2, \ldots, y_t\}$ is plotted as a function of t.

```
In[18]:= ListLinePlot[%, DataRange -> {2, 100},
          AxesLabel -> {"t", ""}, PlotLabel -> "estimated trend"]
```

Out[18]=



$X_{t|t}$ is the estimate of X_t based on the information up to t. However, if we know Y_t up to $t = T$, we can use the information up to T to improve our estimate of X_t. This is called Kalman smoothing.

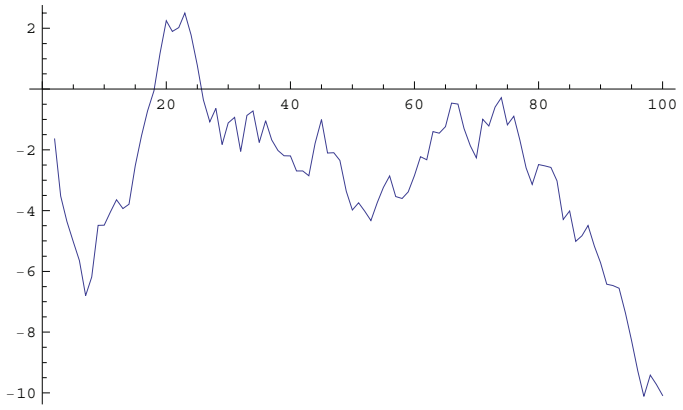
This gives the Kalman smoothing result.

```
In[19]:= KalmanSmoothing[kf, 1];
```

The smoothed values are plotted.

```
In[20]:= ListLinePlot[%[[1]], DataRange -> {2, 100}]
```

Out[20]=



Example 9.2 As a second example, we use the Kalman filter to estimate the trend and the cyclic component in a particular case of the basic structural model given in (9.1) to (9.3).

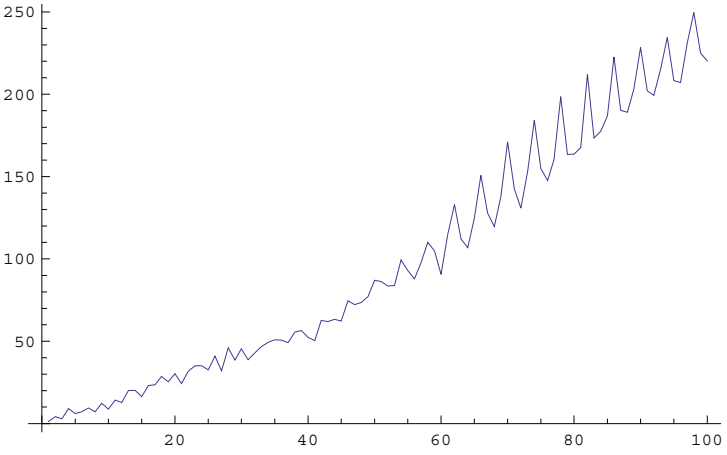
This generates a time series of length 100 from the basic structural model. (See 9.1 to 9.3.)

```
In[21]:= (SeedRandom[3 457 091];
          eta = RandomSequence[0, 1, 100];
          zeta = RandomSequence[0, 0.05, 100];
          beta = Most[Accumulate[Flatten[{0.5, zeta}]]];
          mu = Accumulate[beta + eta];
          gamma = TimeSeries[ARModel[{-1, -1, -1}, 5], 100, {0, 0, 0}];
          y = mu + gamma + RandomSequence[0, 0.5, 100];)
```

The series is plotted here.

```
In[22]:= ListLinePlot[y]
```

Out[22]=



Various matrices that are needed are defined.

```
In[23]:= (F = {{1, 1, 0, 0, 0}, {0, 1, 0, 0, 0},
             {0, 0, -1, -1, -1}, {0, 0, 1, 0, 0}, {0, 0, 0, 1, 0}};
          G = {{1, 0, 1, 0, 0}};
          Q = {{1, 0, 0, 0, 0}, {0, 1, 0, 0, 0}, {0, 0, 1, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}};
          R = 1;)
```

Since $m = 5$, the first five points of data y are needed to calculate the initial values.

```
In[24]:= init = getInit[Take[y, 5], F, G, Q, R]
```

Out[24]=

$$\left\{\{8.64177,\ 1.21377,\ 0.499978,\ -1.32112,\ 2.93425\},\right.$$
$$\left\{\left\{\frac{247}{32}, \frac{7}{2}, \frac{57}{32}, -\frac{75}{32}, -\frac{11}{32}\right\}, \left\{\frac{7}{2}, \frac{27}{8}, \frac{1}{2}, -\frac{7}{8}, 0\right\}, \left\{\frac{57}{32}, \frac{1}{2}, \frac{119}{32}, -\frac{37}{32}, -\frac{37}{32}\right\},\right.$$
$$\left.\left.\left\{-\frac{75}{32}, -\frac{7}{8}, -\frac{37}{32}, \frac{59}{32}, -\frac{1}{32}\right\}, \left\{-\frac{11}{32}, 0, -\frac{37}{32}, -\frac{1}{32}, \frac{47}{32}\right\}\right\}\right\}$$

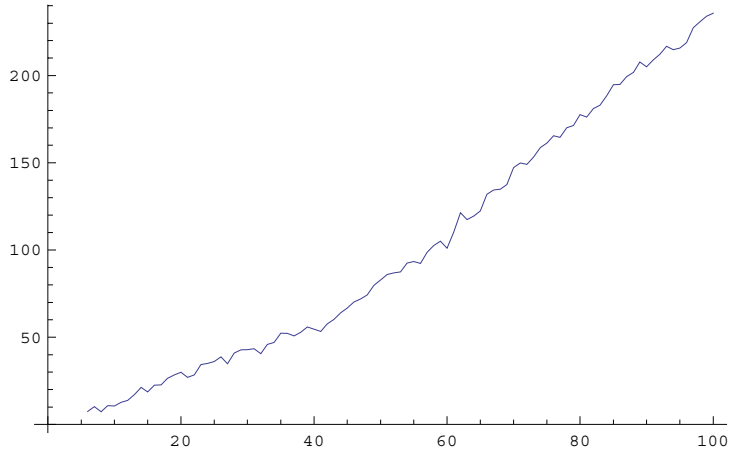
This gives the Kalman filtering result. Note that the numerical value of `init` is used to speed up the calculation.

```
In[25]:= kf = KalmanFilter[Drop[y, 5], N[init], F, G, Q, R];
```

The first component of $\hat{X}_{t|t}$ is the estimated trend $\hat{\mu}_{t|t}$. The trend is extracted and plotted.

```
In[26]:= ListLinePlot[kf[[All, 3, 1]], DataRange -> {6, 100}]
```

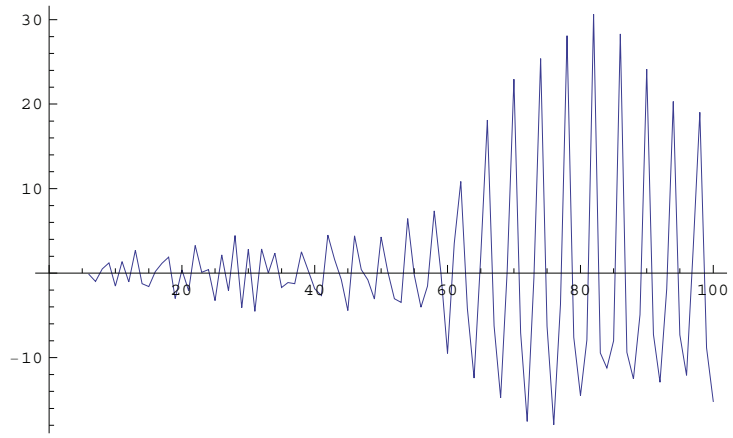
Out[26]=



Similarly, the third component of $\hat{X}_{t|t}$ is the estimated seasonal $\hat{\gamma}_{t|t}$. Its plot is shown here.

```
In[27]:= ListLinePlot[kf[[All, 3, 3]], DataRange -> {6, 100}]
```

Out[27]=



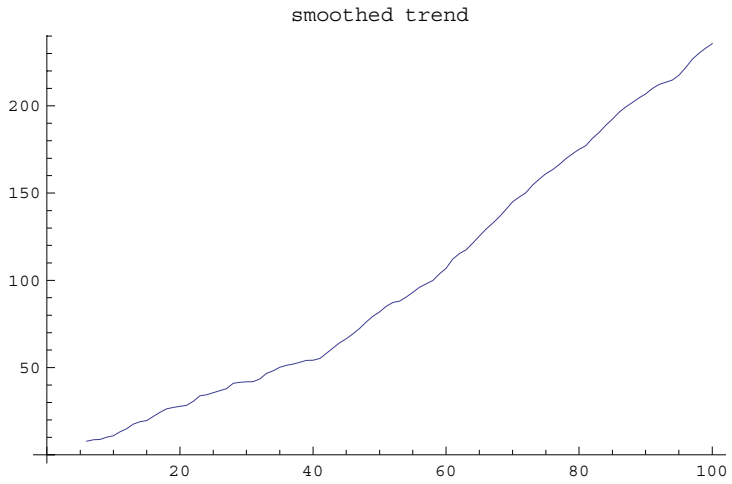
This gives the smoothed estimates of the state variables and their mean square errors.

```
In[28]:= ks = KalmanSmoothing[kf, F];
```

Here is the plot of the smoothed trend.

```
In[29]:= ListLinePlot[ks[[1, All, 1]], DataRange -> {6, 100}, PlotLabel -> "smoothed trend"]
```

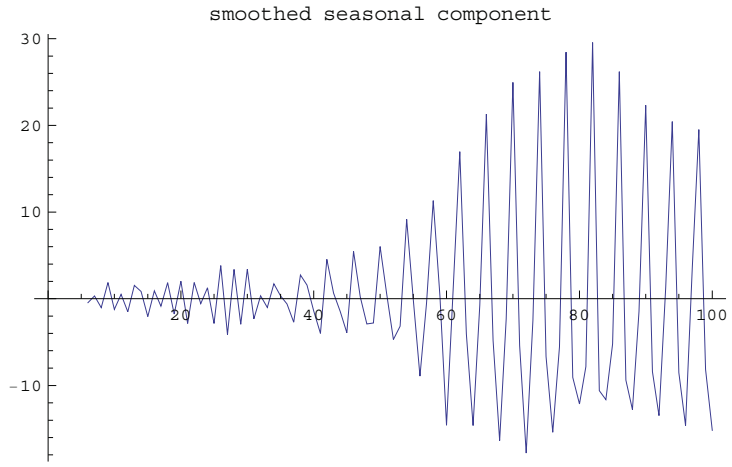
Out[29]=



This is the plot of the smoothed seasonal component.

```
In[30]:= ListLinePlot[ks[[1, All, 3]], DataRange -> {6, 100},
          PlotLabel -> "smoothed seasonal component"]
```

Out[30]=



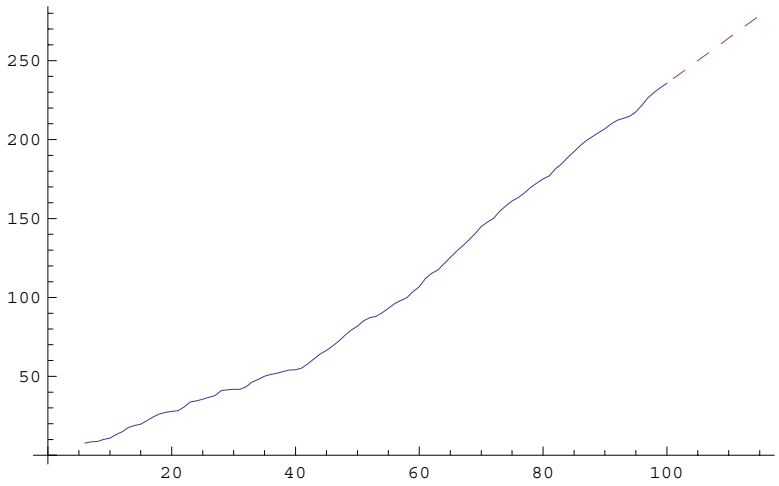
This gives the predicted values of the state variable for the next 15 points.

```
In[31]:= KalmanPredictor[Take[kf[[-1]], 2], F, Q, 15];
```

We plot the smoothed trend together with the next 15 predicted values of the trend.

```
In[32]:= ListLinePlot[{Transpose[{Range[6, 100], ks[[1, All, 1]]}],
          Transpose[{Range[101, 115], %[[1, All, 1]]}]},
         PlotStyle -> {{Thin}, {Dashing[{0.02}]}}]
```

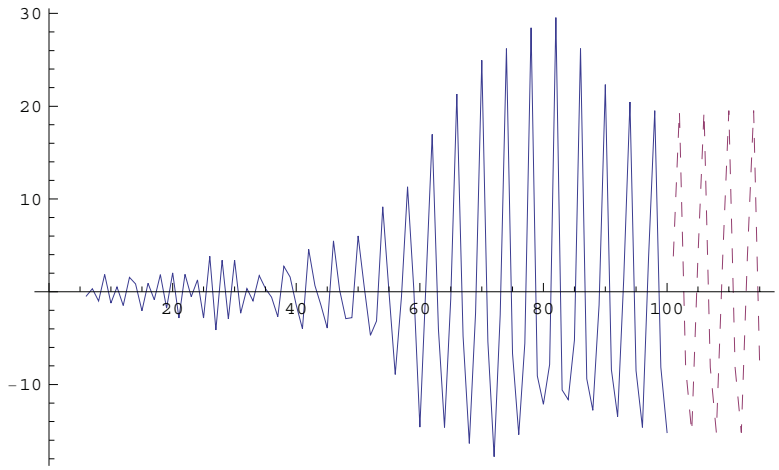
Out[32]=



Here the smoothed seasonal component is plotted together with the next 15 predicted values.

```
In[33]:= ListLinePlot[{Transpose[{Range[6, 100], ks[[1, All, 3]]}],
          Transpose[{Range[101, 115], %%[[1, All, 3]]}]},
         PlotStyle -> {{Thin}, {Dashing[{0.02}]}}]
```

Out[33]=



Parameter Estimation and Prediction

The likelihood function of a state-space time series can be easily calculated using the Kalman filter technique. The joint density of $\{Y_1, Y_2, \ldots, Y_T\}$ is $L = \prod_{t=1}^{T} p(Y_t \mid I_{t-1})$, where $p(Y_t \mid I_{t-1}) = N\left(\hat{Y}_{t|t-1}, f_{t|t-1}\right)$ and $f_{t|t-1} = E\left(Y_t - \hat{Y}_{t|t-1}\right)\left(Y_t - \hat{Y}_{t|t-1}\right)'$. The log likelihood is given by

$$\ln L = -\frac{1}{2}\sum_{t=1}^{T}\ln\left|f_{t|t-1}\right| - \frac{1}{2}\sum_{t=1}^{T}\left(Y_t - \hat{Y}_{t|t-1}\right)' f_{t|t-1}^{-1}\left(Y_t - \hat{Y}_{t|t-1}\right) + const.$$

Note that if the first m values of *data* are used to calculate the starting values, the lower limit of the summations is $t = m + 1$ and the corresponding $Log\, L$ is the conditional log likelihood, conditional on $y_1, y_2, \ldots, y_m$ being fixed.

If you wish to get the logarithm of the likelihood function of a state-space time series, you can use the function

`LogLikelihood[`*data*`,` *init*`,` *F*`,` *G*`,` *Q*`,` *R*`,` *c*`,` *d*`]`

Note that it has the same input arguments as those of `KalmanFilter`. When the first m points of the series are used to calculate the initial values *init*, *data* should start from y_{m+1}; when any one of F, G, Q, R, c, d is time dependent, the above arguments to `LogLikelihood`, F, G, Q, R, c, d, should be replaced by $\{F_{m+2}, F_{m+3}, \ldots, F_{T+1}\}$, $\{G_{m+1}, G_{m+2}, \ldots, G_T\}$, etc. Again, if $c = 0$ and $d = 0$, the last two arguments can be omitted.

Example 9.3 We again look at the local level model (see Example 9.1). This time we try to estimate the variance parameters of the model from the given series.

We generate a time series of length 300 according to the local level model (9.5).

```
In[34]:= (SeedRandom[34 561];
          eta = RandomSequence[0, 1, 300];
          mu = Accumulate[eta];
          data1 = RandomSequence[0, 0.5, 300] + mu;)
```

This gives the initial values. Note that the parameters to be estimated should be in symbolic form.

```
In[35]:= init = getInit[data1[[1]], 1, 1, q, r]

Out[35]= {0.00601132, q + r}
```

To get the maximum likelihood estimate of the parameters, we need to maximize the likelihood function. This is done using the built-in *Mathematica* function `FindMinimum`. Note that the function to be minimized is the negative of the log likelihood function. Also, we define a special function which evaluates only for numerical input to prevent `FindMinimum` from attempting symbolic preprocessing that can potentially take a long time.

```
In[36]:= Clear[f, if];
         if = Function[Evaluate[init /. {q → #1, r → #2}]];
         f[q_?NumericQ, r_?NumericQ] := -LogLikelihood[Rest[data1], if[q, r], 1, 1, q, r]

In[39]:= FindMinimum[f[q, r], {q, 0.8, 0.9}, {r, 0.4, 0.45}]

Out[39]= {241.189, {q → 0.740656, r → 0.676455}}

In[40]:= Clear[f, if];
```

Example 9.4 Here we give another example using the local linear trend model.

We generate a series from the local linear trend model.

```
In[41]:= (SeedRandom[4 563 456];
          eta = RandomSequence[0, 1, 100];
          zeta = RandomSequence[0, 0.05, 100];
          beta = Most[Accumulate[Flatten[{0.2, zeta}]]];
          mu = Accumulate[beta + eta];
          data2 = mu + RandomSequence[0, 0.5, 100];)
```

The initial values depend on the parameters $q1 = \sigma_\eta^2$, $q2 = \sigma_\zeta^2$, and $r = \sigma_\epsilon^2$, which are to be estimated.

```
In[42]:= init = getInit[Take[data2, 2], {{1, 1}, {0, 1}}, {{1, 0}}, {{q1, 0}, {0, q2}}, r]

Out[42]= {{3.26142, 1.19105}, {{2 q1 + q2 + 5 r, q1 + q2 + 3 r}, {q1 + q2 + 3 r, q1 + 2 q2 + 2 r}}}
```

This gives the estimate of the parameters.

```
In[43]:= Clear[f, if];
         if = Function[Evaluate[init /. {q1 → #1, q2 → #2, r → #3}]];
         f[q1_?NumericQ, q2_?NumericQ, r_?NumericQ] :=
          -LogLikelihood[Drop[data2, 2], if[q1, q2, r],
            {{1, 1}, {0, 1}}, {{1, 0}}, {{q1, 0}, {0, q2}}, r]

In[46]:= FindMinimum[f[q1, q2, r], {q1, 0.9, 1.0}, {q2, 0.03, 0.04}, {r, 0.3, 0.4}]

Out[46]= {87.4351, {q1 → 1.65597, q2 → 0.0153902, r → 0.154643}}

In[47]:= Clear[f, if];
```

Having obtained the estimated parameters, we can now use them to predict the future trend $\hat{\mu}_{t+h|t}$ by using `KalmanPredictor`. First, we calculate $\{X_{t+1|t}, P_{t+1|t}\}$ from `KalmanFilter`.

We substitute the estimated parameters.

```
In[48]:= {init, q1, q2, r} = {init, q1, q2, r} /. %%[[2]]

Out[48]= {{{3.26142, 1.19105}, {{4.10054, 2.13529}, {2.13529, 1.99604}}},
          1.65597, 0.0153902, 0.154643}
```

These are the values of $\{X_{t+1|t}, P_{t+1|t}\}$.

```
In[49]:= Take[KalmanFilter[Drop[data2, 2], init,
            {{1, 1}, {0, 1}}, {{1, 0}}, {{q1, 0}, {0, q2}}, r][[-1]], 2]

Out[49]= {{33.9048, 0.0714478}, {{1.99444, 0.181865}, {0.181865, 0.184168}}}
```

This gives the next 20 predicted values of X and their mean square errors.

```
In[50]:= KalmanPredictor[%, {{1, 1}, {0, 1}}, {{q1, 0}, {0, q2}}, 20];
```

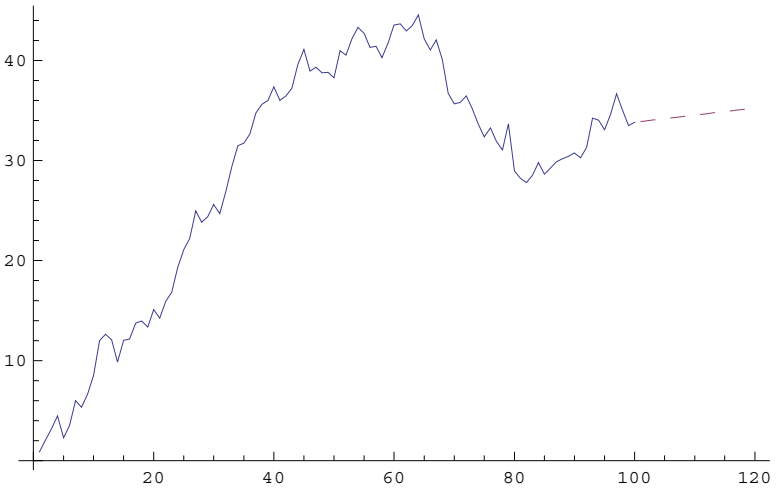
The predicted trend is given by the first component of X.

```
In[51]:= %[[1, All, 1]]

Out[51]= {33.9048, 33.9762, 34.0477, 34.1191, 34.1906, 34.262,
          34.3335, 34.4049, 34.4764, 34.5478, 34.6192, 34.6907, 34.7621,
          34.8336, 34.905, 34.9765, 35.0479, 35.1194, 35.1908, 35.2623}
```

This shows the predicted trend along with the given series.

```
In[52]:= ListLinePlot[{data2, Transpose[{Range[101, 120], %}]},
          PlotStyle -> {{Thin}, {Dashing[{0.02}]}}]
```

Out[52]=



Example 9.5 As a final example, we look at the AR(2) model and show that the Kalman filter approach gives the same estimation of parameters as the method introduced in the previous sections.

This generates an AR(2) series of length 50.

```
In[53]:= (SeedRandom[45 847];
          data3 = TimeSeries[ARModel[{0.9, -0.5}, 1], 50];)
```

This gives the initial $P_{1|0}$.

```
In[54]:= P0 = getP[{{p1, 1}, {p2, 0}}, {{q, 0}, {0, 0}}]
```

$$\text{Out[54]}= \left\{\left\{\frac{\left(1-\text{p2}^2\right)\,\text{q}}{1-\text{p1}^2-2\,\text{p1}^2\,\text{p2}-2\,\text{p2}^2-\text{p1}^2\,\text{p2}^2+\text{p2}^4},\ \frac{\left(\text{p1}\,\text{p2}+\text{p1}\,\text{p2}^2\right)\,\text{q}}{1-\text{p1}^2-2\,\text{p1}^2\,\text{p2}-2\,\text{p2}^2-\text{p1}^2\,\text{p2}^2+\text{p2}^4}\right\},\right.$$
$$\left.\left\{\frac{\left(\text{p1}\,\text{p2}+\text{p1}\,\text{p2}^2\right)\,\text{q}}{1-\text{p1}^2-2\,\text{p1}^2\,\text{p2}-2\,\text{p2}^2-\text{p1}^2\,\text{p2}^2+\text{p2}^4},\ \frac{\left(\text{p2}^2-\text{p2}^4\right)\,\text{q}}{1-\text{p1}^2-2\,\text{p1}^2\,\text{p2}-2\,\text{p2}^2-\text{p1}^2\,\text{p2}^2+\text{p2}^4}\right\}\right\}$$

`FindMinimum` finds the parameter values that maximize the likelihood function.

```
In[55]:= Clear[f, if];
         if = Function[Evaluate[P0 /. {p1 → #1, p2 → #2, q → #3}]];
         f[p1_?NumericQ, p2_?NumericQ, q_?NumericQ] :=
          -LogLikelihood[data3, {{0, 0}, if[p1, p2, q]},
            {{p1, 1}, {p2, 0}}, {{1, 0}}, {{q, 0}, {0, 0}}, 0]

In[58]:= FindMinimum[f[p1, p2, q], {p1, 0.8, 0.9}, {p2, -0.5, -0.55}, {q, 0.8, 0.9}]

Out[58]= {33.0088, {p1 → 0.898357, p2 → -0.567971, q → 1.34557}}
```

We can get the same result using `MLEstimate`. Note that for the one-dimensional ARMA model, the concentrated likelihood is calculated. Therefore, the variance q should not be included as a search parameter.

```
In[59]:= MLEstimate[data3, ARModel[{p1, p2}, q], {p1, {0.8, 0.9}}, {p2, {-0.5, -0.55}}]

Out[59]= ARModel[{0.89833, -0.568009}, 1.34557]
```

Note that the value of the log likelihood is different from that obtained using the Kalman filter. This is because the former calculates the concentrated likelihood whereas the latter, the full likelihood (apart from a constant).

```
In[60]:= LogLikelihood[data3, %]

Out[60]= -0.320353
```

On the other hand, had we used the first two data points to calculate the initial values, we would have effectively obtained the conditional maximum likelihood estimate, conditional on the first two data points being fixed.

