GARCH Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 1

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098 Mail

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`comp.soft-sys.matlab` Newsgroup

`support@mathworks.com` Technical support
`suggest@mathworks.com` Product enhancement suggestions
`bugs@mathworks.com` Bug reports
`doc@mathworks.com` Documentation error reports
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

GARCH Toolbox User's Guide

Printing History: July 1999 First printingNew for Version 1.0, Release 11
November 2000 Reissued for Release 12 (Online only)

Contents

Preface

1 Tutorial

2 Function Reference

A Glossary

B Bibliography

Preface

The “Preface” includes:

- What Is the GARCH Toolbox?
- Related Products
- Using This Guide
- Technical Conventions
- Typographical Conventions

What Is the GARCH Toolbox?

MATLAB and the GARCH Toolbox provide an integrated computing environment for modeling the volatility of univariate economic time series. The GARCH Toolbox uses a general ARMAX/GARCH composite model to perform simulation, forecasting, and parameter estimation of univariate time series in the presence of conditional heteroskedasticity. Supporting functions perform tasks such as pre- and post-estimation diagnostic testing, hypothesis testing of residuals, model order selection, and time series transformations. Graphics capabilities let you plot correlation functions and visually compare matched innovations, volatility, and return series.

More specifically, you can:

- Perform Monte Carlo simulation of univariate returns, innovations, and conditional volatilities
- Specify conditional mean models of general ARMAX form and conditional models of general GARCH form for univariate asset returns
- Estimate parameters of general ARMAX/GARCH composite models via the maximum likelihood method
- Generate minimum mean square error forecasts of the conditional mean and conditional variance of univariate return series
- Perform pre- and post-estimation diagnostic and hypothesis testing, such as Engle's ARCH test, Ljung-Box Q-statistic test, likelihood ratio tests, and AIC/BIC model order selection
- Perform graphical correlation analysis, including auto-correlation, cross-correlation, and partial auto-correlation
- Convert price/return series to return/price series, and transform finite-order ARMA models to infinite-order AR and MA models

Related Products

The MathWorks provides several products that are related to the kinds of tasks you can perform with the GARCH Toolbox.

For more information about any of these products, see either:

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

Note The toolboxes listed below all include functions that extend MATLAB's capabilities. The blocksets all include blocks that extend Simulink's capabilities.

Product	Description
Database Toolbox	Tool for connecting to, and interacting with, most ODBC/JDBC databases from within MATLAB
Datafeed Toolbox	MATLAB functions for integrating the numerical, computational, and graphical capabilities of MATLAB with financial data providers
Excel Link	Tool that integrates MATLAB capabilities with Microsoft Excel for Windows
Financial Time Series Toolbox	Tool for analyzing time series data in the financial markets
Financial Toolbox	MATLAB functions for quantitative financial modeling and analytic prototyping
MATLAB Compiler	Compiler for automatically converting MATLAB M-files to C and C++ code

Product	Description
MATLAB C/C++ Math Library	Library for automatically converting MATLAB applications that contain math and graphics to C and C++ code for stand-alone applications
MATLAB Report Generator	Tool for documenting information in MATLAB in multiple output formats
MATLAB Runtime Server	MATLAB environment in which you can take an existing MATLAB application and turn it into a stand-alone product that is easy and cost-effective to package and distribute. Users access only the features that you provide via your application's graphical user interface (GUI) - they do not have access to your code or the MATLAB command line.
MATLAB Web Server	Tool for the development and distribution of Web-based MATLAB applications
Optimization Toolbox	Tool for general and large-scale optimization of nonlinear problems, as well as for linear programming, quadratic programming, nonlinear least squares, and solving nonlinear equations
Statistics Toolbox	Tool for analyzing historical data, modeling systems, developing statistical algorithms, and learning and teaching statistics

Prerequisites

The GARCH Toolbox requires the Statistics and Optimization Toolboxes. However, you need not read those manuals before reading this one.

Compatibility

The GARCH Toolbox is compatible with MATLAB R11 (Version 5.3) and later.

Using This Guide

Organization of the Document

"Tutorial" provides a brief overview of GARCH, then demonstrates the use of the GARCH Toolbox by estimating GARCH model parameters, and performing pre- and post-estimation analysis. Chapter 1 continues with discussions of simulation, forecasting, and regression, as well as model selection and analysis.

"Function Reference" describes the individual functions that comprise the GARCH Toolbox. The description of each function includes a synopsis of the function syntax, as well as a complete explanation of its arguments and operation. It may also include examples and references to additional reading material.

"Glossary" defines terms associated with modeling the volatility of economic time series.

"Bibliography" lists published materials that support concepts implemented in the GARCH Toolbox.

Expected Background

This guide is a practical introduction to the GARCH Toolbox. In general, it assumes you are familiar with the basic concepts of General Autoregressive Conditional Heteroskedasticity (GARCH) modeling.

In designing the GARCH Toolbox and this manual, we assume your title is similar to one of these:

- Analyst, quantitative analyst
- Risk manager
- Portfolio manager
- Fund manager, asset manager
- Economist
- Financial engineer
- Trader
- Student, professor, or other academic

We also assume your background, education, training, and responsibilities match some aspects of this profile:

- Finance, economics, perhaps accounting
- Engineering, mathematics, physics, other quantitative sciences
- Bachelor's degree minimum; MS or MBA likely; Ph.D. perhaps; CFA
- Comfortable with probability, statistics, and algebra
- May understand linear or matrix algebra, calculus, and differential equations
- Previously doing traditional programming (C, Fortran, etc.)
- May be responsible for instruments or analyses involving large sums of money
- Perhaps new to MATLAB

Technical Conventions

The *GARCH Toolbox User's Guide* uses the following definitions and descriptions:

- The *size* of an array describes the dimensions of the array. If a matrix has `m` rows and `n` columns, its size is `m`-by-`n`. If two arrays are the same size, their dimensions are the same.
- The *length* of a vector indicates only the number of elements in the vector. It does not indicate the size of the vector. If the length of a vector is `n`, it could be a 1-by-`n` (row) vector or an `n`-by-1 (column) vector. Two vectors of length `n`, one a row vector and the other a column vector, do not have the same size.
- The rows of a time series matrix correspond to a time index