The initial values obtained this way correspond to using a diffuse prior.

```
In[61]:= (Clear[p1, p2, q];
          getInit[Take[data3, 2], {{p1, 1}, {p2, 0}}, {{1, 0}}, {{q, 0}, {0, 0}}, 0])

Out[61]= {{2.33614 p1 + 1.42358 p2, 2.33614 p2}, {{q, 0}, {0, 0}}}
```

This gives the estimated parameters.

```
In[62]:= Clear[f, if];
         if = Function[Evaluate[%% /. {p1 → #1, p2 → #2, q → #3}]];
         f[p1_?NumericQ, p2_?NumericQ, q_?NumericQ] :=
          -LogLikelihood[Drop[data3, 2], if[p1, p2, q],
            {{p1, 1}, {p2, 0}}, {{1, 0}}, {{q, 0}, {0, 0}}, 0]
```

```
In[65]:= FindMinimum[f[p1, p2, q], {p1, 0.8, 0.9}, {p2, -0.5, -0.55}, {q, 0.8, 0.9}]

Out[65]= {31.1975, {p1 → 0.894475, p2 → -0.569652, q → 1.34971}}

In[66]:= Clear[f, if]
```

We can get the same result much faster using `ConditionalMLEstimate`.

```
In[67]:= ConditionalMLEstimate[data3, 2]

Out[67]= ARModel[{0.894475, -0.569652}, 1.34972]
```

1.10 Univariate ARCH and GARCH Models

In the time series we have considered so far, the disturbances or errors $\{Z_t\}$ are assumed to be *homoskedastic*, that is, the variance of Z_t is assumed to be independent of t. *Autoregressive Conditional Heteroskedasticity* (ARCH) models and *Generalized Autoregressive Conditional Heteroskedasticity* (GARCH) models are used to model changes in the variance of the errors as a function of time. An ARCH process of order q, ARCH(q), is given by (see Engle (1982))

$$Z_t = \nu_t \sqrt{h_t} \tag{10.1}$$

where $\{\nu_t\}$ is an independently distributed Gaussian random sequence with zero mean and unit variance; h_t is the conditional variance of Z_t conditional on all the information up to time $t-1$, I_{t-1}:

$$E\left(Z_t^2 \mid I_{t-1}\right) = h_t = \alpha_0 + \alpha_1 z_{t-1}^2 + \alpha_2 z_{t-2}^2 + \cdots + \alpha_q z_{t-q}^2. \tag{10.2}$$

GARCH models are generalizations of ARCH models where h_t, the conditional variance at time t, depends on earlier variances. That is, a GARCH(p, q) process is given by (10.1) with (see Bollerslev (1986))

$$h_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i z_{t-i}^2 + \sum_{i=1}^{p} \beta_i h_{t-i}. \tag{10.3}$$

When $p = 0$ we have an ARCH(q) model; when both p and q are zero, Z_t is simply white noise.

An ARCH(q) model and a GARCH(p, q) model are represented in this package by

```
ARCHModel[{α0, α1, …, αq}]
```

and

```
GARCHModel[{α0, α1, …, αq}, {β1, β2, …, βp}],
```

respectively. Note that since the variance is positive, we usually have $\alpha_0 > 0$, $\alpha_i \geq 0$, and $\beta_i \geq 0$ for $i > 0$.

To generate an ARCH or a GARCH process, we can again use the function `TimeSeries`.

Example 10.1 Generate a time series of length 10 according to the ARCH(2) model $z_t = \nu_t \sqrt{h_t}$ where $h_t = 0.05 + 0.2\, z_{t-1}^2 + 0.5\, z_{t-2}^2$.

We load the package first.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

The random number generator is seeded.

```
In[2]:= SeedRandom[23 458];
```

This generates a time series of length 10 according to the given ARCH(2) model.

```
In[3]:= data = TimeSeries[ARCHModel[{0.05, 0.2, 0.5}], 10]

Out[3]= {0.429864, 0.352351, -0.0516678, 0.288368, -0.0652888,
         -0.180396, -0.298827, -0.140906, 0.102956, -0.272234}
```

Note that the initial values of z_t ($t = -q + 1, \ldots, -1, 0$) are needed to generate a time series from an ARCH(q) model. As in the case of generating a time series from an ARMA model, a third argument can be given to `TimeSeries` to specify the initial values $\{z_{-q+1}, z_{-q+2}, \ldots, z_0\}$ (or $\{\{h_{-p+1}, h_{-p+2}, \ldots, h_0\}, \{z_{-q+1}, z_{-q+2}, \ldots, z_0\}\}$ in the case of a GARCH model). If the third argument is omitted, the initial values are all set to zero as was tacitly done in Example 10.1.

Example 10.2 Generate a time series of length 10 according to the GARCH(1, 1) model with $\alpha_0 = 0.05$, $\alpha_1 = 0.5$, and $\beta_1 = 0.32$ and with the initial values $z_0 = 0.1 \; a \; n \; d \; h_0 = 0.3$.

This generates a time series of length 10 according to the given GARCH(1, 1) model.

```
In[4]:= (SeedRandom[23 458];
         data = TimeSeries[GARCHModel[{0.05, 0.5}, {0.32}], 10, {{0.3}, {0.1}}])

Out[4]= {0.747024, 0.733993, -0.0838224, 0.378839,
         -0.10741, -0.200897, -0.4034, -0.197666, 0.114754, -0.338252}
```

The so-called ARCH- or GARCH-regression model is a regression model with the disturbances following an ARCH process (see (10.1) and (10.2)) or a GARCH process (see (10.1) and (10.3)), respectively. That is,

$$Y_t = \mathbf{x}_t' \cdot \mathbf{b} + Z_t \tag{10.4}$$

where $\mathbf{x_t}$ is a known column vector that can contain lagged values of Y (i.e., y_{t-1}, y_{t-2}, … etc.), and $\mathbf{b}$ is a column vector of unknown parameters. The first term on the right-hand side of (10.4) is the conditional mean of Y_t; that is, $E(Y_t \mid I_{t-1}) = \mathbf{x}_t' \cdot \mathbf{b} \equiv m_t$, and the conditional variance of Y_t is that of Z_t and is given by (10.2) or (10.3).

An AR-ARCH model has $\{Y_t\}$ following an AR process and the disturbances $\{Z_t\}$ following an ARCH process. It is easy to see that in this case, $\mathbf{x}_t' = \{y_{t-1}, y_{t-2}, \ldots, y_{t-p}\}$ and $\mathbf{b}' = \{\phi_1, \phi_2, \ldots, \phi_p\}$. If the AR process has a constant term (i.e., nonzero mean), then we have $\mathbf{x}_t' = \{1, y_{t-1}, y_{t-2}, \ldots, y_{t-p}\}$ and $\mathbf{b}' = \{\mu, \phi_1, \phi_2, \ldots, \phi_p\}$.

Example 10.3 Generate a time series of length 100 according to the AR(2)-ARCH(1, 1) model with $\phi_1 = 0.2$, $\phi_2 = -0.5$, $\alpha_0 = 0.06$, $\alpha_1 = 0.4$, and $\beta_1 = 0.7$.

This series is generated from the given GARCH(1, 1) model. This will be used as the disturbances to generate the AR(2) series below.

```
In[5]:= (SeedRandom[4 584 715];
         z = TimeSeries[GARCHModel[{0.06, 0.4}, {0.7}], 100];)
```

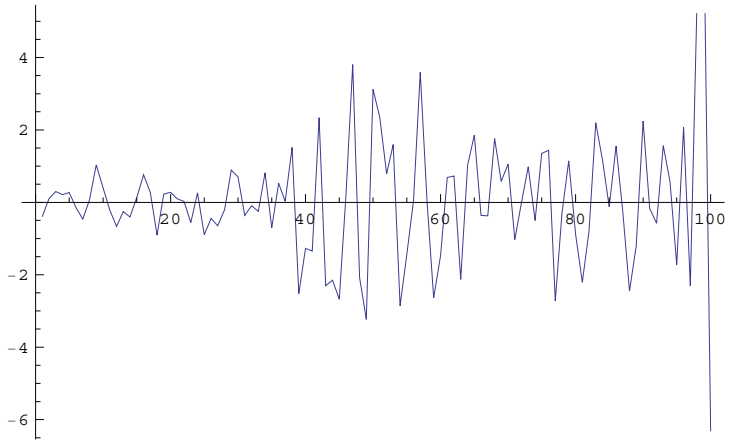
This generates the AR(2) time series with the given GARCH disturbances. Note that the last entry {0, 0} contains the initial values of y, $\{y_{-1}, y_0\}$.

```
In[6]:= data = TimeSeries[ARModel[{0.2, -0.5}], z, {0, 0}];
```

The series is plotted here.

```
In[7]:= ListLinePlot[data]
```

Out[7] =



1.10.1 Estimation of ARCH and GARCH Models

From the definition of the ARCH(q) model, it is clear that the correlation $E(Z_t Z_{t+k})$ ($k \neq 0$) is zero. However, it is easy to see that Z_t^2 follows an AR(q) process. Similarly, if $\{Z_t\}$ is a GARCH(p, q) process, Z_t^2 follows an ARMA(s, p) process, where $s = \text{Max}(p, q)$. This can be used to help identify the orders of ARCH or GARCH models.

The maximum likelihood method is often used to estimate the parameters of an ARCH or GARCH model. The logarithm of the Gaussian likelihood function is given by (apart from an additive constant)

$$\sum_{t=1}^{N}\left(-\frac{1}{2}\ln h_t - \frac{z_t^2}{2h_t}\right), \tag{10.5}$$

where $z_t (= y_t - m_t)$ conditional on I_{t-1} is normally distributed with zero mean and variance h_t. The function

`LogLikelihood[z, model]`

gives the logarithm of the likelihood (10.5), where *model* can be `ARCHModel` or `GARCHModel` and $z = \{z_1, z_2, \ldots, z_N\}$. Note that the presample values of $\{z^2_{-q+1}, \ldots, z^2_{-1}, z^2_0\}$ and $\{h_{-p+1}, \ldots, h_{-1}, h_0\}$ are assumed to be equal to a fixed value *sigma2*, and it can be specified using the option `PresampleValue -> sigma2`. The default setting for `PresampleValue` is `Automatic`, which corresponds to using the sample equivalence of σ^2,

$$sigma2 = \frac{\sum_{t=1}^{N} z_t^2}{N} - \left(\frac{\sum_{t=1}^{N} z_t}{N}\right)^2. \tag{10.6}$$

For example, this calculates the logarithm of the likelihood function for GARCH series `z` given in Example 10.3.

```
In[8]:= LogLikelihood[z, GARCHModel[{0.06, 0.4}, {0.7}]]

Out[8]= -71.0145
```

To calculate the likelihood for series with nonzero conditional mean ($m_t \neq 0$), we can subtract the mean from the data before using the function `LogLikelihood`, or we can use

`LogLikelihood[`*data*`,` *model*`,` *X*`,` *blist*`]`

where X is the matrix in the regression model (10.4) and *blist* is the list of parameters b. If the regression model is an AR model, we can use

`LogLikelihood[`*data*`,` *model*`,` `ARModel[`*philist*`]]`

or

`LogLikelihood[`*data*`,` *model*`,` `ARModel[`*philist*`]` `,` μ`]`

when the AR model has nonzero mean *mu*. For example, when *data* $= \{y_t\}$ is an AR-ARCH (or AR-GARCH) series, the conditional mean vector is $X \cdot \{\phi_1, \ldots, \phi_p\}$, where the matrix $X = \{\mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_N\}$ and $\mathbf{x}'_t = \{y_{t-1}, y_{t-2}, \ldots, y_{t-p}\}$. The following program allows us to get the matrix X given the values $y = \{y_t\}$ and p.

This gives the matrix X defined above.

```
In[9]:= getX[y_List, p_] := Most[Drop[Transpose[NestList[RotateRight, y, p - 1]], p - 1]]
```

Note that if the input y in `getX` is $\{y_1, y_2, \ldots, y_N\}$, we get $X = \{\mathbf{x}'_{p+1}, \mathbf{x}'_{p+2}, \ldots, \mathbf{x}'_N\}$. So we should drop the first p points from the series to get $y_t - \mathbf{x}'_t \cdot \mathbf{b}$ for $t = p + 1, \ldots, N$. (That is, the series now has length $N - p$.) On the other hand, if we want to keep all N data points in our series, we need to prepend p initial points $\{y_{-p+1}, \ldots, y_{-1}, y_0\}$ to y before getting the matrix X. The following example shows the difference.

This gives the logarithm of the likelihood function of the AR-GARCH series *data*. Note that the difference between this value and that for the series z arises because here we drop the first 2 points of the series.

```
In[10]:= (X = getX[data, 2];
          LogLikelihood[Drop[data, 2] - X.{0.2, -0.5}, GARCHModel[{0.06, 0.4}, {0.7}]])

Out[10]= -71.7491
```

Here we include $\{y_{-1}, y_0\} = \{0, 0\}$, and the likelihood is calculated for the series of length N.

```
In[11]:= (X = getX[Join[{0, 0}, data], 2];
          LogLikelihood[data - X.{0.2, -0.5}, GARCHModel[{0.06, 0.4}, {0.7}]])

Out[11]= -71.0145
```

You can, of course, also obtain the log likelihood directly.

```
In[12]:= LogLikelihood[data, GARCHModel[{0.06, 0.4}, {0.7}], ARModel[{0.2, -0.5}]]

Out[12]= -71.7491
```

Once the likelihood function is calculated, we can in principle estimate the model parameters by maximizing the likelihood function (or minimizing its negative). This can be accomplished in some cases by using the built-in function `FindMinimum`. For example, to find the GARCH parameters from the series z, we can do the following.

This gives the maximum likelihood estimate of the GARCH parameters. Note that two initial values are required for each parameter in the search. The result depends in general on the presample values. The default presample values are used here.

```
In[13]:= FindMinimum[-LogLikelihood[z, GARCHModel[{α0, α1}, {β1}]],
          {α0, 0.1, 0.12}, {α1, 0.3, 0.4}, {β1, 0.6, 0.8}]

Out[13]= {69.7163, {α0 → 0.0408018, α1 → 0.499703, β1 → 0.61434}}
```

Using `FindMinimum` we can easily incorporate the constraints imposed on the parameters. For example, in the following, we impose the constraints, $\alpha_2 = \frac{2}{3}\alpha_1, \alpha_3 = \frac{1}{3}\alpha_1$.

This generates an ARCH series with $\alpha_1 = 0.6, \alpha_2 = 2/3\,\alpha_1, \alpha_3 = 1/3\,\alpha_1$.

```
In[14]:= (SeedRandom[4587];
          α1 = 0.6;
          data1 = TimeSeries[ARCHModel[{0.05, α1, 2 / 3 α1, 1 / 3 α1}], 100];)
```

The constraints are put in explicitly and there are only two parameters to estimate.

```
In[15]:= FindMinimum[-LogLikelihood[data1, ARCHModel[{α0, α1, 2 / 3 α1, 1 / 3 α1}]],
          {α0, 0.05, 0.12}, {α1, 0.4, 0.5}]

Out[15]= {-12.3155, {α0 → 0.0590563, 0.6 → 0.527295}}
```

However, when the number of parameters is large, the function `FindMinimum` can either be very slow or can go into parameter regions where the `LogLikelihood` function is complex. In these cases, the function `ConditionalMLEstimate` should be used to estimate ARCH or GARCH parameters.

`ConditionalMLEstimate[`*data*`, `*model*`]`

fits the specified model to *data* using the maximum likelihood method, where *model* can be `ARCHModel` or `GARCHModel`. The presample values of z^2 and h are fixed (or conditioned upon) and the BHHH algorithm (see Berndt et al. (1974)) is used in the numerical search for the maximum of the likelihood function. The starting values of the parameters are given inside *model*. Note that the output of `ConditionalMLEstimate` is a list

containing two parts. The first part is the estimated model, and the second is the root mean square errors given by the last iteration of the BHHH algorithm.

Note that the result is a list containing two parts. The first part is the estimated model, and the second is the root mean square errors given by the last iteration of the BHHH algorithm.

```
In[16]:= model1 = ConditionalMLEstimate[z, GARCHModel[{0.1, 0.1}, {0.9}]]

Out[16]= {GARCHModel[{0.0409048, 0.49959}, {0.614326}], {{0.0544903, 0.161987}, {0.0978399}}}
```

When estimating the parameters of the ARCH and GARCH models using `ConditionalMLEstimate`, we must be cautious about the following points. First, the maximum can depend on the initial conditions, and there is no guarantee that the maximum obtained using a particular set of initial parameter values is the global maximum. Second, since the estimate is conditioned on the presample values, the result can depend on the presample values. The following examples illustrate these dependencies.

Comparing with the above result `model1`, we see that different starting parameter values can give rise to different estimated model parameters. Note that $\beta_1 < 0$ in this estimate.

```
In[17]:= model2 = ConditionalMLEstimate[z, GARCHModel[{0.5, 0.05}, {0.1}]]

Out[17]= {GARCHModel[{0.0406652, 0.49987}, {0.614364}], {{0.0548145, 0.161957}, {0.0980003}}}
```

This shows that `model2` is not a global maximum.

```
In[18]:= {LogLikelihood[z, model1[[1]]], LogLikelihood[z, model2[[1]]]}

Out[18]= {-69.7163, -69.7163}
```

Here the presample values are set to zero through the option `PresampleValue`. Since the likelihood function depends on the presample values, so does its maximum value in general.

```
In[19]:= ConditionalMLEstimate[z, GARCHModel[{0.1, 0.1}, {0.9}], PresampleValue -> 0]

Out[19]= {GARCHModel[{0.042347, 0.385326}, {0.696215}], {{0.0249406, 0.135533}, {0.0999806}}}
```

To estimate an ARCH-regression (or GARCH-regression) model (10.4), we can use

`ConditionalMLEstimate[`*data*`,` *model*`,` X`,` *blist*`]`,

where *model* is either `ARCHModel` or `GARCHModel`, X is the matrix in the regression model (10.4), and *blist* is the list of initial parameter values **b**. If the regression model is an AR model, we can also use

`ConditionalMLEstimate[`*data*`,` *model*`, ARModel[`*philist*`]]`

or

`ConditionalMLEstimate[`*data*`,` *model*`, ARModel[`*philist*`],` μ`]`

when the AR model has a nonzero mean μ. The following example illustrates how to estimate the parameters of an AR-GARCH model using `ConditionalMLEstimate`.

This generates the AR(2) time series with the given GARCH disturbances. Note that the last entry {0, 0} contains the initial values of y, $\{y_{-1}, y_0\}$.

```
In[20]:= (SeedRandom[4 584 715];
          z = TimeSeries[GARCHModel[{0.06, 0.4}, {0.7}], 100];
          data = TimeSeries[ARModel[{0.2, -0.5}], z, {0, 0}];)
```

Note that the corresponding root mean square error is included in the second part of the output.

```
In[21]:= ConditionalMLEstimate[data, GARCHModel[{0.1, 0.1}, {0.9}], ARModel[{0.1, -0.4}]]

Out[21]= {{GARCHModel[{0.0497434, 0.489292}, {0.608463}], ARModel[{0.0174107, -0.413086}]},
          {{{0.062788, 0.169309}, {0.119431}}, {0.120872, 0.112056}}}
```

When using `ConditionalMLEstimate` to estimate the model parameters, the presample values are fixed and do not change during the search for the maximum. When using the default presample value given by (10.6), the presample values can be a function of initial parameters. Therefore, different initial parameter values can affect the result of maximization. We can iterate the maximization process to get a consistent result.

When we change the initial parameter values, the result also changes. A convergent result can be obtained by iteration.

```
In[22]:= ConditionalMLEstimate[data, %[[1, 1]], %[[1, 2]]]

Out[22]= {{GARCHModel[{0.0493484, 0.487968}, {0.60972}], ARModel[{0.0178315, -0.413022}]},
          {{{0.0626363, 0.168979}, {0.119539}}, {0.120956, 0.112103}}}
```

However, the presample values are not fixed when using `FindMinimum`. It changes during the search for optimal parameters.

```
In[23]:= FindMinimum[
          -LogLikelihood[Drop[data, 2] - getX[data, 2].{ϕ1, ϕ2}, GARCHModel[{α0, α1}, {β1}]],
          {α0, 0.05, 0.04}, {α1, 0.1, 0.12},
          {β1, 0.85, 0.9}, {ϕ1, 0.2, 0.25}, {ϕ2, -0.4, -0.45}]

Out[23]= {68.7031, {α0 → 0.0497801, 0.6 → 0.48895, β1 → 0.608726, ϕ1 → 0.0154191, ϕ2 → -0.41687}}
```

1.10.2 ARCH-in-Mean Models

The ARCH-regression model can be generalized to include the cases where the conditional mean m_t is a function of h_t. This is the so-called ARCH-in-mean model (see Engle, Lilien, and Robins (1987) and Domowitz and Hakkio (1985)). For example, for an ARCH-in-mean (or GARCH-in-mean) regression model, (10.4) is generalized to

$$Y_t = \mathbf{x}_t'.\mathbf{b} + \delta f(h_t) + Z_t \tag{10.7}$$

where h_t is given by (10.2) (or (10.3)) and $f(h)$ is usually $\sqrt{h}$ or h. (Other functions like $\ln h$ are also possible.) The ARCH-in-mean or GARCH-in-mean models are represented by

`ARCHModel[`*alphalist*`, `δ`, `*f*`]`

or

`GARCHModel[`*alphalist*`, `*betalist*`, `δ`, `*f*`]`,

respectively. Note that the function f should be input as a symbol representing a built-in function (e.g., `Sqrt`, `Identity`, `Log`, represent $f(h) = \sqrt{h}$, $f(h) = h$, and $f(h) = \ln h$, respectively) or as a pure function (e.g., `Sqrt[#]&`, etc.). The functions introduced in the previous sections can all be used for ARCH-in-mean models. However, the default value for `PresampleValue` is $\alpha_0 / \left(1 - \sum_{i=1}^{q} \alpha_i - \sum_{i=1}^{p} \beta_i\right)$ if the model is ARCH-in-mean or GARCH-in-mean with $q \neq 0$ and $\alpha_0 / \left(1 - \sum_{i=1}^{p} \beta_i\right) + 0.001$ if $q = 0$. (Note the small number 0.001 in the case of $q = 0$. It is added to prevent a constant h_t that would have resulted from using $\alpha_0 / \left(1 - \sum_{i=1}^{p} \beta_i\right)$ as the presample value.)

Example 10.4 Generate a time series of length 150 according to the given ARCH-in-mean model `model`; then estimate its parameters using `ConditionalMLEstimate`.

This is an ARCH-in-mean model.

```
In[24]:= model = ARCHModel[{0.05, 0.3}, 0.5, Sqrt]

Out[24]= ARCHModel[{0.05, 0.3}, 0.5, Sqrt]
```

This generates the time series of length 150 with the given model.

```
In[25]:= (SeedRandom[38 470];
          data1 = TimeSeries[model, 150];)
```

This gives estimated model parameters. Again, the second part of the output is the root mean square errors of the estimates.

```
In[26]:= ConditionalMLEstimate[data1, ARCHModel[{0.01, 0.1}, 0.7, Sqrt]]

Out[26]= {ARCHModel[{0.039909, 0.550819}, 0.577203, Sqrt], {{0.00707764, 0.1902}, 0.0946216}}
```

Note that if there is a constant in the regression or AR model, the constant term and δ may not be distinguished as the following example shows.

This generates a GARCH-in-mean series.

```
In[27]:= (SeedRandom[345 763];
          TimeSeries[GARCHModel[{0.05, 0.3}, {0.4}, 0.5, Sqrt], 150];)
```

This generates an AR-GARCH model with a nonzero mean.

```
In[28]:= data = TimeSeries[ARModel[{0.5}], %, {0}] + 0.8;
```

We see that the estimate for the constant term and δ are significantly off. The root mean square error for δ is very large.

```
In[29]:= ConditionalMLEstimate[data,
          GARCHModel[{0.03, 0.2}, {0.2}, 0.5, Sqrt], ARModel[{0.3}], 0.5]

Out[29]= {{GARCHModel[{0.0420259, 0.492268}, {0.290676}, 0.39608, Sqrt], ARModel[{0.465656}],
           0.497084}, {{{0.0162677, 0.155282}, {0.155344}, 0.324422}, {0.0715291}, 0.118398}}
```

As we have said earlier, the result can depend on the presample values that are specified in the beginning; in this case, the presample values are determined by the initial parameter values. We can iterate the estimation to make sure that the result has converged.

This shows that the result has converged.

```
In[30]:= ConditionalMLEstimate[data, Sequence @@ %[[1]]]

Out[30]= {{GARCHModel[{0.0438585, 0.499831}, {0.268788}, 0.32215, Sqrt], ARModel[{0.463658}],
           0.526102}, {{{0.0167441, 0.159663}, {0.160427}, 0.313867}, {0.0711064}, 0.118476}}
```

This gives the log likelihood.

```
In[31]:= LogLikelihood[data, Sequence @@ %[[1]]]

Out[31]= 78.4871
```

1.10.3 Testing for ARCH

Various standard procedures are available to test the existence of ARCH or GARCH. A commonly used test is the Lagrange multiplier (LM) test. Consider the null hypothesis that there is no ARCH, that is, $\alpha_1 = \alpha_2 = \cdots = \alpha_q = 0$. It is known that (see, for example, Bollerslev (1986), Eqs. (27) and (28)) the LM statistic has an asymptotic χ^2 distribution with q degrees of freedom under the null hypothesis. If the LM statistic evaluated under the null hypothesis is greater than $\chi^2_{1-\alpha}(q)$, the null hypothesis is rejected at level α. The function

LMStatistic[*data*, *model*]

gives the LM statistic, where `model` is either an ARCH or GARCH model. For an ARCH (or GARCH) regression model (10.7), we can use

LMStatistic[*data*, *model*, *X*, *blist*],

or if the regression model is an AR model,

LMStatistic[*data*, *model*, ARModel[*philist*]],

or

LMStatistic[*data*, *model*, ARModel[*philist*], μ],

when the AR model has nonzero mean *mu*.

For example, in Example 10.4 we estimated the ARCH-in-mean model with $q = 1$. We can test the hypothesis that both conditional mean and variance are in fact constant by testing the null hypothesis $q = 0$. In the Lagrange multiplier test, the parameters used are those obtained under the null hypothesis. So we first estimate the model parameters with $q = 0$.

This gives the estimated model parameters under the null hypothesis.

```
In[32]:= ConditionalMLEstimate[data1, ARCHModel[{0.01}, 0.7, Sqrt]]

Out[32]= {ARCHModel[{0.0895768}, 0.54372, Sqrt], {{0.0105124}, 0.10468}}
```

The alternative hypothesis is $q = 1$. This is the estimated ARCH-in-mean(1) model obtained under the null hypothesis.

```
In[33]:= modelnull = MapAt[Append[#, 0] &, %[[1]], {1}]

Out[33]= ARCHModel[{0.0895768, 0}, 0.54372, Sqrt]
```

Note that under the null hypothesis the series is simply white noise with a mean of $\delta\sqrt{\alpha_0}$ and variance of α_0. Therefore, we can estimate α_0 and δ simply using the sample estimates.

This gives the sample mean.

```
In[34]:= Mean[data1]

Out[34]= 0.162833
```

This is the sample variance. It is the estimated α_0.

```
In[35]:= CovarianceFunction[data1, 0]

Out[35]= {0.0896269}
```

This gives the estimated δ.

```
In[36]:= %% / Sqrt[%]

Out[36]= {0.543906}
```

The LM statistic is obtained using the model estimated under the null hypothesis.

```
In[37]:= lms = LMStatistic[data1, modelnull]

Out[37]= 37.3643
```

The number $\chi^2_{0.95}(1)$ is much smaller than the above `lms`. So we can reject the null hypothesis of $q = 0$.

```
In[38]:= Quantile[ChiSquareDistribution[1], 0.95]

Out[38]= 3.84146
```

Here is another example. Again we first generate an ARCH(3) model.

```
In[39]:= (SeedRandom[4587];
          data = TimeSeries[ARCHModel[{0.05, 0.6, 0.4, 0.2}], 100];)
```

The null hypothesis is an ARCH(1) model and the parameters are estimated under the null hypothesis.

```
In[40]:= ConditionalMLEstimate[data, ARCHModel[{0.2, 0.4}]]

Out[40]= {ARCHModel[{0.104522, 0.858552}], {0.0280633, 0.270444}}
```

The alternative hypothesis is ARCH(3); here we compute the LM statistic for this case.

```
In[41]:= LMStatistic[data, ARCHModel[Join[%[[1, 1]], {0, 0}]]]

Out[41]= 5.32774
```

The above LM statistic does not exceed this number by a large amount. So the null hypothesis of ARCH(1) is not rejected.

```
In[42]:= Quantile[ChiSquareDistribution[2], 0.95]

Out[42]= 5.99146
```

Here we generate a GARCH(2,1) series.

```
In[43]:= (SeedRandom[563 651];
          data = TimeSeries[GARCHModel[{0.05, 0.3, 0.5}, {0.2}], 100];)
```

The null hypothesis is AR(1) and this estimates its parameters.

```
In[44]:= ConditionalMLEstimate[data, ARCHModel[{0.1, 0.5}]][[1]]

Out[44]= ARCHModel[{0.355812, -0.0545774}]
```

In this case there is evidence to reject the null hypothesis.

```
In[45]:= {LMStatistic[data, GARCHModel[Join[%[[1]], {0}], {0}]],
          Quantile[ChiSquareDistribution[2], 0.95]}

Out[45]= {6.01793, 5.99146}
```

Note that it is known that a general test for GARCH(p, q) is not feasible (see Bollerslev (1986)).

1.11 Examples of Analysis of Time Series

We illustrate the procedures and *Mathematica* functions outlined in earlier sections by considering a few examples of time series analysis. The reader should keep in mind that in each of the following examples the interpretation of the data and the model fitted are by no means unique; often there can be several equally valid interpretations consistent with the data. No package can be a substitute for experience and good judgment.

Example 11.1 The file `csp.dat` contains the common stock prices from 1871 to 1970. We read in the data using `ReadList`.

We load the package first.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

This reads in the data.

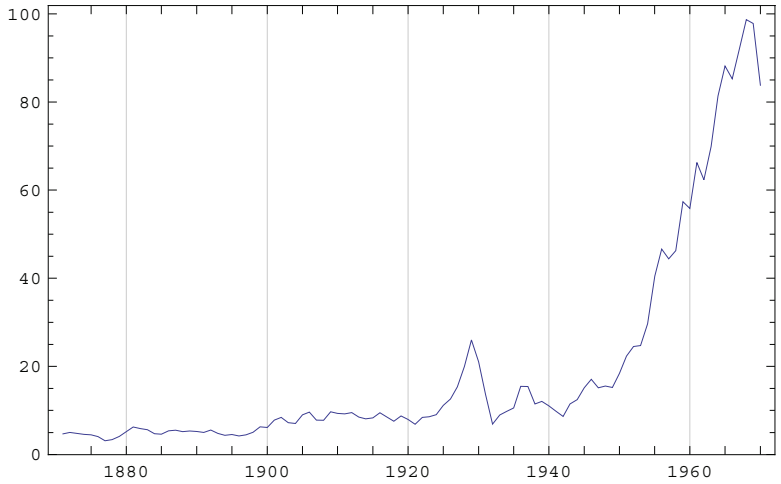
```
In[2]:= csp = ReadList[ToFileName[{"TimeSeries", "Data"}, "csp.dat"], Number];
```

As we have emphasized earlier, the first thing to do in analyzing a time series is to look at its time plot.

Here is a plot of the data.

```
In[3]:= DateListPlot[csp, {{1871}, {1970}},
         PlotRange -> All, Joined -> True, AxesLabel -> {"t", "xt"}]
```

Out[3]=

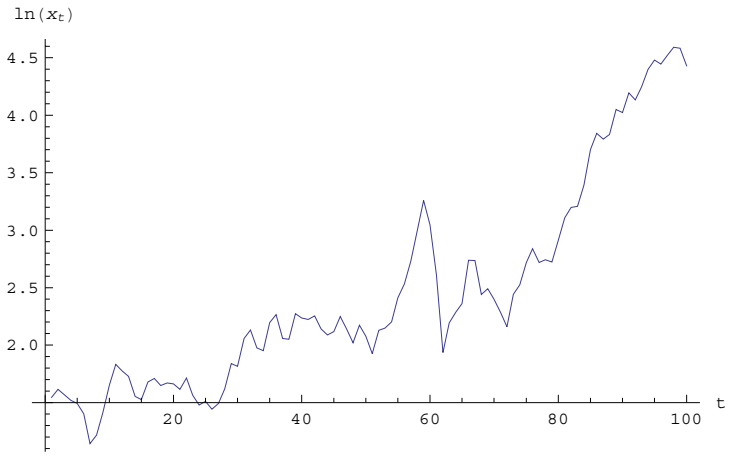


We see that the series rises sharply with time, clearly signaling the nonstationary nature of the series, and indicating the need to transform the data to make it stationary. Here is the time plot of the series after the logarithmic transformation.

This plots the transformed data.

```
In[4]:= ListLinePlot[Log[csp], AxesLabel -> {"t", "ln(x_t)"}]
```

Out[4]=



The variability in the series has been reduced. However, a rising trend still persists. We difference the series to obtain a constant mean series.

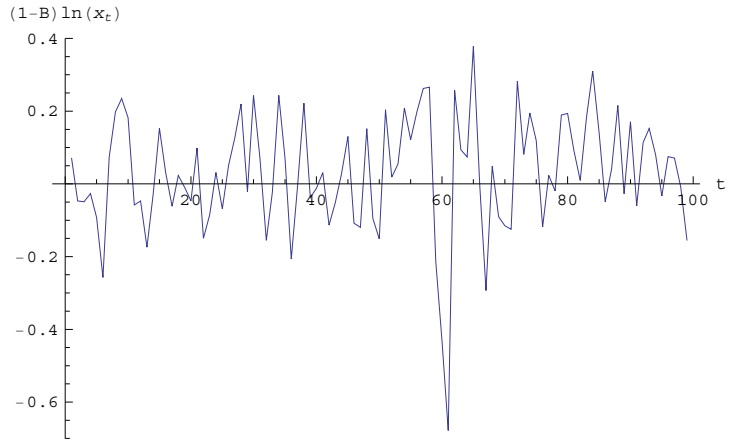
`data` contains the differenced series.

```
In[5]:= data = ListDifference[Log[csp], 1];
```

This is the plot of `data`.

```
In[6]:= ListLinePlot[data, AxesLabel -> {"t", "(1-B)ln(x_t)"}]
```

Out[6]=



Now that the mean appears steady, we assume the series has a constant mean (the reader is encouraged to devise tests to check this) and evaluate it.

This gives the mean.

```
In[7]:= Mean[data]

Out[7]= 0.0291236
```

We subtract this sample mean from the series to make it a zero-mean series.

The data are transformed to a zero-mean series.

```
In[8]:= data = data - Mean[data];
```

The length of the data is n.

```
In[9]:= n = Length[data]

Out[9]= 99
```

Now we can fit `data` to a stationary time series model. In order to select an appropriate model, we first look at its correlation and partial correlation functions.

This gives the sample correlation function.

```
In[10]:= corr = CorrelationFunction[data, 20];
```

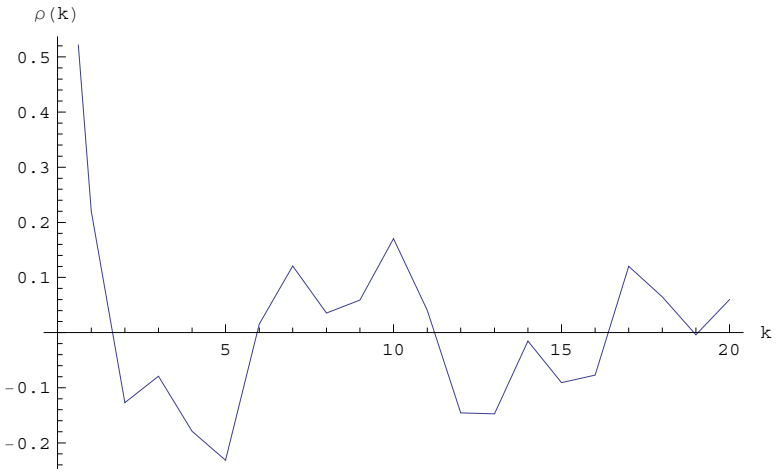
To plot the correlation function, we redefine the function `plotcorr` here.

```
In[11]:= plotcorr[corr_, opts___] := ListPlot[corr, DataRange -> {0, Length[corr] - 1}, opts]
```

This is the correlation plot.

```
In[12]:= plotcorr[corr, Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

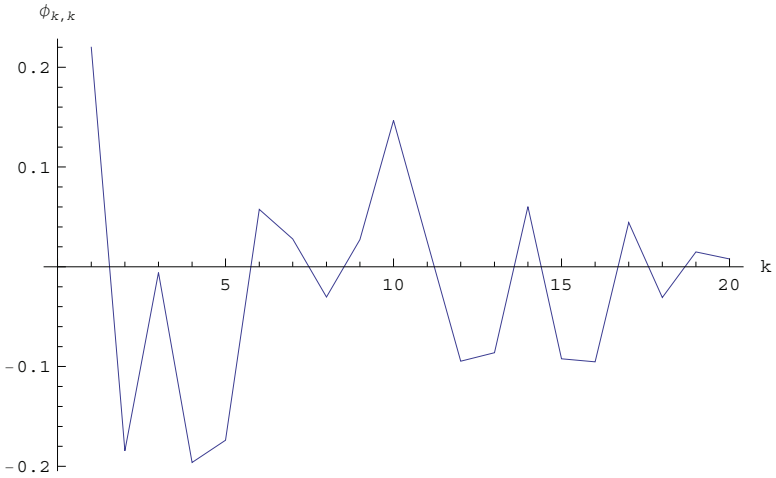
Out[12]=



Here the sample partial correlation function is plotted.

```
In[13]:= ListLinePlot[PartialCorrelationFunction[data, 20], AxesLabel -> {"k", "ϕk,k"}]
```

Out[13]=



We see that both the correlation and partial correlation functions decay quickly to small values. This indicates that the series can be fitted to models of relatively low order. We use the Hannan-Rissanen procedure to select models and get preliminary estimates.

These are five models selected by the Hannan-Rissanen procedure.

```
In[14]:= HannanRissanenEstimate[data, 10, 6, 6, 5]

Out[14]= {ARModel[{0.223037}, 0.0246806], MAModel[{0.218863}, 0.0250601],
          ARModel[{0.265727, -0.1874}, 0.0239958], MAModel[{0.231584, -0.153218}, 0.0246679],
          ARMAModel[{0.523774, -0.255097}, {-0.306381}, 0.0238409]}
```

These selected models with their estimated parameters are used as input to the function that performs the conditional maximum likelihood estimate.

The five models are further estimated using the conditional maximum likelihood method.

```
In[15]:= models = ConditionalMLEstimate[data, #] & /@ %

Out[15]= {ARModel[{0.223037}, 0.0246806], MAModel[{0.30951}, 0.0239137],
          ARModel[{0.265727, -0.1874}, 0.0239958], MAModel[{0.251582, -0.131713}, 0.0235999],
          ARMAModel[{0.937435, -0.318558}, {-0.718786}, 0.0233558]}
```

We obtain the AIC values for the five models, AR(1), MA(1), AR(2), MA(2), and ARMA(2, 1), specified above.

This gives the AIC values.

```
In[16]:= AIC[#, n] & /@ %

Out[16]= {-3.68154, -3.7131, -3.68947, -3.70611, -3.6963}
```

We see that the lowest AIC values correspond to MA(1) and MA(2) models. We choose these two models for further exploration using the maximum likelihood method.

This gives the maximum likelihood estimate of the MA(1) model.

```
In[17]:= ma1 = MLEstimate[data, MAModel[{θ1}, 1], {θ1, {0.3, 0.305}}]

Out[17]= MAModel[{0.309508}, 0.0239101]
```

This is the AIC value of the MA(1) model.

```
In[18]:= AIC[%, n]

Out[18]= -3.71325
```

This gives the maximum likelihood estimate of the MA(2) model.

```
In[19]:= ma2 =
          MLEstimate[data, MAModel[{θ1, θ2}, 1], {θ1, {0.25, 0.255}}, {θ2, {-0.13, -0.125}}]

Out[19]= MAModel[{0.256541, -0.122621}, 0.0235922]
```

This is the AIC value of the MA(2) model.

```
In[20]:= AIC[%, n]

Out[20]= -3.70643
```

Again MA(1) is superior according to the AIC criterion. Note that in this particular example, the estimated parameters using the maximum likelihood method are very close to those obtained using the conditional maximum likelihood method. We now tentatively choose MA(1) as our model and proceed to check its adequacy.

From (5.6) the variance of the sample correlation for an MA(1) model for lags $k > 1$ can be calculated.

This calculates the variance of the sample correlation.

```
In[21]:= (1 + 2 corr[[2]]^2) / n

Out[21]= 0.0110795
```

Here is the bound.

```
In[22]:= 2 Sqrt[%]

Out[22]= 0.210518
```

The sample correlation function shown above is within this bound with the exception of $\hat{\rho}(5)$. So the behavior of the sample correlation function obtained from the data is consistent with the MA(1) model. Next we calculate the residuals and see if they form a white noise process.

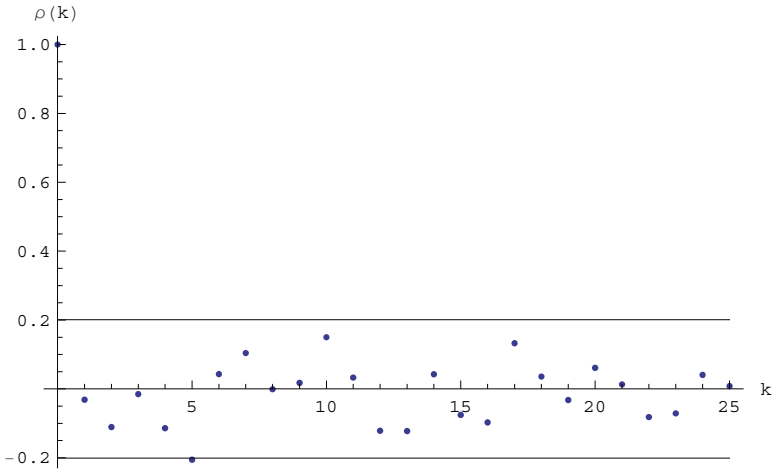
The residuals are calculated first.

```
In[23]:= res = Residual[data, ma1];
```

The residual correlation function is displayed along with the bounds.

```
In[24]:= Show[plotcorr[CorrelationFunction[res, 25]],
          Plot[{-2 / Sqrt[n], 2 / Sqrt[n]}, {x, 0, 25}, PlotStyle -> {{Black}}],
          PlotRange -> All, AxesLabel -> {"k", "ρ(k)"}]
```

Out[24]=



We see that the correlation function of the residuals resembles that of a white noise. The portmanteau test also confirms the adequacy of the MA(1) model.

This gives the portmanteau statistic Q_h with $h = 20$.

```
In[25]:= PortmanteauStatistic[res, 20]

Out[25]= 19.7014
```

The model passes the test.

```
In[26]:= Quantile[ChiSquareDistribution[19], 0.95]

Out[26]= 30.1435
```

So we finally arrive at the model $(1 - B) \ln X_t = 0.029 + Z_t + 0.31\, Z_{t-1}$ where the noise $\{Z_t\}$ has a variance 0.024.

We can get the standard error of $\hat{\theta}_1$ by calculating the asymptotic variance of the estimator.

Here is the asymptotic variance of $\hat{\theta}_1$.

```
In[27]:= AsymptoticCovariance[ma1]

Out[27]= {{0.904205}}
```

This is the standard error of $\hat{\theta}_1$.

```
In[28]:= Sqrt[% / n]

Out[28]= {{0.0955687}}
```

The disadvantage of choosing an MA(1) model is that it cannot account for the relatively large value of the sample correlation at lag 5. To account for the large $\hat{\gamma}(5)$ a higher-order model is needed. For example, the AR(5) model obtained from the maximum likelihood estimate (with ϕ_3 fixed to be 0)

```
ARModel[{0.223396, -0.209017, 0, -0.147311, -0.171337}, 0.0219622]
```

produces a correlation function that better resembles the sample correlation function. It also has a slightly lower AIC value than the MA(1) model.

Example 11.2 The file `unemp.dat` contains the U.S. monthly unemployment figures in thousands from 1948 to 1981 for men of ages between 16 and 19.

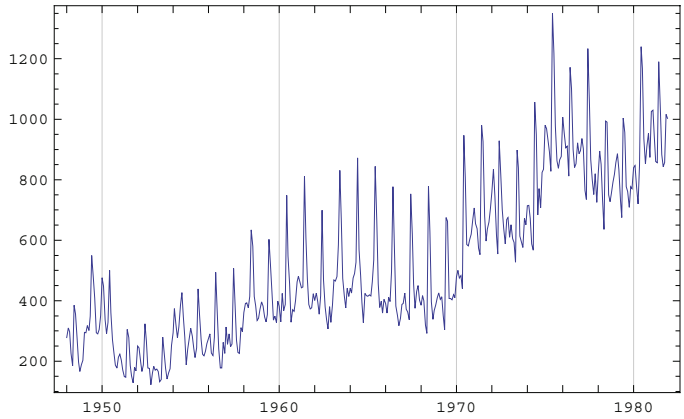
This reads in the data.

```
In[29]:= unemp = ReadList[ToFileName[{"TimeSeries", "Data"}, "unemp.dat"], Number];
```

Here is a plot of the data.

```
In[30]:= DateListPlot[unemp, {{1948, 1}, {1981, 12}}, Joined -> True, AxesLabel -> {"t", "x_t"}]
```

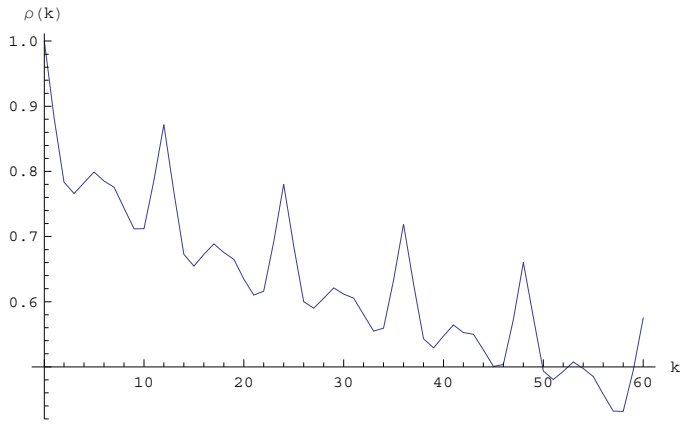
Out[30]=



We plot the sample correlation function.

```
In[31]:= plotcorr[CorrelationFunction[unemp, 60], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

Out[31]=



The clear periodic structure in the time plot of the data is reflected in the correlation plot. The pronounced peaks at lags that are multiples of 12 indicate that the series has a seasonal period $s = 12$. We also observe that the correlation function decreases rather slowly to zero both for $\hat{\rho}(k\,s)$ ($k = 1, 2, \ldots$) and for $\hat{\rho}(k)$ (k is not a multiple of 12). This suggests that the series may be nonstationary and both seasonal and regular differencing may be required.

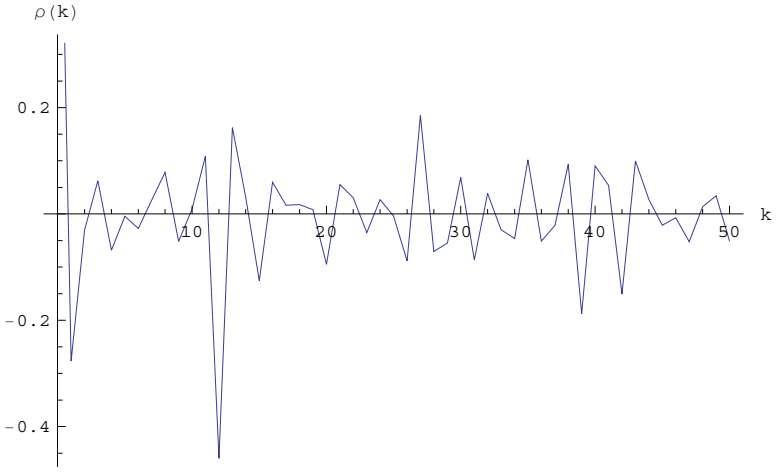
We first transform the series by differencing.

```
In[32]:= data = N[ListDifference[unemp, {1, 1}, 12]];
```

This is the plot of the sample correlation function of the transformed series.

```
In[33]:= plotcorr[CorrelationFunction[data, 50], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

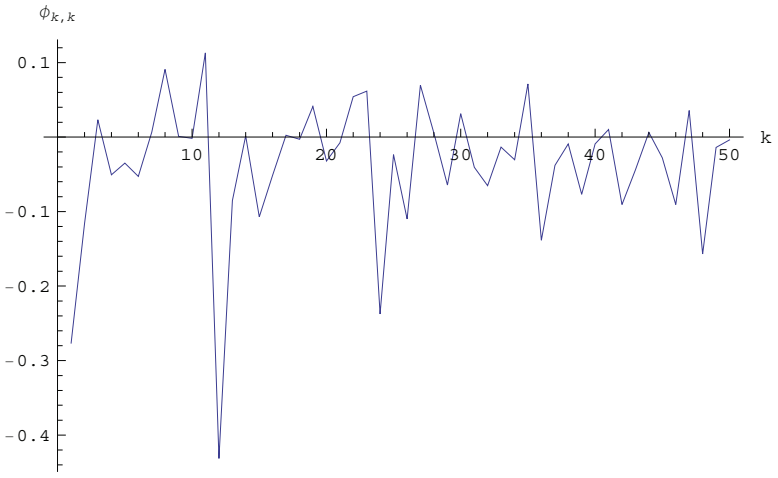
Out[33]=



This is the plot of the sample partial correlation function of the transformed series.

```
In[34]:= ListLinePlot[PartialCorrelationFunction[data, 50],
          AxesLabel -> {"k", "ϕk,k"}, PlotRange -> All]
```

Out[34]=



The single prominent dip in the correlation function at lag 12 shows that the seasonal component probably only persists in the MA part and $Q = 1$. This is also consistent with the behavior of the partial correlation function

that has dips at lags that are multiples of the seasonal period 12. The correlation plot also suggests that the orders of the regular AR and MA parts are small since $\hat{\gamma}(k)$ is prominent only at $k = 1$. Based on these observations, we consider the models with $P = 0$, $Q = 1$ and $p \le 1$, $q \le 1$ and the reader is encouraged to try higher-order models. We will estimate the parameters of these models using the conditional maximum likelihood method. Specifically, we consider the following three models.

This defines the three models.

```
In[35]:= models = {SARIMAModel[{0, 0}, 12, {-0.3}, {}, {}, {-0.5}, 1],
            SARIMAModel[{0, 0}, 12, {}, {}, {-0.3}, {-0.5}, 1],
            SARIMAModel[{0, 0}, 12, {-0.3}, {}, {-0.3}, {-0.5}, 1]};
```

The initial values -0.3 and -0.5 for the parameters are guessed from looking at the sample correlation and partial correlation functions. For an AR(1) or MA(1) model, $\rho(1)$ is proportional to ϕ_1 or θ_1. So we pick our initial value for ϕ_1 or θ_1 to be $\hat{\rho}(1) \approx -0.3$. For the partial correlation function of an MA(1) model we have $\phi_{k+1,k+1} / \phi_{k,k} \approx -\theta_1$ (see Example 2.11). From our sample partial correlation function, we see that $\hat{\phi}_{24,24} / \hat{\phi}_{12,12} \approx 0.5$, so we choose the initial value of Θ_1 to be -0.5. Before estimating the parameters we first make sure that the mean value of the series is zero.

This gives the sample mean of the data.

```
In[36]:= μ = Mean[data]

Out[36]= 0.283544
```

The mean is subtracted from the data.

```
In[37]:= data = data - μ;
```

n is the length of the data.

```
In[38]:= n = Length[data]

Out[38]= 395
```

This gives the conditional maximum likelihood estimate of the three models.

```
In[39]:= ConditionalMLEstimate[data, #] & /@ models

Out[39]= {SARIMAModel[{0, 0}, 12, {-0.257958}, {}, {}, {-0.66107}, 2377.72],
          SARIMAModel[{0, 0}, 12, {}, {}, {-0.324612}, {-0.663367}, 2337.27],
          SARIMAModel[{0, 0}, 12, {0.207394}, {}, {-0.514409}, {-0.675095}, 2322.52]}
```

These are the AIC values of the models.

```
In[40]:= AIC[#, n] & /@ %

Out[40]= {7.78402, 7.76687, 7.7656}
```

We drop the first model because it has a higher AIC value. We use the exact maximum likelihood method to get better estimates for the parameters of the second and third models.

This gives the maximum likelihood estimate of the second model.

```
In[41]:= mamodel = MLEstimate[data, SARIMAModel[{0, 0}, 12, {}, {}, {θ}, {Θ}, 1],
           {θ, {-0.32, -0.325}}, {Θ, {-0.66, -0.665}}]

Out[41]= SARIMAModel[{0, 0}, 12, {}, {}, {-0.325634}, {-0.663346}, 2306.78]
```

This is the AIC value of the model.

```
In[42]:= AIC[%, n]

Out[42]= 7.75374
```

Here is the maximum likelihood estimate of the third model.

```
In[43]:= MLEstimate[data, SARIMAModel[{0, 0}, 12, {ϕ}, {}, {θ}, {Θ}, 1],
           {ϕ, {0.195, 0.2}}, {θ, {-0.5, -0.505}}, {Θ, {-0.665, -0.67}}]

Out[43]= SARIMAModel[{0, 0}, 12, {0.209597}, {}, {-0.51579}, {-0.670341}, 2295.62]
```

We obtain the AIC value of this model.

```
In[44]:= AIC[%, n]

Out[44]= 7.75395
```

So the second model, SARIMA$(0, 1, 1)(0, 1, 1)_{12}$, is marginally superior and we choose it to be our model. We check to see if the model is invertible.

The model is invertible.

```
In[45]:= InvertibleQ[mamodel]

Out[45]= True
```

To test the adequacy of the model we first calculate the residuals.

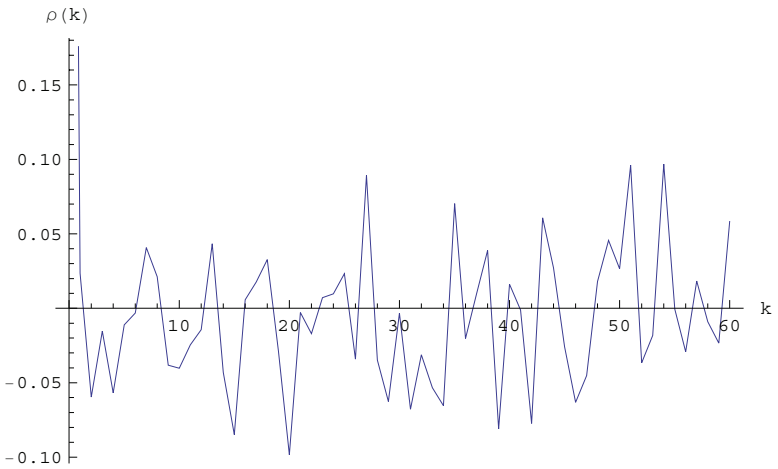
We find the residuals.

```
In[46]:= res = Residual[data, mamodel];
```

This plots the sample correlation function of the residuals.

```
In[47]:= plotcorr[CorrelationFunction[res, 60], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

Out[47]=



We see that the residual correlation is within the rough bound $2/\sqrt{n} \approx 0.1$. Even though the asymptotic standard deviations of residual correlations can be smaller than $1/\sqrt{n}$ for small lags, our residual correlations for small lags are much smaller than the rough bound 0.1.

This gives the portmanteau statistic Q_{40}.

```
In[48]:= PortmanteauStatistic[res, 40]

Out[48]= 33.0707
```

The model passes the portmanteau test.

```
In[49]:= Quantile[ChiSquareDistribution[38], 0.95]

Out[49]= 53.3835
```

So we conclude that our model is $(1-B)\left(1-B^{12}\right)X_t = 0.2835 + (1-0.326\,B)\left(1-0.663\,B^{12}\right)Z_t$ where the white noise $\{Z_t\}$ has variance 2306.8.

To estimate the standard errors of the estimated parameters we invert the estimated information matrix.

This gives the inverse of the information matrix.

```
In[50]:= Inverse[InformationMatrix[data, mamodel]]

Out[50]= {{0.89234, 0.0353462}, {0.0353462, 0.581618}}
```

The standard errors are calculated from the diagonal elements.

```
In[51]:= Sqrt[Diagonal[%] / n]

Out[51]= {0.0475299, 0.0383725}
```

The standard errors are about 0.05 for $\hat{\theta}_1$ and 0.04 for $\hat{\Theta}_1$.

To forecast the future values we first get the predictors for the differenced, zero-mean series `data`.

This gives the predicted values for `data`.

```
In[52]:= BestLinearPredictor[data, mamodel, 15, Exact -> False][[1]];
```

We have used the option `Exact -> False` since n is large and hence it makes little difference in the predicted values. To get the predicted values for the original series we use the function `IntegratedPredictor`.

This gives the predicted values for the original series.

```
In[53]:= xhat = IntegratedPredictor[unemp, {1, 1}, 12, % + μ]

Out[53]= {1079.07, 1095.41, 1031.91, 952.135, 951.854, 1323.82, 1225.77,
          1032.37, 986.318, 994.78, 1085.28, 1058.67, 1147.18, 1163.8, 1100.58}
```

The mean square errors of the prediction are obtained by using the original series and the full model. In this case the full model is obtained by replacing the differencing orders {0, 0} in `mamodel` (see `Out[44]`) by {1, 1}.

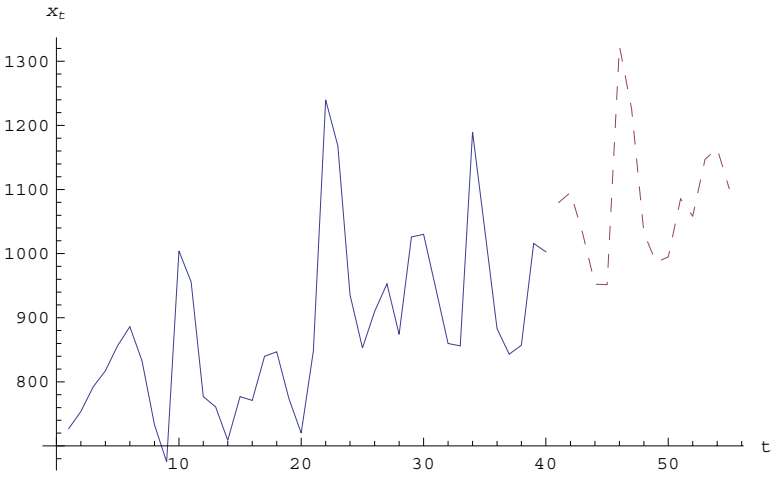
These are the mean square errors of the prediction.

```
In[54]:= BestLinearPredictor[unemp, mamodel /. {0, 0} -> {1, 1}, 15][[2]]

Out[54]= {2306.78, 3355.83, 4404.89, 5453.94, 6502.99, 7552.05, 8601.1,
          9650.15, 10 699.2, 11 748.3, 12 797.3, 13 846.4, 16 204.3, 18 078.5, 19 952.8}
```

Here we display the next 15 values of the series predicted from the estimated SARIMA model along with the last 40 observed data points.

```
In[55]:= ListLinePlot[{Take[unemp, -40], Transpose[{Range[41, 55], xhat}]},
          PlotStyle -> {{Thin}, {Dashing[{0.02}]}}, AxesLabel -> {"t", "x_t"}]
```

Out[55]=



We can also define a function, say, `plotdata`, to plot the data and the predicted values together.

`plotdata` takes the data and the predicted values and plots them together.

```
In[56]:= plotdata[x_, xhat_, opts___] := ListLinePlot[
           {x,  Transpose[{Range[Length[x] + 1, Length[x] + Length[xhat]], xhat}]},
           PlotStyle -> {{Thin}, {Dashing[{0.02}]}}, opts]
```

Example 11.3 In this example, we model the celebrated data recorded for the number of lynx trapped annually in northwest Canada from 1821 to 1934. The series has been analyzed by various authors and researchers and a survey of the work can be found in Campbell and Walker (1977). The lynx data are contained in the file `lynx.dat` and we first read in the data and plot them.

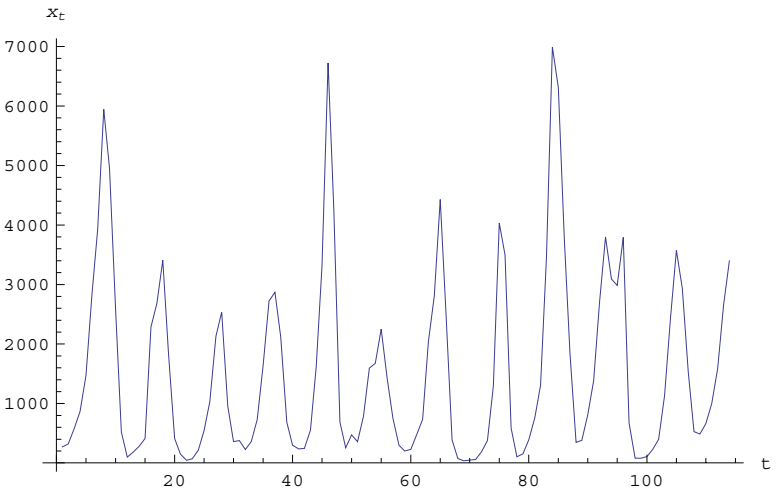
This reads in the data.

```
In[57]:= data = N[ReadList[ToFileName[{"TimeSeries", "Data"}, "lynx.dat"], Number]];
```

We plot the number of lynx trapped against time. Note that the time scale is such that $t = 1$ corresponds to the year 1821. See Example 4.2.

```
In[58]:= ListLinePlot[data, AxesLabel -> {"t", "x_t"}]
```

Out[58]=



We can see that the series oscillates with an approximate period of 10. In order to make the variance more uniform, we first take the logarithm (base 10) of the data and subtract out the mean.

`n` is the length of the data.

```
In[59]:= n = Length[data]

Out[59]= 114
```

We obtain the sample mean.

```
In[60]:= μ = Mean[Log[10, data]]

Out[60]= 2.90366
```

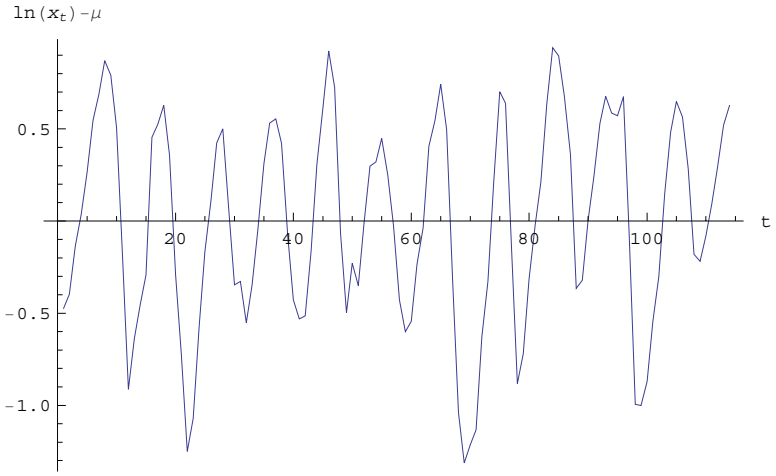
The series is transformed to a zero-mean one.

```
In[61]:= data1 = Log[10, data] - %;
```

This is the plot of the transformed data.

```
In[62]:= ListLinePlot[data1, AxesLabel -> {"t", "ln(x_t)-μ"}]
```

Out[62]=

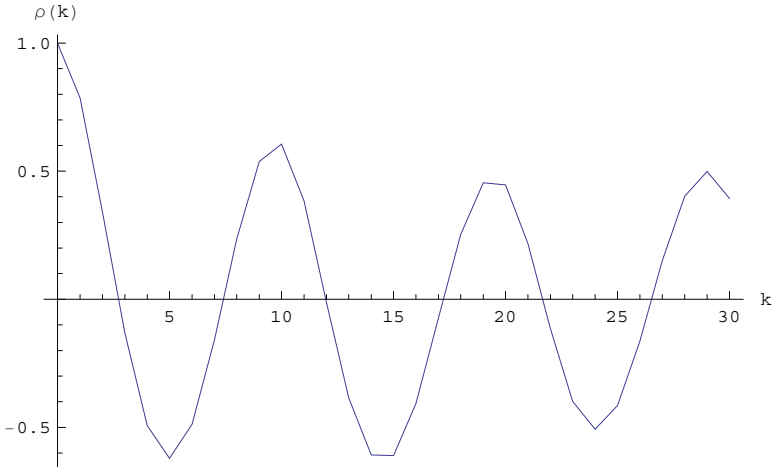


Compared with the original plot the time plot now looks more symmetric. We proceed to calculate and plot the correlation and partial correlation functions.

This is the plot of the sample correlation function.

```
In[63]:= plotcorr[CorrelationFunction[data1, 30], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

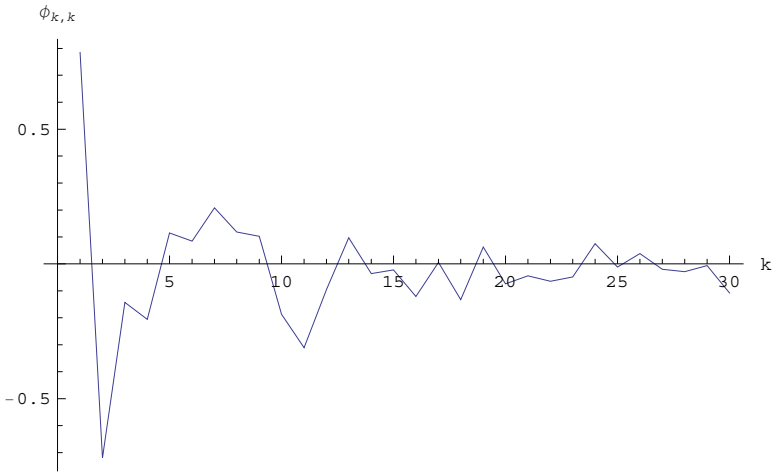
Out[63]=



This is the plot of the sample partial correlation function.

```
In[64]:= ListLinePlot[PartialCorrelationFunction[data1, 30],
          AxesLabel -> {"k", "ϕk,k"}, PlotRange -> All]
```

Out[64]=



We notice that, like the time series itself, the correlation function oscillates with an approximate cycle of period 10. Also the slowly decaying behavior of the correlation signals that the series may be nonstationary or the AR polynomial $\phi(x)$ has zeros very close to the unit circle. As a preliminary estimate we use the Hannan-Rissanen method to select possible models. Since the period here is close to 10 we will use the long AR order and the maximum AR and MA orders greater than 10.

Hannan-Rissanen method is used to select models.

```
In[65]:= models = HannanRissanenEstimate[data1, 15, 15, 15, 30];
```

These are the AIC values of the models.

```
In[66]:= aic = AIC[#, n] & /@ %
```

```
Out[66]= {-3.08725, -3.10169, -3.12223, -3.0767, -3.11083, -3.08439, -3.10754, -3.07451,
          -2.96607, -3.10672, -3.03437, -3.17608, -3.05049, -3.09683, -2.92848,
          -2.9738, -2.973, -3.06685, -3.08944, -3.00982, -3.05714, -3.07833, -3.05084,
          -2.97709, -3.11752, -3.09279, -3.11432, -3.01811, -3.08997, -3.16163}
```

There are two models that have much lower AIC values than the rest.

We pick out the models with AIC value less than -3.15. They are models 12 and 30.

```
In[67]:= Position[aic, #] & /@ Select[aic, # < -3.15 &]
```

```
Out[67]= {{{12}}, {{30}}}
```

This gives the two models.

```
In[68]:= newmodels = models[[#]] & /@ Flatten[%]

Out[68]= {ARMAModel[{1.4814, -0.917287, 0.483792, -0.38725, 0.255133, -0.199222,
     0.123935, -0.0844358, 0.208183, 0.0841331, -0.295}, {-0.415827}, 0.0338232],
   ARMAModel[{1.53925, -1.11897, 0.669973, -0.485153, 0.296318, -0.242446, 0.147927,
     -0.110583, 0.233119, 0.0513117, -0.260468}, {-0.441095, 0.169184}, 0.0337188]}
```

They correspond to ARMA(11, 1) and ARMA(11, 2).

```
In[69]:= ARMAModel[Length[#[[1]]], Length[#[[2]]]] & /@ %

Out[69]= {ARMAModel[11, 1], ARMAModel[11, 2]}
```

We further estimate the model parameters using the conditional maximum likelihood method. Note that the above models are used as the input to the function `ConditionalMLEstimate`.

This gives the conditional maximum likelihood estimate.

```
In[70]:= ConditionalMLEstimate[data1, #] & /@ newmodels

Out[70]= {ARMAModel[{1.46137, -0.910323, 0.487405, -0.431681, 0.293584, -0.239092,
     0.14703, -0.087387, 0.196867, 0.0649484, -0.264829}, {-0.365644}, 0.035317],
   ARMAModel[{1.64868, -1.41519, 0.945867, -0.682653, 0.455565, -0.390716, 0.255153,
     -0.179856, 0.278468, -0.0212403, -0.187538}, {-0.555271, 0.351705}, 0.0344438]}
```

Here are the AIC values.

```
In[71]:= AIC[#, 114] & /@ %

Out[71]= {-3.13286, -3.14036}
```

We can also try to estimate some nearby models to see if they give a lower AIC value. For example, we can try AR(12), ARMA(10, 1), etc. It turns out that AR(12) has a lower AIC value. The reader can show that AR(13) and ARMA(10, 1) yield higher AIC values than AR(12).

This gives the estimate of AR(12) model.

```
In[72]:= ar12 = ConditionalMLEstimate[data1, 12]

Out[72]= ARModel[{1.06557, -0.443781, 0.272538, -0.298825, 0.142786, -0.152363,
     0.0723808, -0.0562743, 0.191533, 0.139729, -0.217959, -0.129305}, 0.0338232]
```

Here is the AIC value.

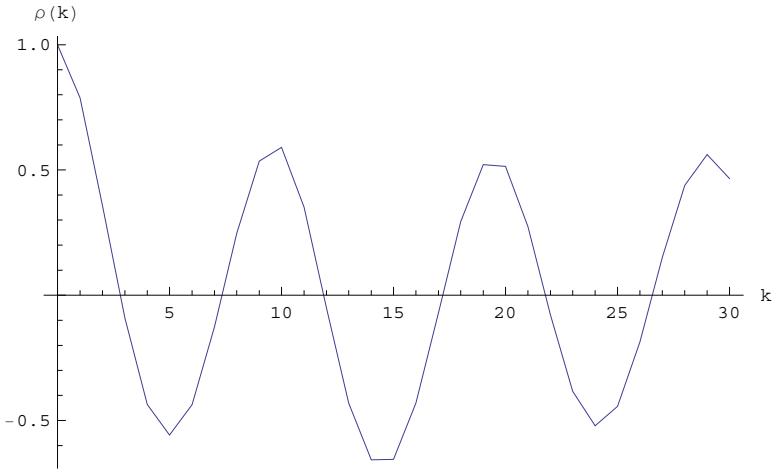
```
In[73]:= AIC[%, n]

Out[73]= -3.17608
```

The correlation and the partial correlation function of the AR(12) model resemble those obtained from `data1` (see earlier plots).

This is the plot of the correlation function of the AR(12) model.

```
In[74]:= plotcorr[CorrelationFunction[ar12, 30], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

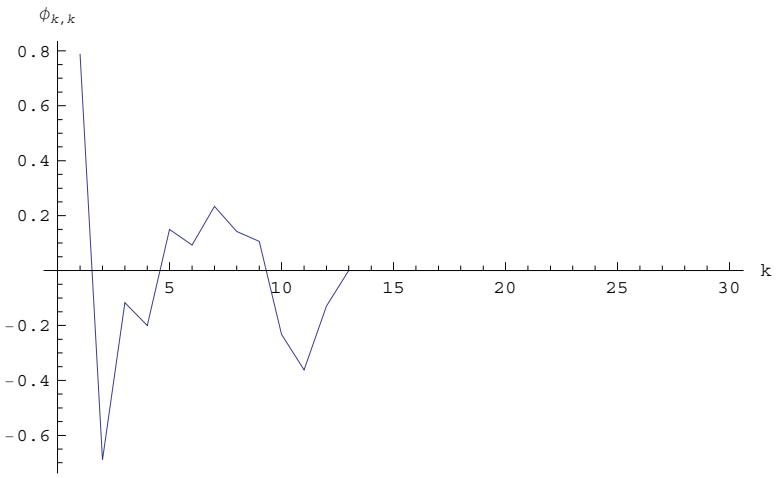
Out[74]=



This is the plot of the partial correlation function of the AR(12) model.

```
In[75]:= ListLinePlot[PartialCorrelationFunction[ar12, 30],
          AxesLabel -> {"k", "ϕk,k"}, PlotRange -> All]
```

Out[75]=



To test the adequacy of the model we first calculate the residuals and their correlation function.

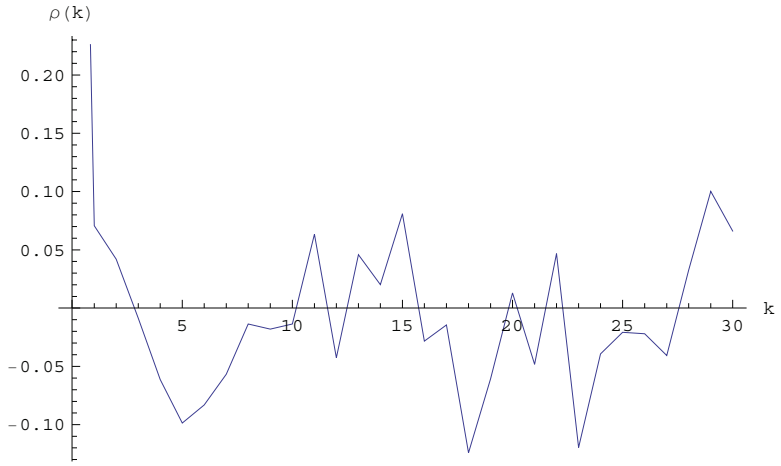
The residuals are calculated first.

```
In[76]:= res = Residual[data1, ar12];
```

This is the plot of the sample correlation function of the residuals.

```
In[77]:= plotcorr[CorrelationFunction[res, 30], Joined -> True, AxesLabel -> {"k", "ρ(k)"}]
```

Out[77]=



The correlation function of the residuals is well within the bound of $2/\sqrt{114} \approx 0.187$.

This is the portmanteau statistic for $h = 25$.

```
In[78]:= PortmanteauStatistic[res, 25]

Out[78]= 11.5382
```

So the AR(12) model passes the test.

```
In[79]:= Quantile[ChiSquareDistribution[13], 0.95]

Out[79]= 22.362
```

To get the standard errors of the estimated parameters we first get their asymptotic covariance matrix. This can be done either by inverting the information matrix or using the function `AsymptoticCovariance`. Here we use the latter method.

The asymptotic covariance matrix is calculated.

```
In[80]:= AsymptoticCovariance[ar12];
```

This gives the standard errors.

```
In[81]:= Sqrt[Diagonal[%] / n]

Out[81]= {0.0928723, 0.134791, 0.140445, 0.141614, 0.144257, 0.144717,
          0.144717, 0.144257, 0.141614, 0.140445, 0.134791, 0.0928723}
```

We put the estimated parameters together with their standard errors using `TableForm`.

```
In[82]:= TableForm[Transpose[{ar12[[1]], %}]]

Out[82]//TableForm=
      1.06557     0.0928723
      -0.443781   0.134791
      0.272538    0.140445
      -0.298825   0.141614
      0.142786    0.144257
      -0.152363   0.144717
      0.0723808   0.144717
      -0.0562743  0.144257
      0.191533    0.141614
      0.139729    0.140445
      -0.217959   0.134791
      -0.129305   0.0928723
```

So we can safely regard ϕ_5, ϕ_6, ϕ_7, ϕ_8, and ϕ_{10} to be zero. However we will use the model `ar12` to forecast the future values.

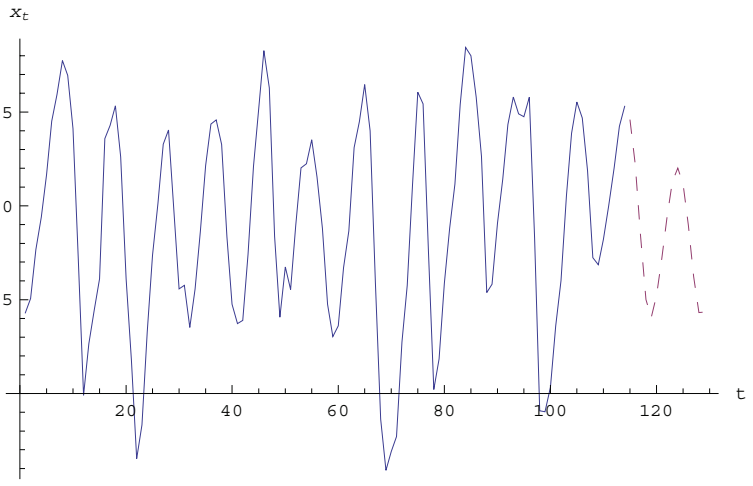
This yields the forecast and mean square forecast errors for the number of lynx trappings for the next 15 years.

```
In[83]:= {xhat, error} = BestLinearPredictor[data1, ar12, 15]

Out[83]= {{0.556341, 0.317203, -0.0612981, -0.402819,
            -0.499095, -0.382077, -0.183767, 0.0370242, 0.225049, 0.297504,
            0.212359, -0.00674495, -0.280673, -0.471951, -0.46962},
           {0.0338232, 0.0722275, 0.0884085, 0.0981504, 0.100376, 0.100454, 0.100968,
            0.102826, 0.105228, 0.10551, 0.108664, 0.113813, 0.114546, 0.114786, 0.117508}}
```

We plot the logarithm of the lynx data along with the next 15 predicted values.

```
In[84]:= plotdata[Log[10, data], xhat + μ, AxesLabel -> {"t", "x_t"}]
```

Out[84]=



Example 11.4 The file `salesad.dat` contains the Lydia E. Pinkham annual sales ($\{x_{t1}\}$) and advertising expenditure ($\{x_{t2}\}$) data. They have the following time plots.

This reads in the data.

```
In[85]:= data =
           N[ReadList[ToFileName[{"TimeSeries", "Data"}, "salesad.dat"], {Number, Number}]];
```

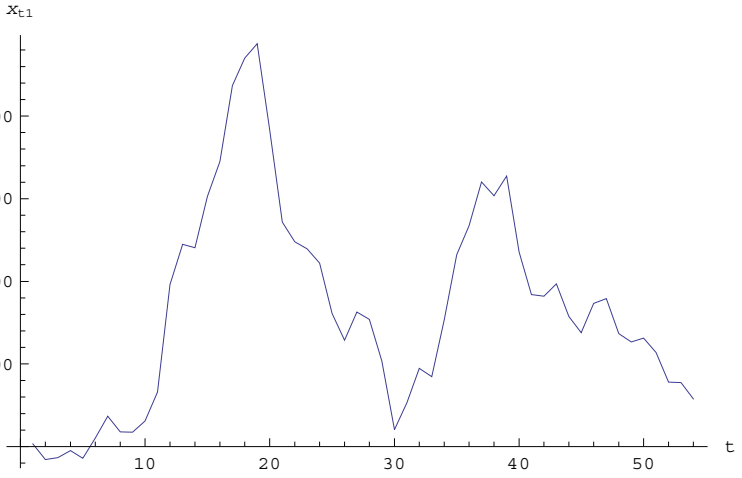
n is the length of the data.

```
In[86]:= n = Length[data]

Out[86]= 54
```

Here is the time plot of series 1.

```
In[87]:= ListLinePlot[data[[All, 1]], AxesLabel -> {"t", "x_t1"}]
```

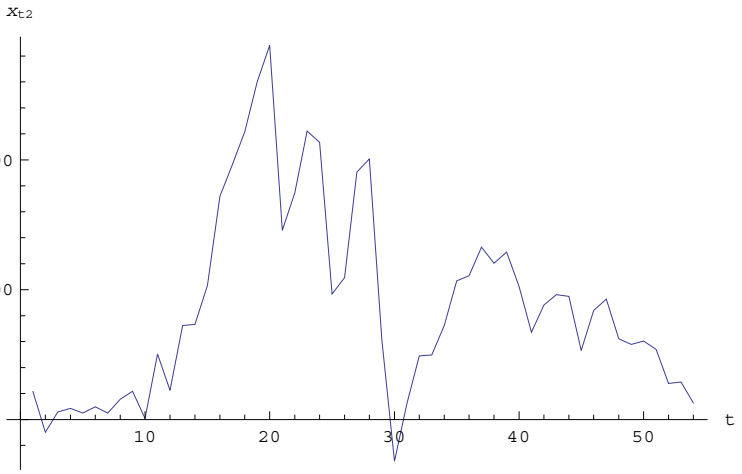
Out[87]=



Here is the time plot of series 2.

```
In[88]:= ListLinePlot[data[[All, 2]], AxesLabel -> {"t", "x_t2"}]
```

Out[88]=



Since we do not see a convincing systematic trend, we transform the series to zero-mean by simply subtracting the sample mean of the series.

This is the sample mean.

```
In[89]:= μ = Mean[data]

Out[89]= {1829.48, 934.519}
```

The mean is subtracted.

```
In[90]:= data0 = (# - μ) & /@ data;
```

We use the Hannan-Rissanen procedure to select preliminary models.

```
In[91]:= models = HannanRissanenEstimate[data0, 5, 5, 5, 6];

       Inverse::luc : Result for Inverse of badly conditioned matrix
          {{1.60792×10^7, <<9>>, <<6>>}, {7.98007×10^6, <<9>>, <<6>>}, <<7>>, {<<1>>}, <<6>>}
          may contain significant numerical errors. >>
```

This allows us to see the orders of each selected model.

```
In[92]:= If[Head[#] === ARMAModel,
            Head[#][Length[#[[1]]], Length[#[[2]]]], Head[#][Length[#[[1]]]]] & /@ models

Out[92]= {ARModel[1], ARModel[3], ARMAModel[2, 1], ARMAModel[1, 1], ARModel[2], ARMAModel[1, 2]}
```

This gives their AIC values.

```
In[93]:= AIC[#, 54] & /@ models

Out[93]= {20.9673, 20.7213, 20.7263, 20.9195, 20.9213, 20.842}
```

Since the second (AR(3)) and third (ARMA(2, 1)) models have the smallest AIC values, we choose them to be our preliminary models. The conditional maximum likelihood method gives the following parameter estimates.

This gives the conditional maximum likelihood estimate.

```
In[94]:= {ar3, arma21} = ConditionalMLEstimate[data0, #] & /@ Take[models, {2, 3}]

Out[94]= {ARModel[{{{1.36898, -0.162305}, {0.585785, 0.51886}},
             {{-0.30868, -0.280303}, {-0.536465, -0.339297}},
             {{0.00254444, 0.100379}, {0.264333, 0.273705}}},
            {{36490.3, 14660.1}, {14660.1, 23427.2}}},
          ARMAModel[{{{1.27204, -0.197557}, {-0.597596, 0.77419}},
             {{-0.180859, -0.235908}, {0.924474, -0.630635}}},
            {{{0.0445299, 0.0728988}, {1.22388, -0.360301}}},
            {{35762.5, 15279.6}, {15279.6, 23141.6}}}}
```

In fact, it is unnecessary to further estimate the parameters of AR(3) model as we have done above, since the Hannan-Rissanen method and the conditional maximum likelihood method give identical parameter estimates for an AR(p) model. The reader should convince himself/herself of this.

This gives the AIC values.

```
In[95]:= AIC[#, 54] & /@ %

Out[95]= {20.7213, 20.6471}
```

Since AR(2, 1) model has a lower AIC value, we choose it as our model and plot the residual correlation functions and cross-correlation function.

This gives the residuals.

```
In[96]:= res = Residual[data0, arma21];
```

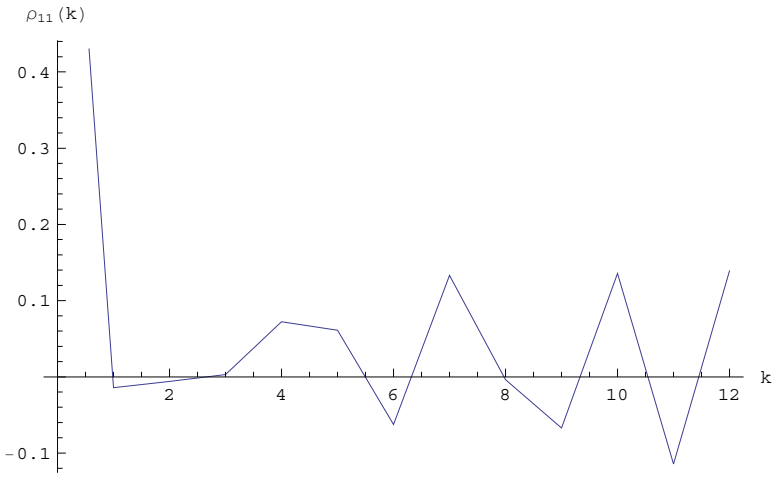
This calculates the correlation function of the residuals.

```
In[97]:= corr = CorrelationFunction[res, 12];
```

Here is the plot of the residual correlation function for series 1.

```
In[98]:= plotcorr[#[[1, 1]] & /@ corr, Joined -> True, AxesLabel -> {"k", "ρ11 (k)"}]
```

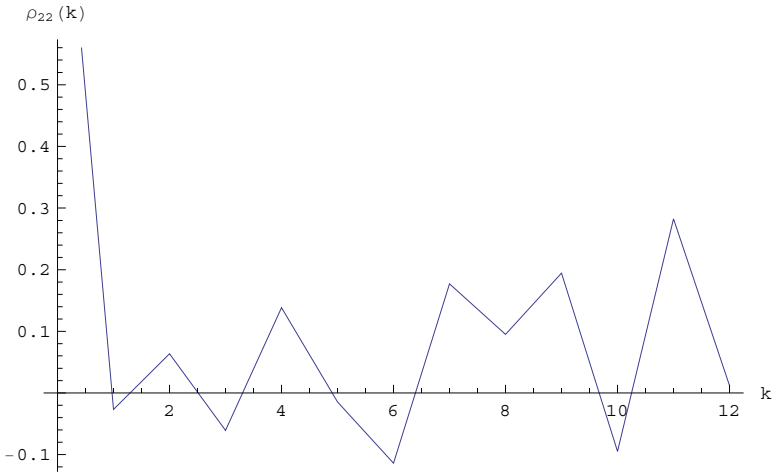
Out[98]=



Here is the plot of the residual correlation function for series 2.

```
In[99]:= plotcorr[#[[2, 2]] & /@ corr, Joined -> True, AxesLabel -> {"k", "ρ22 (k)"}]
```

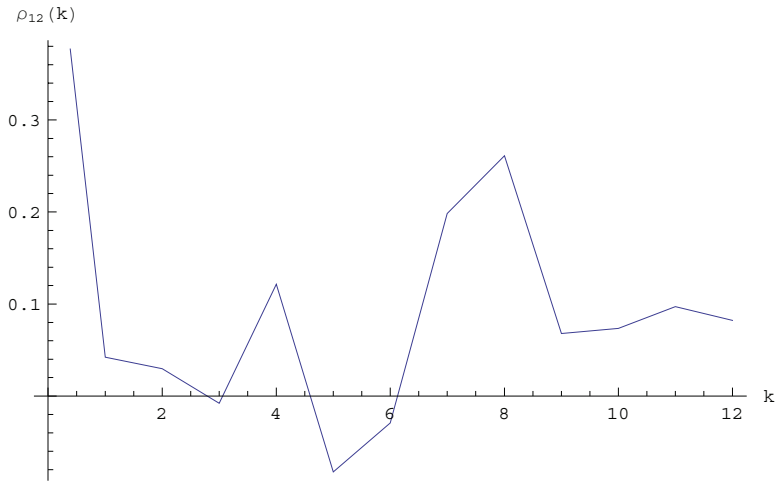
Out[99]=



This is the plot of the residual cross-correlation function.

```
In[100]:=
    plotcorr[#[[1, 2]] & /@ corr, Joined -> True, AxesLabel -> {"k", "ρ12 (k)"}]
```

Out[100]=



The residual correlation functions resemble those of white noise (since $2/\sqrt{54} \approx 0.272$). The model also passes the portmanteau test.

This gives Q_{12}.

```
In[101]:=
    PortmanteauStatistic[res, 12]

Out[101]=
    38.9857
```

The model passes the test.

```
In[102]:=
Quantile[ChiSquareDistribution[36], 0.95]

Out[102]=
50.9985
```

This calculates the inverse of the information matrix.

```
In[103]:=
Inverse[InformationMatrix[data0, arma21]];
```

Here are the standard errors of estimated parameters.

```
In[104]:=
Sqrt[Diagonal[%] / n]

Out[104]=
{0.300932, 0.214004, 0.430938, 0.389822, 0.30574, 0.192439,
 0.446683, 0.337158, 0.345876, 0.225473, 0.449526, 0.425352}
```

These are the standard errors of $\{(\Phi_1)_{11}, (\Phi_1)_{12}, (\Phi_1)_{21}, \ldots, (\Theta_1)_{22}\}$. We can display the estimated parameters and their standard errors together using `TableForm`.

The parameters are displayed along with their standard errors.

```
In[105]:=
TableForm[Transpose[{{Phi[1], Phi[2], Theta[1]},
   Flatten[{arma21[[1]], arma21[[2]]}, 1], Partition[Partition[%, 2], 2]}]]

Out[105]//TableForm=
```

Phi[1]	1.27204	−0.197557	0.300932	0.214004
	−0.597596	0.77419	0.430938	0.389822
Phi[2]	−0.180859	−0.235908	0.30574	0.192439
	0.924474	−0.630635	0.446683	0.337158
Theta[1]	0.0445299	0.0728988	0.345876	0.225473
	1.22388	−0.360301	0.449526	0.425352

Again we can set several elements of the parameter matrices to zero. We find the forecast values for the next 15 time steps.

This gives the predicted values.

```
In[106]:=
BestLinearPredictor[data0, arma21, 15][[1]];
```

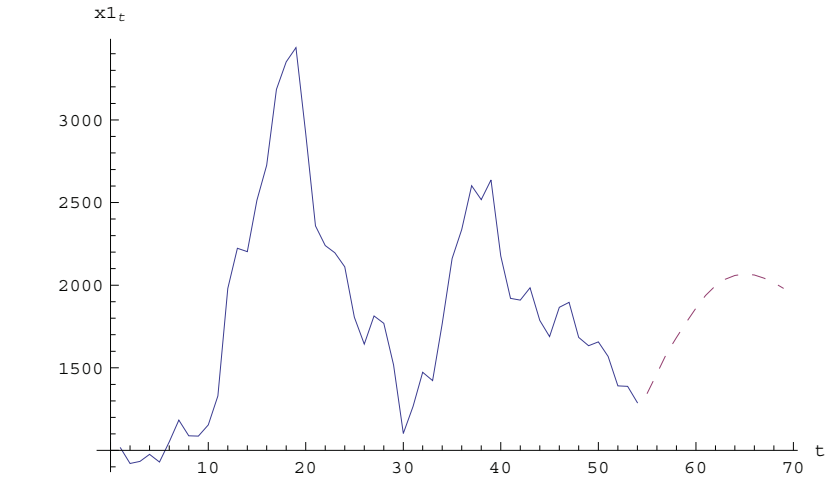
The mean is added to the prediction.

```
In[107]:=
xhat = # + μ & /@ %;
```

This plots series 1 and the next 15 predicted values.

```
In[108]:=
plotdata[data[[All, 1]], xhat[[All, 1]], AxesLabel -> {"t", "x1t"}]
```

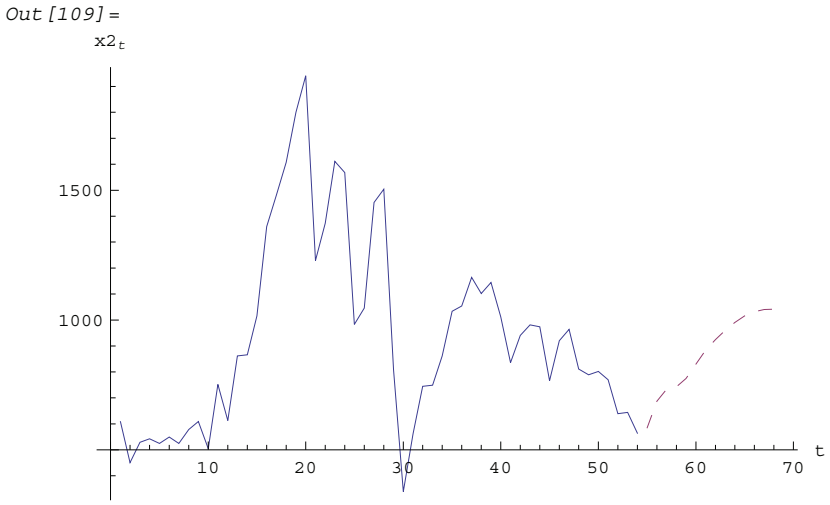
```
Out[108]=
```



This plots series 2 and the next 15 predicted values.

```
In[109]:=
plotdata[data[[All, 2]], xhat[[All, 2]], AxesLabel -> {"t", "x2t"}]
```

```
Out[109]=
```



Part 2.
Summary of *Time Series* Functions

2.1 Model Properties

A discrete time series consists of a set of observations $\{x_1, x_2, \ldots, x_t, \ldots\}$ of some phenomenon, taken at equally spaced time intervals. (We assume that x_t is real.) The main purpose of time series analysis is to understand the underlying mechanism that generates the observed data and, in turn, to forecast future values. We assume that the generating mechanism is probabilistic and that the observed series $\{x_1, x_2, \ldots, x_t, \ldots\}$ is a realization of a stochastic process $\{X_1, X_2, \ldots, X_t, \ldots\}$. Typically, the process is assumed to be stationary and described by a class of linear models called autoregressive moving average (ARMA) models.

An ARMA model of orders p and q (ARMA(p, q)) is defined by

$$X_t - \phi_1 X_{t-1} - \phi_2 X_{t-2} - \ldots - \phi_p X_{t-p} = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \ldots + \theta_q Z_{t-q}$$

where $\{\phi_i\}$ and $\{\theta_i\}$ are the coefficients of the autoregressive (AR) and moving average (MA) parts, respectively, and $\{Z_t\}$ is white noise with mean zero and variance σ^2. (We assume Z_t is normally distributed.) Using the backward shift operator B defined by $B^j X_t = X_{t-j}$, the ARMA(p, q) model above can be written as

$$\phi(B) X_t = \theta(B) Z_t$$

where $\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p$ and $\theta(B) = 1 + \theta_1 B + \theta_2 B^2 + \ldots + \theta_q B^q$. We assume that the polynomials $\phi(x)$ and $\theta(x)$ have no common zeros. When X_t is a vector, we have a multivariate or vector ARMA model; in this case, the AR and MA coefficients are matrices and the noise covariance is also a matrix denoted by Σ. When all the zeros of the polynomial $\phi(x)$ (or its determinant in the multivariate case) are outside the unit circle, the model is said to be stationary and the ARMA(p, q) model can be expanded formally as an MA(∞) model ($X_t = \sum_{i=0}^{\infty} \psi_i Z_{t-i}$). Similarly, if the zeros of $\theta(x)$ are all outside the unit circle, the ARMA(p, q) model is said to be invertible and can be expanded as an AR(∞) model.

Autoregressive integrated moving average (ARIMA) models are used to model a special class of nonstationary series. An ARIMA(p, d, q) (d non-negative integer) model is defined by

$$(1 - B)^d \phi(B) X_t = \theta(B) Z_t.$$

Seasonal models are used to incorporate cyclic components in models. A class of commonly encountered seasonal models is that of seasonal ARIMA (SARIMA) models. A SARIMA(p, d, q)(P, D, Q)$_s$ model (d and D are non-negative integers and s is the period) is defined by

$$(1 - B)^d (1 - B^s)^D \phi(B) \Phi(B^s) X_t = \theta(B) \Theta(B^s) Z_t$$

where $\Phi(x)$ and $\Theta(x)$ are polynomials that describe the seasonal part of the process.

The commonly used time series models are represented in this package by objects of the generic form `model[`*param*$_1$`,` *param*$_2$`, ...]`. Each of these objects serves to specify a particular model and does not itself evaluate to anything. They can be entered as arguments of time series functions as well as generated as output.

`ARModel[`*philist*`, `σ^2`]`	AR(p) model with p AR coefficients in *philist* and normally distributed noise with variance σ^2
`MAModel[`*thetalist*`, `σ^2`]`	MA(q) model with q MA coefficients in *thetalist* and normally distributed noise with variance σ^2
`ARMAModel[`*philist, thetalist*`, `σ^2`]`	ARMA(p, q) model with p AR and q MA coefficients in *philist* and *thetalist*, respectively, and normally distributed noise with variance σ^2
`ARIMAModel[`*d, philist*`, `*thetalist*`, `σ^2`]`	ARIMA(p, d, q) model with p AR and q MA coefficients in *philist* and *thetalist*, respectively, and normally distributed noise with variance σ^2
`SARIMAModel[{`*d*`, `*D*`}, `*s*`, `*philist*`, `*Philist*`, `*thetalist*`, `*Thetalist*`, `σ^2`]`	SARIMA(p, d, q)(P, D, Q)$_s$ model with p regular and P seasonal AR coefficients in *philist* and *Philist* and q regular and Q seasonal MA coefficients in *thetalist* and *Thetalist*, respectively, and normally distributed noise with variance σ^2

Time series models.

Here, when *model* is used as a *Mathematica* function argument it means the model object defined above. The notation *philist* denotes a list of AR coefficients $\{\phi_1, \phi_2, \ldots, \phi_p\}$, *thetalist* specifies a list of MA coefficients $\{\theta_1, \theta_2, \ldots, \theta_q\}$, and so on. The noise is zero-mean Gaussian white noise, and its variance or covariance matrix will be called the noise parameter. d (or D) is the order of differencing, and s is the seasonal period. To extract any of these arguments from a *model*, we can use the function `Part` or one of the following functions.

`ARCoefficients[`*model*`]`	extract the AR coefficients of *model*
`MACoefficients[`*model*`]`	extract the MA coefficients of *model*
`SeasonalARCoefficients[`*model*`]`	extract the seasonal AR coefficients of *model*
`SeasonalMACoefficients[`*model*`]`	extract the seasonal MA coefficients of *model*
`SeasonalPeriod[`*model*`]`	extract the seasonal period of *model*
`DifferencingOrder[`*model*`]`	extract the differencing order of *model*
`NoiseParameter[`*model*`]`	extract the noise parameter of *model*

Functions extracting the parameters of time series models.

All of the functions in this package with the exception of `AsymptoticCovariance` and the functions analyzing ARCH models work for both univariate and multivariate cases, although some are illustrated using examples of univariate series only.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

`ARMAModel[{0.5, -0.2}, {0.7}, 0.8]` represents the ARMA(2, 1) model $X_t - 0.5\,X_{t-1} + 0.2\,X_{t-2} = Z_t + 0.7\,Z_{t-1}$ with noise variance 0.8. Note that it does not evaluate to anything.

```
In[2]:= ARMAModel[{0.5, -0.2}, {0.7}, 0.8]

Out[2]= ARMAModel[{0.5, -0.2}, {0.7}, 0.8]
```

We can extract the MA coefficient using `MACoefficients`.

```
In[3]:= MACoefficients[%]

Out[3]= {0.7}
```

We can also use the function `Part` to extract the argument.

```
In[4]:= %%[[2]]

Out[4]= {0.7}
```

This is a SARIMA(2, 2, 1)(1, 3, 0)$_4$ model. Note that the absence of the seasonal MA part is represented by *thetalist* = { } (or *thetalist* = {0}).

```
In[5]:= SARIMAModel[{2, 3}, 4, {0.4, 0.1}, {-0.5}, {0.8}, {}, 1]

Out[5]= SARIMAModel[{2, 3}, 4, {0.4, 0.1}, {-0.5}, {0.8}, {}, 1]
```

This is how we get the differencing orders of both the seasonal and ordinary parts.

```
In[6]:= DifferencingOrder[%]

Out[6]= {2, 3}
```

`ToARModel[`*model*`, `*n*`]`	give the AR(*n*) model that is the order *n* truncation of the AR(∞) expansion of *model*
`ToMAModel[`*model*`, `*n*`]`	give the MA(*n*) model that is the order *n* truncation of the MA(∞) expansion of *model*
`ToARMAModel[`*model*`]`	convert an ARIMA or a SARIMA *model* to an ARMA *model*

Conversions of time series models.

This computes the first four coefficients in the expansion of an AR(1) model as an MA(∞) model and returns the corresponding MA(4) model.

```
In[7]:= ToMAModel[ARModel[{ϕ1}, σ^2], 4]

Out[7]= MAModel[{ϕ1, ϕ1^2, ϕ1^3, ϕ1^4}, σ^2]
```

The first four coefficients in the expansion of an ARMA(1, 1) model as an AR(∞) model are calculated and the corresponding AR(4) model is returned.

```
In[8]:= ToARModel[ARMAModel[{ϕ1}, {θ1}, σ^2], 4]

Out[8]= ARModel[{θ1 + ϕ1, θ1 (-θ1 - ϕ1), -θ1^2 (-θ1 - ϕ1), θ1^3 (-θ1 - ϕ1)}, σ^2]
```

This computes the first three coefficients in the expansion of a vector AR(2) model as an MA(∞) model and returns the corresponding vector MA(3) model.

```
In[9]:= ToMAModel[ARModel[
          {{{0.2, -0.3}, {0.2, -0.7}}, {{0.1, 0.7}, {0.3, -0.8}}}, {{1, 0}, {0, 1}}], 3]

Out[9]= MAModel[{{{0.2, -0.3}, {0.2, -0.7}}, {{0.08, 0.85}, {0.2, -0.37}},
          {{0.116, -0.239}, {-0.224, 0.899}}}, {{1, 0}, {0, 1}}]
```

This converts an ARIMA(2, 2, 1) into the equivalent ARMA(4, 1) model.

```
In[10]:= ToARMAModel[ARIMAModel[2, {0.3, 0.9}, {1.2}, 1]]

Out[10]= ARMAModel[{2.3, -0.7, -1.5, 0.9}, {1.2}, 1]
```

This converts a SARIMA(1, 2, 1) $(1, 1, 1)_2$ model into the equivalent ARMA(7, 3) model.

```
In[11]:= ToARMAModel[SARIMAModel[{2, 1}, 2, {0.5}, {-0.7}, {0.4}, {0.3}, 1]]

Out[11]= ARMAModel[{2.5, -1.7, -0.25, 1.3, -1.9, 1.4, -0.35}, {0.4, 0.3, 0.12}, 1]
```

Given a model, its covariance function and correlation function can be calculated. For a univariate zero-mean, stationary process, the covariance function at lag h is defined by $\gamma(h) = E(X_{t+h}\, X_t)$ (E denotes the expectation); the correlation function, by $\rho(h) = \gamma(h)/\gamma(0)$. For a multivariate process, the matrix covariance function is defined by $\Gamma(h) = E\left(X_{t+h}\, X_t'\right)$ (X' denotes the transpose of X), and the matrix correlation function at lag h, $R(h)$, is given by $R((h))_{ij} = \rho_{ij}(h) = \gamma_{ij}(h) \Big/ \sqrt{\gamma_{ii}(0)\,\gamma_{jj}(0)}$ where $\gamma_{ij}(h) = \Gamma((h))_{ij}$. Another useful function is the partial correlation function. The partial correlation function at lag k, $\phi_{k,k}$, is defined to be the correlation between X_{t+k} and X_t with all intervening variables fixed. The power spectrum of a univariate ARMA(p, q) process is given by $\frac{\sigma^2}{2\pi} \left|\theta\left(e^{-i\,\omega}\right)\right|^2 \Big/ \left|\phi\left(e^{-i\,\omega}\right)\right|^2$, and for a multivariate process the power spectrum is $\frac{1}{2\pi}\, \phi^{-1}\left(e^{-i\,\omega}\right) \theta\left(e^{-i\,\omega}\right) \Sigma\theta'\left(e^{i\,\omega}\right) \phi'^{-1}\left(e^{i\,\omega}\right)$.

`StationaryQ[`*model*`]`	give `True` if *model* is stationary, `False` otherwise
`StationaryQ[`*philist*`]`	give `True` if the model with its AR coefficients in *philist* is stationary, `False` otherwise
`InvertibleQ[`*model*`]`	give `True` if *model* is invertible, `False` otherwise
`InvertibleQ[`*thetalist*`]`	give `True` if the model with its MA coefficients in *thetalist* is invertible, `False` otherwise
`CovarianceFunction[`*model*`, n]`	give the covariance function of *model* up to lag *n*
`CorrelationFunction[`*model*`, n]`	give the correlation function of *model* up to lag *n*
`PartialCorrelationFunction[` *model*`, n]`	give the partial correlation function of *model* up to lag *n*
`Spectrum[`*model*`, ω]`	give the power spectrum of *model* as a function of ω

Properties of time series models.

This time series model has the AR polynomial $\phi(x) = 1 - 0.5\,x + 0.2\,x^2 - 1.5\,x^3$. $\phi(x) = 0$ has roots inside the unit circle, so the process is not stationary.

```
In[12]:= StationaryQ[ARMAModel[{0.5, -0.2, 1.5}, {0.2}, 1]]

Out[12]= False
```

We can also just enter the AR coefficients to check stationarity.

```
In[13]:= StationaryQ[{0.5, -0.2, 1.5}]

Out[13]= False
```

These vector AR coefficients indicate a stationary process. The noise covariance can be omitted as we have done here since it is irrelevant for determining stationarity.

```
In[14]:= StationaryQ[ARModel[{{{0.2, -0.3}, {0.2, -0.7}}, {{0.1, 0.7}, {0.3, -0.8}}}]]

Out[14]= True
```

A given time series model has the MA polynomial $\theta(x) = 1 + 0.3\,x - 0.1\,x^2$. The roots of $\theta(x) = 0$ are all outside the unit circle, so the process is invertible.

```
In[15]:= InvertibleQ[ARMAModel[{0.5}, {0.3, -0.1}]]

Out[15]= True
```

This gives the covariance function $\gamma(h)$ of an MA(1) model for $h = 0, 1, 2, 3$. Note that for an MA(q) model $\gamma(h) = 0$ for $h > q$.

```
In[16]:= CovarianceFunction[MAModel[{θ1}, σ^2], 3]
```

$$\text{Out[16]= } \left\{\sigma^2 \left(1 + \theta_1^2\right),\ \sigma^2 \theta_1,\ 0,\ 0\right\}$$

This yields the covariance function of a vector ARMA(1, 1) model up to lag 2.

```
In[17]:= CovarianceFunction[ARMAModel[{{{0.2, -1.3}, {0.2, -0.7}}},
           {{{0.3, 0.5}, {0.2, -0.5}}}, {{1.2, 0.1}, {0.1, 0.8}}], 2]
```

```
Out[17]= {{{3.90967, 1.91333}, {1.91333, 2.46403}},
          {{-1.29539, -2.39058}, {-0.367393, -1.72216}},
          {{0.218534, 1.76069}, {-0.00190227, 0.727395}}}
```

The correlation function of an AR(1) model is calculated up to lag 5.

```
In[18]:= CorrelationFunction[ARModel[{ϕ1}, σ^2], 5]
```

$$\text{Out[18]= } \left\{1,\ \phi_1,\ \phi_1^2,\ \phi_1^3,\ \phi_1^4,\ \phi_1^5\right\}$$

This gives the partial correlation function $\phi_{k,k}$ of an AR(2) model at lags $k = 1, 2, 3$, and 4. Note that the partial correlation function starts at lag 1, and for an AR(p) process, $\phi_{k,k} = 0$ for $k > p$.

```
In[19]:= PartialCorrelationFunction[ARModel[{0.5, -0.2}, 1], 4]
```

```
Out[19]= {0.416667, -0.2, 0, 0}
```

Here is the power spectrum of an ARMA(1, 1) model as a function of ω.

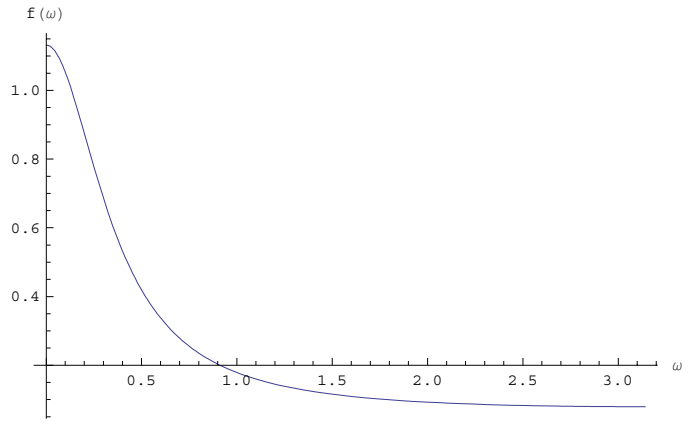
```
In[20]:= Spectrum[ARMAModel[{ϕ1}, {θ1}, 1], ω]
```

$$\text{Out[20]= } \frac{1 + 2\,\text{Cos}[\omega]\,\theta_1 + \theta_1^2}{2\,\pi\left(1 - 2\,\text{Cos}[\omega]\,\phi_1 + \phi_1^2\right)}$$

This yields the plot of the spectrum calculated above with $\phi_1 = 0.7$ and $\theta_1 = -0.2$.

```
In[21]:= Plot[% /. {ϕ1 -> 0.7, θ1 -> -0.2},
          {ω, 0, Pi}, AxesLabel -> {"ω", "f(ω)"}, PlotRange -> All]
```

Out[21]=



`RandomSequence`$[\mu, \sigma^2, n]$	generate a random sequence of length n distributed normally with mean μ and variance σ^2
`TimeSeries`$[model, n]$	generate a time series of length n from *model*
`TimeSeries`$[model, n, \{x_{-p+1}, x_{-p+2}, \ldots, x_0\}]$	generate a time series with p given initial values
`TimeSeries`$[model, n, \{z_{-q+1}, z_{-q+2}, \ldots, z_n\}, \{x_{-p+1}, x_{-p+2}, \ldots, x_0\}]$	generate a time series with a given noise sequence $\{z\}$ and initial values $\{x\}$

Generating random sequences and time series.

Both `RandomSequence` and `TimeSeries` uses the current default random number generator. Sequences generated prior to Version 6.0 of *Mathematica* can be obtained by using the legacy generator via `SeedRandom[Method -> Legacy]` or `SeedRandom[`*seed*`, Method->"Legacy"]`.

A sequence of 5 random numbers is generated. They are normally distributed with mean 0 and variance 1.2.

```
In[22]:= RandomSequence[0, 1.2, 5]

Out[22]= {-0.0561913, 0.687886, -1.96931, 1.00539, -0.363847}
```

This generates a sequence of 4 random vectors distributed normally with mean $\{0.5, 0.7\}$ and covariance matrix $\{\{0.9, -0.2\}, \{-0.2, 1.0\}\}$. Note that we have used `SeedRandom[1039]` to start the random number generator with seed 1039.

```
In[23]:= (SeedRandom[1039];
          RandomSequence[{0.5, 0.7}, {{0.9, -0.2}, {-0.2, 1.0}}, 4])

Out[23]= {{-0.612201, 1.21709}, {1.96981, 1.54269},
          {-0.0190734, 0.737721}, {2.14867, 0.760779}}
```

Here is a scalar time series of length 7 generated from the ARMA(2, 1) model $X_t - 0.5\,X_{t-1} + 0.2\,X_{t-2} = Z_t + 0.7\,Z_{t-1}$. The normally distributed noise has a mean 0 and a variance of 0.8.

```
In[24]:= (SeedRandom[1234];
          TimeSeries[ARMAModel[{0.5, -0.2}, {0.7}, 0.8], 7])

Out[24]= {0.755487, -0.12346, -0.115633, 0.825265, 0.939603, -0.0773533, -0.575972}
```

This generates a time series $\{x_1, x_2, \ldots, x_7\}$ from the same model as above with given values of $x_{-1} = -0.3$, and $x_0 = 0.5$.

```
In[25]:= (SeedRandom[1234];
          TimeSeries[ARMAModel[{0.5, -0.2}, {0.7}, 0.8],
         7, {-0.3, 0.5}])

Out[25]= {1.0077, -0.188289, -0.198489, 0.796802, 0.941943, -0.0704908, -0.573008}
```

Here is a vector time series of length 5 generated from a vector MA(2) model. Note that the MA parameters and the noise covariance are now matrices.

```
In[26]:= (SeedRandom[1673];
          TimeSeries[MAModel[{{{0.1, -0.2}, {-0.2, 0.1}},
         {{0.7, -1.2}, {0.3, -0.2}}}, {{1, 0}, {0, 1}}], 5])

Out[26]= {{2.28449, -0.659337}, {-0.945795, 0.0369965},
          {0.298284, -0.210925}, {-0.0754738, -1.0219}, {-0.129903, -1.41094}}
```

This generates a scalar time series of length 8 from the SARIMA(2, 2, 2)(2, 1, 1)$_2$ model $(1 - B)^2\,(1 - B^2)\,(0.2\,B^2 - 0.5\,B + 1)\,(0.1\,B^4 - 0.3\,B^2 + 1)\,X_t = (1 + 0.6\,B - 0.1\,B^2)\,(1 - 0.5\,B^2)\,Z_t$ with noise variance 0.9.

```
In[27]:= (SeedRandom[1234];
          TimeSeries[
           SARIMAModel[{2, 1}, 2, {0.5, -0.2}, {0.3, -0.1}, {0.6, -0.1}, {-0.5}, 0.9], 8])

Out[27]= {-7.69176, -9.38373, -17.4059, -18.9872, -26.734, -28.3097, -36.5849, -37.9153}
```

2.2 Analysis of ARMA Time Series

Given a time series, we can analyze its properties. Since many algorithms for estimating model parameters assume the data form a zero-mean, stationary process, appropriate transformations of the data are often needed to make them zero-mean and stationary. The function `ListDifference` does appropriate differencing on ARIMA or SARIMA data. Note that all time series data should be input as a list of the form $\{x_1, x_2, \dots\}$ where x_i is a number for a scalar time series and is itself a list, $x_i = \{x_{i1}, x_{i2}, \dots, x_{im}\}$, for an m-variate time series.

`ListDifference[`*data*`, `*d*`]`	difference *data* *d* times
`ListDifference[`*data*`, {`*d*`, `*D*`}, `*s*`]`	difference *data* *d* times with period 1 and *D* times with period *s*

Sample mean and transformations of data.

After appropriate transformations of the time series data, we can calculate the properties of the series. The sample covariance function $\hat{\gamma}(h)$ for a zero-mean time series of n observations is defined to be $\frac{1}{n}\sum_{t=1}^{n-h} X_{t+h} X_t'$. The sample correlation function $\hat{\rho}(h)$ is the sample covariance function of the corresponding standardized series, and the sample partial autocorrelation function is defined here as the last coefficient in a Levinson-Durbin estimate of AR coefficients. The sample power spectrum $\hat{f}(\omega)$ is the Fourier transform of the sample covariance function $\hat{f}(\omega) = \frac{1}{2\pi}\sum_{k=-(n-1)}^{n-1} \hat{\gamma}(k)\, e^{-ik\omega}$. The smoothed spectrum $\hat{f}_S$ using the spectral window $\{W(0), W(1), \dots, W(M)\}$ is defined to be $\hat{f}_S(\omega_j) = \sum_{k=-M}^{M} W(k)\hat{f}(\omega_{j-k})$ where $W(k) = W(-k)$, while the smoothed spectrum $\hat{f}_L$ using the lag window $\{\lambda(0), \lambda(1), \dots, \lambda(M)\}$ is defined by $\hat{f}_L(\omega) = \frac{1}{2\pi}\sum_{k=-M}^{M} \lambda(k)\hat{\gamma}(k)\, e^{-ik\omega}$ where $\lambda(k) = \lambda(-k)$.

`CovarianceFunction[`*data*`, `*n*`]` `CovarianceFunction[`*data*$_1$`, `*data*$_2$`, `*n*`]`	give the sample covariance function of *data* or the cross-covariance function of *data*$_1$ and *data*$_2$ up to lag *n*
`CorrelationFunction[`*data*`, `*n*`]` `CorrelationFunction[`*data*$_1$`, `*data*$_2$`, `*n*`]`	give the sample correlation function of *data* or the cross-correlation function of *data*$_1$ and *data*$_2$ up to lag *n*
`PartialCorrelationFunction[`*data*`, `*n*`]`	give the sample partial correlation function of *data* up to lag *n*
`Spectrum[`*data*`]` `Spectrum[`*data*$_1$`, `*data*$_2$`]`	give the sample power spectrum of *data* or the cross-spectrum of *data*$_1$ and *data*$_2$
`SmoothedSpectrumS[`*spectrum*`, `*window*`]`	give the smoothed spectrum using the spectral window *window*
`SmoothedSpectrumL[`*cov*`, `*window*`, `ω`]`	give the smoothed spectrum as a function of ω using the lag window *window*

Properties of observed data.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

The increasing magnitude of the data indicates a nonstationary series. We know this is the case, since it is generated from an ARIMA(1, 2, 0) process. Note that since the model has no MA part, the list of the MA parameters can be specified by { } or by {{0, 0},{0, 0}}.

```
In[2]:= (SeedRandom[3456];
         TimeSeries[ARIMAModel[2, {{{0.3, -0.1}, {0.4, 0.5}}}, {}, {{1, 0}, {0, 1}}], 10])

Out[2]= {{-5.09594, -4.22676}, {-8.62944, -9.74584}, {-9.73182, -15.3683},
         {-9.95509, -21.2617}, {-8.29291, -27.4899}, {-7.038, -32.8041},
         {-4.27451, -37.8621}, {-1.06048, -38.8798}, {2.15982, -38.9356}, {4.74157, -38.4119}}
```

Differencing the above time series twice yields a new series that is stationary.

```
In[3]:= ListDifference[%, 2]

Out[3]= {{2.43112, -0.103382}, {0.879105, -0.270887},
         {1.88546, -0.334898}, {-0.407277, 0.914094}, {1.50858, 0.256116},
         {0.450541, 4.04031}, {0.00627729, 0.961922}, {-0.638555, 0.57953}}
```

A SARIMA(2, 2, 0)$(1, 1, 0)_2$ series is transformed into a stationary series by differencing.

```
In[4]:= (SeedRandom[123];
         tseries =
          TimeSeries[SARIMAModel[{2, 1}, 2, {0.5, -0.2}, {0.5}, {0}, {0}, 1], 10];
         ListDifference[tseries, {2, 1}, 2])

Out[4]= {-0.201361, -2.67625, -1.95732, -2.6338, -1.6277, -1.1512}
```

This generates a time series of length 200 from an AR(2) model.

```
In[5]:= (SeedRandom[1234];
         tseries = TimeSeries[ARModel[{0.5, -0.2}, 0.75], 200];)
```

This computes the sample covariance function of the above series up to lag 5.

```
In[6]:= CovarianceFunction[tseries, 5]

Out[6]= {0.75797, 0.33583, 0.0458689, -0.0248131, -0.0930331, -0.0327725}
```

Here is the sample correlation function of the same series up to lag 5. Note that the correlation at lag 0 is 1.

```
In[7]:= CorrelationFunction[tseries, 5]

Out[7]= {1., 0.443065, 0.0605154, -0.0327362, -0.12274, -0.0432372}
```

This is the sample partial correlation function of the same series at lags 1 to 5. Note that the lag starts from 1 and not 0.

```
In[8]:= PartialCorrelationFunction[tseries, 5]

Out[8]= {0.443065, -0.168959, 0.0138086, -0.129604, 0.0870806}
```

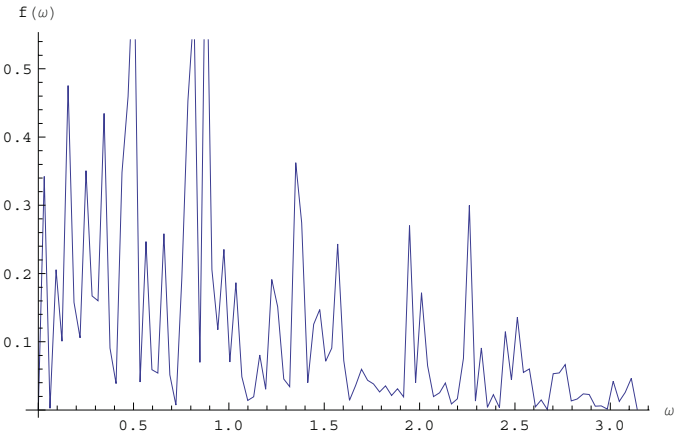
The power spectrum of the series is calculated here.

```
In[9]:= spec = Spectrum[tseries];
```

This is the plot of the spectrum in the frequency range $[0, \pi]$.

```
In[10]:= ListLinePlot[Transpose[{Table[2 Pi / 200 * k, {k, 0, 100}], Take[spec, 101]}],
          AxesLabel -> {"ω", "f(ω)"}]
```

Out[10]=



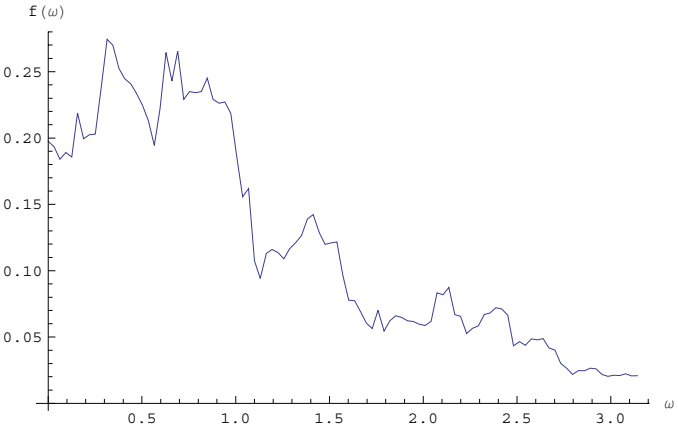
This gives the smoothed spectrum using the Daniell spectral window.

```
In[11]:= SmoothedSpectrumS[spec, Table[1 / 13, {7}]];
```

We plot the smoothed spectrum.

```
In[12]:= ListLinePlot[Transpose[{Table[2 Pi / 200 * k, {k, 0, 100}], Drop[%, -99]}],
          AxesLabel -> {"ω", "f(ω)"}]
```

Out[12]=



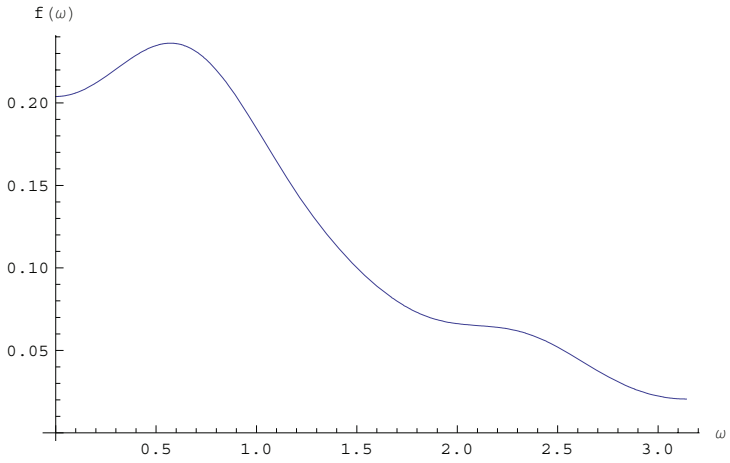
This yields the smoothed spectrum using the Hanning lag window.

```
In[13]:= SmoothedSpectrumL[CovarianceFunction[tseries, 10],
           Table[0.5 + 0.5 Cos[Pi k / 10], {k, 0, 10}], ω];
```

This is the plot of the smoothed spectrum.

```
In[14]:= Plot[%, {ω, 0, Pi}, AxesLabel -> {"ω", "f(ω)"}]
```

Out[14]=



We calculate the correlation function up to lag 2 from data generated from a vector MA(1) model.

```
In[15]:= (tseries = TimeSeries[MAModel[{{{0.5, -0.1}, {0.4, 1.2}}}, {{1, 0}, {0, 1}}], 50];
          CorrelationFunction[tseries, 2])

Out[15]= {{{1., -0.28224}, {-0.28224, 1.}}, {{0.309885, -0.300356}, {-0.109964, 0.522227}},
          {{0.00525678, -0.142099}, {-0.187691, 0.000553525}}}
```

When fitting a given set of data to a particular ARMA type of model, the orders have to be selected first. Usually the sample partial correlation or sample correlation function can give an indication of the order of an AR or an MA process. An AIC or a BIC criterion is also used to select a model. The AIC criterion chooses p and q to minimize the value of $\ln \hat{\sigma}^2_{p,q} + 2(p+q)/n$ by fitting the time series of length n to an ARMA(p, q) model (here $\hat{\sigma}^2_{p,q}$ is the noise variance estimate, usually found via maximum likelihood estimation). The BIC criterion seeks to minimize $\ln \hat{\sigma}^2_{p,q} + (p+q)\ln n / n$.

AIC[*model*, *n*]	give the AIC value of *model* fitted to data of length *n*
BIC[*model*, *n*]	give the BIC value of *model* fitted to data of length *n*

AIC and BIC values.

Given a model, various methods exist to fit the appropriately transformed data to it and estimate the parameters. HannanRissanenEstimate uses the Hannan-Rissanen procedure to both select orders and perform parameter estimation. As in the long AR method, the data are first fitted to an AR(k) process, where k (less than some given *kmax*) is chosen by the AIC criterion. The orders p and q are selected among all $p \le$ Min[pmax, k] and $q \le$ *qmax* using BIC.

`YuleWalkerEstimate[data, p]`	give the Yule-Walker estimate of AR(p) model
`LevinsonDurbinEstimate[data, p]`	give the Levinson-Durbin estimate of AR(i) model for $i = 1, 2, \ldots, p$
`BurgEstimate[data, p]`	give the Burg estimate of AR(i) model for $i = 1, 2, \ldots, p$
`InnovationEstimate[data, q]`	give the innovation estimate of MA(i) model for $i = 1, 2, \ldots, q$
`LongAREstimate[data, k, p, q]`	give the estimate of ARMA(p, q) model by first finding the residuals from AR(k) process
`HannanRissanenEstimate[data, kmax, pmax, qmax]`	give the estimate of the model with the lowest BIC value
`HannanRissanenEstimate[data, kmax, pmax, qmax, n]`	give the estimate of the n models with the lowest BIC values

Estimations of ARMA models.

This generates a vector AR(2) series.

```
In[16]:= (SeedRandom[123];
          newseries =
           TimeSeries[ARModel[{{{0.3, -0.1}, {0.4, 0.2}}, {{-0.5, 0.1}, {-0.2, 0.7}}},
             {{1, 0}, {0, 1}}], 200];)
```

Observe how the vector time series data are defined and input into functions below.

```
In[17]:= Short[newseries, 4]

Out[17]//Short=
     {{2.26787, 0.019026}, {1.51063, 2.42142}, <<197>>, {0.844117, -0.226315}}
```

The Yule-Walker estimate of the coefficients of the AR(2) series and the noise variance are returned inside the object `ARModel`.

```
In[18]:= armodel = YuleWalkerEstimate[newseries, 2]

Out[18]= ARModel[{{{0.255218, -0.0108376}, {0.439688, 0.218125}},
           {{-0.453402, -0.0190763}, {-0.0559706, 0.712374}}},
          {{0.962922, 0.118976}, {0.118976, 0.828492}}]
```

Here the AR(p) ($p \le 3$) coefficients and noise variance are estimated using the Levinson-Durbin algorithm. Note that the small magnitude of ϕ_3 for AR(3) indicates that $p = 2$ is the likely order of the process.

```
In[19]:= (SeedRandom[123];
          tseries = TimeSeries[ARModel[{0.5, -0.2}, 1], 200];
          LevinsonDurbinEstimate[tseries, 3])

Out[19]= {ARModel[{0.410985}, 0.968709], ARModel[{0.506934, -0.233462}, 0.91591],
          ARModel[{0.486158, -0.188349, -0.0889921}, 0.908656]}
```

The same parameters are estimated using the Burg algorithm.

```
In[20]:= BurgEstimate[tseries, 3]

Out[20]= {ARModel[{0.426314}, 0.971849], ARModel[{0.523447, -0.227845}, 0.921397],
          ARModel[{0.506452, -0.1888, -0.0745929}, 0.91627]}
```

This gives the estimate of MA(q) ($q \leq 4$) coefficients and noise variance using the innovations algorithm.

```
In[21]:= (SeedRandom[123];
          tseries = TimeSeries[MAModel[{0.5, -0.2}, 1], 200];
          InnovationEstimate[tseries, 4])

Out[21]= {MAModel[{0.306685}, 1.08259], MAModel[{0.389053, -0.14926}, 1.0045],
          MAModel[{0.401514, -0.130418, -0.101435}, 1.00234],
          MAModel[{0.408681, -0.16681, -0.0736189, -0.113278}, 0.978422]}
```

This estimates the same parameters using the long AR method. The data are first fitted to an AR(10) model. The residuals together with the data are fitted to an MA(2) process using regression.

```
In[22]:= LongAREstimate[tseries, 10, 0, 2]

Out[22]= MAModel[{0.516564, -0.197653}, 0.905695]
```

Here the parameters of the ARMA(1,2) model are estimated using the long AR method.

```
In[23]:= (SeedRandom[123];
          tseries = TimeSeries[ARMAModel[{0.2}, {0.2, -0.5}, 2], 250];
          LongAREstimate[tseries, 8, 1, 2])

Out[23]= ARMAModel[{0.399529}, {0.00710284, -0.580108}, 1.79702]
```

This calculates the AIC value for the model estimated above.

```
In[24]:= AIC[%, 250]

Out[24]= 0.610128
```

The Hannan-Rissanen method can select the model orders as well as estimate the parameters. Often the order selection is only suggestive and it should be used in conjunction with other methods.

```
In[25]:= HannanRissanenEstimate[tseries, 8, 3, 3]

Out[25]= ARMAModel[{0.360899}, {0.0449159, -0.560618}, 1.80737]
```

Here we select three models using the Hannan-Rissanen method.

```
In[26]:= HannanRissanenEstimate[tseries, 8, 3, 3, 3]

Out[26]= {ARMAModel[{0.360899}, {0.0449159, -0.560618}, 1.80737],
          MAModel[{0.405558, -0.440279}, 1.8654],
          MAModel[{0.398491, -0.431242, -0.155303}, 1.82901]}
```

This gives the BIC value for each model estimated above.

```
In[27]:= BIC[#, 250] & /@ %

Out[27]= {0.658128, 0.667646, 0.670034}
```

`MLEstimate` gives the maximum likelihood estimate of an ARMA type of model by maximizing the exact likelihood function that is calculated using the innovations algorithm. The built-in function `FindMinimum` is used and the same options apply. Two sets of initial values are needed for each parameter, and they are usually taken from the results of various estimation methods given above. Since finding the exact maximum likelihood estimate is generally slow, a conditional likelihood is often used. `ConditionalMLEstimate` gives an estimate of an ARMA model by maximizing the conditional likelihood using the Levenberg-Marquardt algorithm.

`ConditionalMLEstimate[`*data*`, `*p*`]`	fit an AR(*p*) model to *data* using the conditional maximum likelihood method
`ConditionalMLEstimate[`*data*`, `*model*`]`	fit *model* to *data* using the conditional maximum likelihood method with initial values of parameters as the arguments of *model*
`MLEstimate[`*data*`, `*model*`, {`$\phi 1$`, {`$\phi 1_1$`, `$\phi 1_2$`}}, ...]`	fit *model* to *data* using maximum likelihood method with initial values of parameters $\{\phi 1_1, \phi 1_2\}$, …
`LogLikelihood[`*data*`, `*model*`]`	give the logarithm of Gaussian likelihood for the given *data* and *model*

Maximum likelihood estimations and the logarithm of Gaussian likelihood.

option name	*default value*	
`MaxIterations`	`30`	maximum number of iterations in searching for minimum

Option for `ConditionalMLEstimate`.

A vector AR(1) series of length 200 is generated.

```
In[28]:= (SeedRandom[1234];
          TimeSeries[ARModel[{{{0.3, -0.1}, {0.4, 0.5}}}, {{1, 0}, {0, 1}}], 200];)
```

This yields the conditional maximum likelihood estimate of the parameters of a vector AR process. In the absence of an MA part, this estimate is equivalent to a least squares estimate and no initial values for parameters are needed.

```
In[29]:= ConditionalMLEstimate[%, 1]

Out[29]= ARModel[{{{0.29656, -0.144169}, {0.379962, 0.428523}}},
           {{0.848146, -0.0439261}, {-0.0439261, 0.849261}}]
```

This gives the conditional maximum likelihood estimate of an ARMA(2,1) model. The initial parameter values ($\phi_1 = 0.4$, $\phi_2 = -0.35$, $\theta_1 = 0.6$) have to be provided as the arguments of `ARMAModel`.

```
In[30]:= (SeedRandom[9283];
          ts1 = TimeSeries[ARMAModel[{0.7, -0.3}, {0.5}, 1], 100];
          armamodel = ConditionalMLEstimate[ts1, ARMAModel[{0.4, -0.35}, {0.6}, 1]])

Out[30]= ARMAModel[{0.549703, -0.342903}, {0.628324}, 0.711585]
```

The Hannan-Rissanen method is used to select a model. The estimated model parameters can serve as the initial values for parameters in maximum likelihood estimation.

```
In[31]:= (SeedRandom[9381];
          tseries =
           TimeSeries[MAModel[{{{0.2, -0.7}, {-0.2, 0.9}}}, {{1, 0}, {0, 1}}], 100];
          HannanRissanenEstimate[tseries, 10, 3, 3])

Out[31]= MAModel[{{{0.102844, -0.543006}, {-0.0697756, 0.640399}}},
          {{1.23301, -0.314665}, {-0.314665, 1.40438}}]
```

The above result is input here for the conditional maximum likelihood estimate.

```
In[32]:= ConditionalMLEstimate[tseries, %]

Out[32]= MAModel[{{{0.274936, -0.755129}, {-0.180682, 0.899108}}},
          {{0.944411, -0.0383107}, {-0.0383107, 1.08534}}]
```

A SARIMA$(1, 0, 1)(0, 0, 1)_2$ series is generated.

```
In[33]:= (SeedRandom[2958];
          tseries = TimeSeries[SARIMAModel[{0, 0}, 2, {-0.5}, {0}, {-0.3}, {0.8}, 1], 50];)
```

This yields the maximum likelihood estimate of the parameters of a SARIMA$(1, 0, 1)(0, 0, 1)_2$ model. Note that the parameters to be estimated are entered symbolically inside *model* and two initial values are needed for each of them. Since the calculation of the likelihood of a univariate series is independent of the noise variance, it should not be entered in symbolic form.

```
In[34]:= sarima = MLEstimate[tseries, SARIMAModel[{0, 0}, 2, {ϕ1}, {0}, {Θ1}, {Θ1}, 1],
            {ϕ1, {-0.4, -0.41}}, {Θ1, {-0.2, -0.22}}, {Θ1, {0.7, 0.75}}]

Out[34]= SARIMAModel[{0, 0}, 2, {-0.740659}, {0}, {-0.300039}, {0.524104}, 0.854466]
```

A bivariate MA(1) series is generated.

```
In[35]:= (SeedRandom[4567];
          ts = TimeSeries[MAModel[{{{0.5, 0.2}, {-0.1, 0.4}}}, {{1, 0}, {0, 1}}], 50];)
```

For a vector ARMA series, the calculation of maximum likelihood is not independent of the covariance matrix; the covariance matrix has to be input as symbolic parameters.

```
In[36]:= MLEstimate[ts, MAModel[{{{t1, t2}, {t3, t4}}}, {{w1, w2}, {w2, w3}}],
          {t1, {0.4, 0.45}}, {t2, {0.15, 0.16}}, {t3, {-0.09, -0.1}},
          {t4, {0.3, 0.35}}, {w1, {1, 0.9}}, {w2, {0, 0.01}}, {w3, {0.9, 1}}]

Out[36]= MAModel[{{{0.680734, 0.226766}, {-0.143536, 0.281516}}},
          {{0.743654, -0.0202879}, {-0.0202879, 0.734156}}]
```

This gives the logarithm of the Gaussian likelihood.

```
In[37]:= LogLikelihood[ts, %]

Out[37]= -35.2266
```

Let $\beta = (\phi_1, \phi_2, \ldots, \phi_p, \theta_1, \theta_2, \ldots, \theta_q)'$ be the parameters of a stationary and invertible ARMA(p, q) model and $\hat{\beta}$ the maximum likelihood estimator of β. Then, as $n \to \infty$, we have $n^{1/2}(\hat{\beta} - \beta) \Rightarrow N(\mathbf{0}, V(\beta))$. For a univariate ARMA model, `AsymptoticCovariance[`*model*`]` calculates the asymptotic covariance V from *model*. The function `InformationMatrix[`*data*`,` *model*`]` gives the estimated asymptotic information matrix whose inverse can be used as the estimate for the asymptotic covariance.

`AsymptoticCovariance[`*model*`]`	give the covariance matrix V of the asymptotic distribution of the maximum likelihood estimators
`InformationMatrix[`*data*`,` *model*`]`	give the estimated asymptotic information matrix

Asymptotic covariance and information matrix.

This gives the asymptotic covariance matrix of the estimators of an MA(3) model.

```
In[38]:= AsymptoticCovariance[MAModel[{θ1, θ2, θ3}, σ^2]];
```

The above asymptotic covariance is displayed in matrix form.

```
In[39]:= MatrixForm[%]
```

```
Out[39]//MatrixForm=
⎛ 1 - θ3^2          θ1 - θ2 θ3          θ2 - θ1 θ3        ⎞
⎜ θ1 - θ2 θ3        1 + θ1^2 - θ2^2 - θ3^2   θ1 - θ2 θ3   ⎟
⎝ θ2 - θ1 θ3        θ1 - θ2 θ3          1 - θ3^2          ⎠
```

This gives the estimate of the information matrix.

```
In[40]:= InformationMatrix[ts1, armamodel];
```

The above information matrix is displayed here.

```
In[41]:= MatrixForm[%]

Out[41]//MatrixForm=
        ( 1.30344   0.540457   0.650374  )
        ( 0.540457  1.30015    -0.397378 )
        ( 0.650374  -0.397378  1.64371   )
```

There are various ways to check the adequacy of a chosen model. The residuals $\{\hat{Z}_t\}$ of a fitted ARMA(p, q) process are defined by $\hat{Z}_t = \theta^{-1}(B)\,\phi(B)\,X_t$ where $t = 1, 2, \ldots, n$. One can infer the adequacy of a model by looking at the behavior of the residuals. The portmanteau test uses the statistic $Q_h = n\,(n+2)\sum_{i=1}^{h} \hat{\rho}^2\,(i)/(n-i)$ where $\hat{\rho}(i)$ is the sample correlation function of the residuals; Q_h is approximately chi-squared with $h - p - q$ degrees of freedom. Q_h for an m-variate time series is similarly defined, and is approximately chi-squared with $m^2(h - p - q)$ degrees of freedom.

`Residual[`*data*`,` *model*`]`	give the residuals of fitting *model* to *data*
`PortmanteauStatistic[`*residual*`,` *h*`]`	calculate the portmanteau statistic Q_h from *residual*

Residuals and test statistic.

The adequacy of the fitted model for the earlier example is accepted at level 0.05 for $h = 20$ since $\chi^2_{.95}(72) > Q$.

```
In[42]:= Q = PortmanteauStatistic[Residual[newseries, armodel], 20]

Out[42]= 64.8689
```

After establishing the adequacy of a model, we can proceed to forecast future values of the series. The best linear predictor is defined as the linear combination of observed data points that has the minimum mean-square distance from the true value. `BestLinearPredictor` gives the exact best linear predictions and their mean squared errors using the innovations algorithm. When the option `Exact` is set to `False` the approximate best linear predictor is calculated. For an ARIMA or SARIMA series $\{X_t\}$ with a constant term, the prediction for future values of $\{X_t\}$ can be obtained from the predicted values of $\{Y_t\}$, $\{\hat{Y}\}$, where $Y_t = (1-B)^d\,(1-B^s)^D\,X_t$, using `IntegratedPredictor`.

`BestLinearPredictor[` *data*`,` *model*`,` *n*`]`	give the prediction of *model* for the next *n* values and their mean square errors
`IntegratedPredictor[` *xlist*`,` {*d*`,` *D*}`,` *s*`,` *yhatlist*`]`	give the predicted values of $\{X_t\}$ from the predicted values *yhatlist*

Predicting time series.

option name	*default value*	
`Exact`	`True`	whether to calculate exactly

Option for `BestLinearPredictor`.

This gives the prediction for the next three data points and their mean square errors for the vector ARMA(1, 1) process.

```
In[43]:= (model = ARMAModel[{{{0.5, -0.2}, {0.3, -0.3}}},
            {{{-0.5, 0.2}, {0.8, 0.3}}}, {{1, 0}, {0, 1}}];
          tseries1 = TimeSeries[model, 20];
          BestLinearPredictor[tseries1, model, 3])
```

$$\text{Out[43]=}\ \left\{\{\{0.0878762,\ 1.16349\},\ \{-0.188761,\ -0.322686\},\ \{-0.0298433,\ 0.0401774\}\},\ \left\{\left\{\left\{1.,\ -2.8716\times10^{-9}\right\},\ \left\{-2.8716\times10^{-9},\ 1.\right\}\right\},\ \left\{\left\{1.,\ 1.26535\times10^{-9}\right\},\ \left\{1.26535\times10^{-9},\ 2.21\right\}\right\},\ \{\{1.0484,\ 0.0726\},\ \{0.0726,\ 2.3189\}\}\right\}\right\}$$

Here is the prediction for the next four data points and their mean square errors for the previously fitted SARIMA(1, 0, 1)(0, 0, 1)$_2$ model.

```
In[44]:= BestLinearPredictor[tseries, sarima, 4]
```

```
Out[44]= {{-0.341681, 0.196892, -0.0795141, 0.0588928}, {0.854466, 1.7799, 3.21265, 4.27749}}
```

Here is the prediction for the next four data points and their mean square errors for an ARIMA(2, 2, 2) model.

```
In[45]:= (model = ARIMAModel[2, {1, -0.5}, {0.3, 0.6}, 1];
          SeedRandom[3986];
          tseries = TimeSeries[model, 20];
          BestLinearPredictor[tseries, model, 4, Exact -> False])
```

```
Out[45]= {{-165.981, -182.705, -200.078, -217.858}, {1, 11.89, 60.89, 191.992}}
```

2.3 The Kalman Filter

The Kalman filter is a technique that can be used to recursively estimate unobservable quantities called state variables, $\{X_t\}$, from an observed time series $\{Y_t\}$. Many time series models (including the ARMA models) can be cast into a state-space form given by

$$\begin{aligned} Y_t &= G_t X_t + d_t + W_t \\ X_t &= F_t X_{t-1} + c_t + V_t, \end{aligned}$$

where $\{Y_t\}$ is the time series we observe and $\{X_t\}$ is the state variable. F, G, c, and d are known matrices or vectors, and they can be dependent on time. $\{W_t\}$ and $\{V_t\}$ are independent Gaussian white noise variables with zero mean and covariance matrices $E\, V_t\, V_s' = \delta_{t\,s}\, Q_t$ and $E\, W_t\, W_s' = \delta_{t\,s}\, R_t$, respectively.

Let $\hat{X}_{t|s}$ be the best linear estimate of X_t and $P_{t|s}$ be its mean square error, given I_s, the information up to time s. Given the initial values $\{\hat{X}_{m+1|m},\, P_{m+1|m}\}$, the Kalman filter yields $\{\hat{X}_{m+2|m+1},\, P_{m+2|m+1},\, \hat{X}_{m+1|m+1},\, P_{m+1|m+1}\}$.

`KalmanFilter`$[Y_t,\ \{\hat{X}_{t\,|\,t+1},\ P_{t-1}\},\ F_{t+1},\ G_t,\ Q_{t+1},\ R_t,\ c_{t+1},\ d_t]$ give $\{\hat{X}_{t+1|t}, P_{t+1|t}, \hat{X}_{t|t}, P_{t|t}\}$
`KalmanFilter`$[\{Y_{m+1},\ Y_{m+2},\ \ldots,\ Y_T\}, \{\hat{X}_{m+1|m},\ P_{m+1|m}\},\ F,\ G,\ Q,\ R,\ c,\ d]$ give $\{\{\hat{X}_{m+2|m+1}, P_{m+2|m+1}, \hat{X}_{m+1|m+1}, P_{m+1|m+1}\}, \ldots, \{\hat{X}_{T+1|T}, P_{T+1|T}, \hat{X}_{T|T}, P_{T|T}\}\}$ if F, G, R, c, and d are time independent

Kalman filtering.

Note that if any one of F, G, Q, R, c, *or* d is time dependent, F, G, Q, R, c, *and* d in the input to `KalmanFilter` shown above should be replaced by $\{F_{m+2}, F_{m+3}, \ldots, F_{T+1}\}$, $\{G_{m+1}, G_{m+2}, \ldots, G_T\}$, $\{Q_{m+2}, Q_{m+3}, \ldots, Q_{T+1}\}$, $\{R_{m+1}, R_{m+2}, \ldots, R_T\}$, $\{c_{m+2}, c_{m+3}, \ldots, c_{T+1}\}$, and $\{d_{m+1}, d_{m+2}, \ldots, d_T\}$, respectively. Note also that if $c \equiv 0$ and $d \equiv 0$, the last two arguments of `KalmanFilter` can be omitted.

A fixed-point Kalman smoother gives the estimated value of the state variable at t based on all the available information up to T, where $T > t$. The idea is that as new data are made available, we can improve our estimation result from the Kalman filter by taking into account the additional information.

`KalmanSmoothing`$[$*filterresult*, `F`$]$	"smooth" the result of the Kalman filtering

Kalman smoothing.

`KalmanSmoothing` gives $\{\{\hat{X}_{m+1|T}, \hat{X}_{m+2|T}, \ldots, \hat{X}_{T|T}\}, \{P_{m+1|T}, P_{m+2|T}, \ldots, P_{T|T}\}\}$. The first argument of `KalmanSmoothing`, *filterresult*, is the result of `KalmanFilter`, that is, $\{\{\hat{X}_{m+2|m+1}, P_{m+2|m+1}, \hat{X}_{m+1|m+1}, P_{m+1|m+1}\}, \ldots, \{\hat{X}_{T+1|T}, P_{T+1|T}, \hat{X}_{T|T}, P_{T|T}\}\}$, and F is the transition matrix in the above

state equation. If F is time dependent, the second argument of `KalmanSmoothing` should be $\{F_{m+2}, F_{m+3}, \ldots, F_{T+1}\}$.

The Kalman prediction estimates the state variable X_{t+h} based on the information at t, I_t.

`KalmanPredictor`$\left[\left\{\hat{X}_{t+1|t},\ P_{t+1|t}\right\},\ F,\ Q,\ c,\ h\right]$

give the next h predicted values and their mean square errors

Kalman predictor.

`KalmanPredictor`$\left[\left\{\hat{X}_{t+1|t},\ P_{t+1|t}\right\},\ F,\ Q,\ c,\ h\right]$ or `KalmanPredictor`$\left[\left\{\hat{X}_{t+1|t},\ P_{t+1|t}\right\},\right.$

$\{F_{t+2}, \ldots, F_{t+h}\}$, $\{Q_{t+2}, \ldots, Q_{t+h}\}$, $\{c_{t+2}, \ldots, c_{t+h}\}\left.\right]$ gives the next h predicted values and their mean square errors $\left\{\left\{\hat{X}_{t+1|t}, \hat{X}_{t+2|t}, \ldots, \hat{X}_{t+h|t}\right\}, \left\{P_{t+1|t}, P_{t+2|t}, \ldots, P_{t+h|t}\right\}\right\}$. Again, the argument c can be omitted if it is always 0.

A simple structural model is the local level model. It is given by

$$\begin{aligned} y_t &= \mu_t + \epsilon_t \\ \mu_t &= \mu_{t-1} + \eta_t. \end{aligned}$$

The model is in state-space form with $F = 1$, $G = 1$, and $c = d = 0$.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

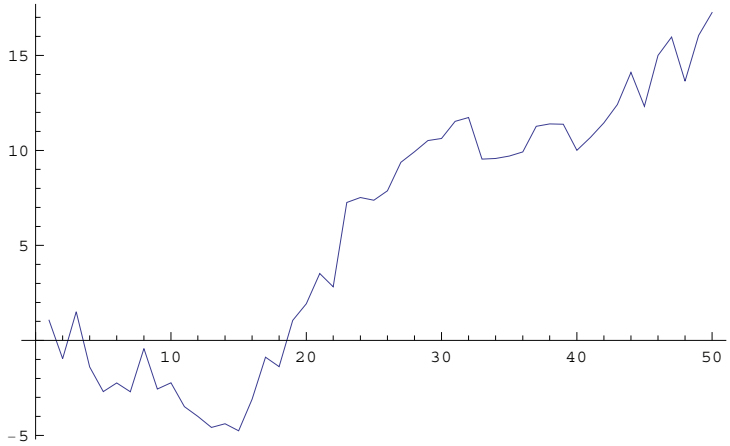
In the state-space model given earlier, if we set $F = 1$, $G = 1$, and $c = d = 0$, we have a local level model. Here we generate a time series of length 30 according to the local level model with $Q = 1.2$ and $R = 0.6$.

```
In[2]:= (SeedRandom[354 101];
         eta = RandomSequence[0, 1.2, 50];
         mu = Accumulate[eta];
         data1 = RandomSequence[0, 0.6, 50] + mu;)
```

This plots the series.

```
In[3]:= ListLinePlot[data1]
```

Out[3]=



To start the Kalman filter iteration, initial values are needed. Here we use the so-called diffuse prior.

```
In[4]:= init =
         Limit[Take[KalmanFilter[data1[[1]], {0, k}, 1, 1, 1.2, 0.6], 2], k -> Infinity]

Out[4]= {1.05868, 1.8}
```

We can also use the function `getInit` to get the initial values.

```
In[5]:= getInit[y_?AtomQ, F_, G_, Q_, R_, c_:0, d_:0] := {# * (y - d) + c, #^2 * R + Q} &[F / G];

In[6]:= getInit[y_?VectorQ, F_, G_, Q_, R_, c_:0, d_:0] :=
         Module[{m = Length[y], invF = Inverse[F], temp, zero, finvtemp, tempzero},
          zero = Table[0, {m}];
          temp = Flatten[NestList[#.invF &, G, m - 1], 1];
          finvtemp = F.Inverse[Reverse[temp]];
          tempzero = Append[Reverse[Rest[temp]], zero];
          {finvtemp.(y - d + If[c === 0, 0, Reverse[Accumulate[Rest[temp]]].c]) + c,
           finvtemp.(Outer[Total[MapThread[Dot, {#1, #2}]] &,
                NestList[Append[Rest[#], zero] &, #.Q & /@ tempzero, m - 1],
                NestList[Append[Rest[#], zero] &, tempzero, m - 1], 1] +
               R IdentityMatrix[m]).Transpose[finvtemp] + Q}]

In[7]:= init = getInit[data1[[1]], 1, 1, 1.2, 0.6]

Out[7]= {1.05868, 1.8}
```

This gives the result of Kalman filtering. Note that the initial values are $\{\hat{X}_{2|1}, P_{2|1}\}$ and the series now starts from $t = 2$.

```
In[8]:= kf = KalmanFilter[Rest[data1], init, 1, 1, 1.2, 0.6];
```

Here the form of the output of `KalmanFilter` is shown. It is a list of $\{\hat{X}_{t+1|t}, P_{t+1|t}, \hat{X}_{t|t}, P_{t|t}\}$ for $t = 2, 3, \ldots, T$, with $T = 50$.

```
In[9]:= Short[kf, 6]

Out[9]//Short=
      {{-0.453302, 1.65, -0.453302, 0.45},
       {0.983304, 1.64, 0.983304, 0.44}, <<46>>, {16.7961, 1.63923, 16.7961, 0.43923}}
```

This gives the smoothed estimates of the trend and their mean square errors.

```
In[10]:= KalmanSmoothing[kf, 1];
```

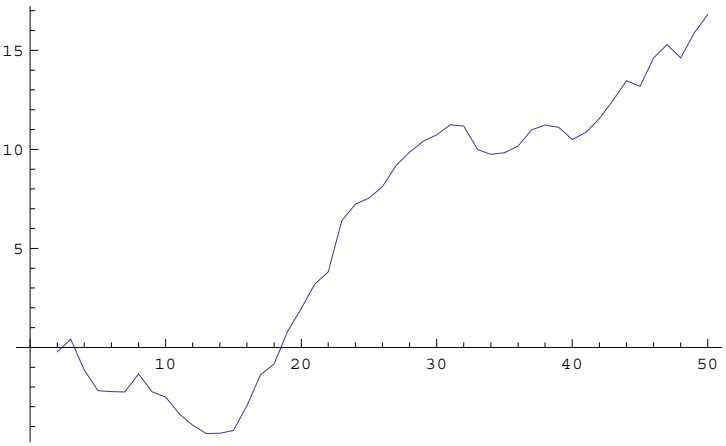
Note that the output of `KalmanSmoothing` is a list of two lists. The first list contains $\hat{X}_{t|T}$ for $t = 2, 3, \ldots, T$, with $T = 50$ and the second list contains the corresponding mean square errors.

```
In[11]:= Dimensions[%]

Out[11]= {2, 49}
```

The smoothed values are plotted here.

```
In[12]:= ListLinePlot[%%[[1]], DataRange -> {2, 50}]
```

Out[12]=



This gives the next three predicted values and their mean square errors.

```
In[13]:= KalmanPredictor[Take[Last[kf], 2], 1, 1.2, 0, 3]

Out[13]= {{16.7961, 16.7961, 16.7961}, {1.63923, 2.83923, 4.03923}}
```

The Kalman filter can be used to calculate the likelihood function of a time series in state-space form.

`LogLikelihood`$[data, init, F, G, Q, R, c, d]$	give the logarithm of the Gaussian likelihood of *data*

Log likelihood.

Now the variances R and Q are to be estimated and the initial values depends on them.

```
In[14]:= init = getInit[data1[[1]], 1, 1, q, r]

Out[14]= {1.05868, q + r}
```

This gives maximum likelihood estimate of Q and R.

```
In[15]:= Clear[f, if];
         if = Function[Evaluate[init /. {q → #1, r → #2}]];
         f[q_?NumericQ, r_?NumericQ] := -LogLikelihood[Rest[data1], if[q, r], 1, 1, q, r]

In[16]:= FindMinimum[f[q, r], {q, 0.8, 0.9}, {r, 0.3, 0.4}]

Out[16]= {44.003, {q → 1.65215, r → 0.302524}}
```

2.4 Univariate ARCH and GARCH Models

Autoregressive Conditional Heteroskedasticity (ARCH) models and Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models are used to model the changes in variance as a function of time. A GARCH(p, q) model is

$$Z_t = \nu_t \sqrt{h_t}$$

where $\{\nu_t\}$ is an independently distributed Gaussian random sequence with zero mean and unit variance; h_t is the conditional variance of Z_t conditional on all the information up to time $t-1$, I_{t-1}:

$$E\left(Z_t^2 \mid I_{t-1}\right) = h_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i z_{t-i}^2 + \sum_{i=1}^{p} \beta_i h_{t-i}.$$

When $p = 0$, we have an ARCH(q) model. A GARCH-in-mean (or ARCH-in-mean) model is defined by

$$Y_t = \delta f(h_t) + Z_t.$$

That is, the conditional mean is also a function of conditional variance h_t, where $f(h)$ is usually $\sqrt{h}$ or h. An ARCH (or GARCH) regression model is

$$Y_t = \mathbf{x}_t'.\mathbf{b} + Z_t$$

or

$$Y_t = \mathbf{x}_t'.\mathbf{b} + \delta f\,(h_t) + Z_t.$$

`ARCHModel[`*alphalist*`]`	ARCH(q) model with the coefficients $\{\alpha_0, \alpha_1,\ldots,\alpha_q\}$ in *alphalist*
`GARCHModel[`*alphalist*`, `*betalist*`]`	GARCH(p, q) model with the coefficients $\{\alpha_0, \alpha_1,\ldots,\alpha_q\}$ and $\{\beta_1,\ldots,\beta_q\}$ in *alphalist* and *betalist*, respectively
`ARCHModel[`*alphalist*`, `δ`, `f`]`	ARCH-in-mean model
`GARCHModel[`*alphalist*`, `*betalist*`, `δ`, `f`]`	GARCH-in-mean model

ARCH and GARCH models.

Note that the function f should be a symbol representing a *Mathematica* built-in function or a pure function. Note also these models will be referred as *archmodel* to differentiate them from ARMA type models.

TimeSeries[*archmodel*, *n*]	generate a time series of length *n* from *archmodel*
TimeSeries[*archmodel*, *n*, *init*]	generate a time series with given initial values *init*

Generating ARCH time series.

This loads the package.

```
In[1]:= Needs["TimeSeries`TimeSeries`"]
```

This generates a GARCH time series of length 10. Note that the third argument is optional. It specifies the initial values $\{\{h_{-p+1}, h_{-p+2}, \ldots, h_0\}, \{z_{-q+1}, z_{-q+2}, \ldots, z_0\}\}$, or $\{z_{-q+1}, z_{-q+2}, \ldots, z_0\}$ as in the case of an ARCH model.

```
In[2]:= (SeedRandom[4567];
         TimeSeries[GARCHModel[{0.05, 0.2}, {0.3}], 10, {{0.1}, {0.2}}])

Out[2]= {0.273984, -0.0604305, 0.195588, -0.034382,
         0.269706, 0.1712, -0.00232378, -0.278077, -0.567274, 0.315893}
```

Here we generate an ARCH-in-mean series of length 10. Note that the function *f* should be a symbol representing a *Mathematica* built-in function (here Log) or a pure function.

```
In[3]:= (SeedRandom[3457];
         TimeSeries[ARCHModel[{0.05, 0.3}, 0.5, Log], 10])

Out[3]= {-1.13392, -1.87536, -0.471234, -0.957423,
         -1.52742, -1.38422, -1.6642, -1.62875, -0.771731, -0.371803}
```

This generates the so-called AR-ARCH series. That is, the noise or error used in generating AR series is an ARCH process, with $\alpha_0 = 0.02$ and $\alpha_1 = 0.3$.

```
In[4]:= (SeedRandom[43 658];
         z = TimeSeries[ARCHModel[{0.02, 0.3}], 10];
         data = TimeSeries[ARModel[{0.5, -0.1}], z, {0, 0}])

Out[4]= {-0.116087, -0.294447, -0.0515721, 0.256337,
         0.0973735, -0.222807, -0.0720606, 0.167836, 0.159846, -0.167463}
```

LogLikelihood[*data*, *archmodel*]	give the logarithm of the Gaussian likelihood for the given *data* and *archmodel*
LogLikelihood[*data*, *archmodel*, *X*, *blist*]	give the logarithm of the Gaussian likelihood of an ARCH or GARCH regression series *data*
LogLikelihood[*data*, *archmodel*, ARModel[*philist*]]	give the logarithm of the Gaussian likelihood of an AR-ARCH or AR-GARCH series *data*
LogLikelihood[*data*, *archmodel*, ARModel[*philist*], μ]	give the logarithm of the Gaussian likelihood of an AR-ARCH or AR-GARCH series with a constant mean μ

Log likelihood for ARCH series.

option name	*default value*	
`PresampleValue`	`Automatic`	presample value

Option for `LogLikelihood`.

Note that the X is the matrix $X = \{\mathbf{x}_1', \mathbf{x}_2', \ldots, \mathbf{x}_N'\}$ and *blist* is a list of parameters **b** defined in the ARCH regression model. The presample values of $\{z_{-q+1}^2, \ldots, z_{-1}^2, z_0^2\}$ and $\{h_{-p+1}, \ldots, h_{-1}, h_0\}$ are assumed to be equal to a fixed value *sigma2*, and it can be specified using the option `PresampleValue -> sigma2`. The default setting for `PresampleValue` is `Automatic`, which corresponds to using the sample equivalence of σ^2 for GARCH models and to using $\alpha_0 / \left(1 - \sum_{i=1}^{q} \alpha_i - \sum_{i=1}^{p} \beta_i\right)$ for GARCH-in-mean models.

This gives the log likelihood of *data*.

```
In[5]:= LogLikelihood[data, ARCHModel[{0.02, 0.3}], ARModel[{0.5, -0.1}]]

Out[5]= 9.28712
```

`ConditionalMLEstimate[`*data*`,` *archmodel*`]`	fit *archmodel* to *data* using the conditional maximum likelihood method
`ConditionalMLEstimate[`*data*`,` *archmodel*`,` *X*`,` *blist*`]`	fit ARCH or GARCH regression model to *data*
`ConditionalMLEstimate[`*data*`,` *archmodel*`,` `ARModel[`*philist*`]]`	fit AR-ARCH or AR-GARCH model to *data*
`ConditionalMLEstimate[`*data*`,` *archmodel*`,` `ARModel[`*philist*`]`, μ`]`	fit nonzero mean AR-ARCH or AR-GARCH model to *data*

Conditional maximum likelihood estimations.

option name	*default value*	
`MaxIterations`	`30`	maximum number of iterations in searching for minimum
`PresampleValue`	`Automatic`	presample value

Options for `ConditionalMLEstimate`.

This generates a GARCH-in-mean series of length 200.

```
In[6]:= (SeedRandom[456 717];
         z = TimeSeries[GARCHModel[{0.06, 0.5}, {0.3}, 0.2, Sqrt], 200];)
```

`ConditionalMLEstimate` is used to estimate the model parameters. Note that the output contains the estimated model and the root mean square errors of the estimated parameters.

```
In[7]:= ConditionalMLEstimate[z, GARCHModel[{0.05, 0.4}, {0.3}, 0.1, Sqrt]]

Out[7]= {GARCHModel[{0.118552, 0.668799}, {-0.029961}, 0.270006, Sqrt],
          {{0.0240911, 0.177117}, {0.0554242}, 0.0554242}}
```

We generate an AR-ARCH series of length 50.

```
In[8]:= (SeedRandom[4561];
         z = TimeSeries[ARCHModel[{0.01, 0.5}], 50];
         data = TimeSeries[ARModel[{-0.7}], z];)
```

This gives the conditional maximum likelihood estimate of the model parameters and their corresponding root mean square errors.

```
In[9]:= ConditionalMLEstimate[data, ARCHModel[{0.01, 0.4}], ARModel[{-0.5}]]

Out[9]= {{ARCHModel[{0.00823786, 0.299001}], ARModel[{-0.780304}]},
          {{0.00210545, 0.255393}, {0.149147}}}
```

Note that we can also first estimate the parameter of the AR model.

```
In[10]:= ConditionalMLEstimate[data, 1]

Out[10]= ARModel[{-0.723085}, 0.0110195]
```

Then we estimate the ARCH parameters from the residuals. The two estimates give comparable results.

```
In[11]:= ConditionalMLEstimate[Residual[data, %], ARCHModel[{0.01, 0.4}]]

Out[11]= {ARCHModel[{0.00860008, 0.248135}], {0.00208182, 0.232919}}
```

The Lagrange multiplier (LM) test is commonly used to test for the existence of ARCH. The null hypothesis is no ARCH, that is, $\alpha_1 = \alpha_2 = \cdots = \alpha_q = 0$ and the LM statistic has a χ^2_q distribution asymptotically under the null hypothesis.

`LMStatistic[`*data*`,` *archmodel*`]`	calculate LM statistic with estimated parameters under the null inside *archmodel*
`LMStatistic[`*data*`,` *archmodel*`,` *X*`,` *blist*`]`	calculate LM statistic of an ARCH or GARCH regression series
`LMStatistic[`*data*`,` *archmodel*`, ARModel[`*philist*`]]`	calculate LM statistic of an AR-ARCH or AR-GARCH series
`LMStatistic[`*data*`,` *archmodel*`, ARModel[`*philist*`],` μ`]`	calculate LM statistic of an AR-ARCH or AR-GARCH series with nonzero mean μ

LM statistic.

option name	*default value*	
PresampleValue	Automatic	presample value

Option for LMStatistic.

To test the existence of ARCH using the LM test, we need to estimate the parameters under the null hypothesis. Note that the null hypothesis is no ARCH, that is, a pure AR(1) model.

```
In[12]:= ConditionalMLEstimate[data, ARCHModel[{0.01}], ARModel[{-0.5}], MaxIterations -> 100]

Out[12]= {{ARCHModel[{0.0109821}], ARModel[{-0.72387}]}, {{0.00184937}, {0.123171}}}
```

This is the alternative model with the parameters estimated under the null hypothesis. Note that $\alpha_1 = 0$ is inserted.

```
In[13]:= MapAt[Append[#, 0] &, %[[1]], {1, 1}]

Out[13]= {ARCHModel[{0.0109821, 0}], ARModel[{-0.72387}]}
```

This gives the LM statistic.

```
In[14]:= LMStatistic[data, Sequence @@ %]

Out[14]= 1.39372
```

Since this number is larger than the the above LM statistic, the null hypothesis is accepted.

```
In[15]:= Quantile[ChiSquareDistribution[1], 0.95]

Out[15]= 3.84146
```

References

1 Berndt, E. K., Hall, B. H., Hall, R. E., and Hausman, J. A. (1974), Estimation and inference in nonlinear structural models, *Annals of Economic and Social Measurement*, **3**, 653–665.

2 Bollerslev, T. (1986), Generalized autoregressive conditional heteroskedasticity, *Journal of Econometrics*, **31**, 307–327.

3 Box, G. E. P. and Jenkins, G. M. (1970), *Time Series Analysis: Forecasting and Control*, Holden-Day, San Francisco.

4 Brockwell, P. J. and Davis, R. A. (1987), *Time Series: Theory and Methods*, Springer-Verlag, New York.

5 Campbell, M. J. and Walker A. M. (1977), A survey of statistical work on the Mackenzie River series of annual Canadian lynx trappings for the years 1821–1934 and a new analysis, *Journal of the Royal Statistical Society, Series A*, **140**, 411–431.

6 Domowitz, I. and Hakkio, C. S. (1985), Conditional variance and the risk premium in the foreign exchange market, *Journal of International Economics*, **19**, 47–66.

7 Engle, R. F. (1982), Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation, *Econometrica*, **50**, No. 4, 987–1007.

8 Engle, R. F., Lilien, D. M., and Robins, R. P. (1987), Estimating time varying risk premia in the term structure: the ARCH-M Model, *Econometrica*, **55**, No. 2, 391–407.

9 Godolphin, E. J. and Unwin, J. M. (1983), Evaluation of the covariance matrix for the maximum likelihood estimator of a Gaussian autoregressive moving average process, *Biometrika* **70**, 279–284.

10 Granger, C. W. J. and Newbold, P. (1986), *Forecasting Economic Time Series*, Academic Press, New York.

11 Harvey, A. C. (1981), *Time Series Models*, Philip Allan, Oxford.

12 Harvey, A. C. (1989), *Forecasting, Structural Time Series Models and the Kalman Filter*, Cambridge University Press, Cambridge.

13 Jones, R. H. (1978), Multivariate autoregression estimation using residuals, *Applied Time Series Analysis* (ed. D. F. Findley), Academic Press, New York, 139–147.

14 Kendall, M. and Ord, J. K. (1990), *Time Series*, 3rd ed., Edward Arnold, London.

15 Morettin, P. A. (1984), The Levinson algorithm and its applications in time series analysis, *International Statistical Review*, **52**, 83–92.

16 Priestley, M. B. (1981), *Spectral Analysis and Time Series*, Vols. 1 and 2, Academic Press, New York.

17 Reinsel, G. C. (1993), *Elements of Multivariate Time Series Analysis*, Springer-Verlag, New York.

18 Wilson, G. T. (1973), The Estimation of Parameters in Multivariate Time Series Models, *Journal of the Royal Statistical Society, Series B*, **35**, 76–85.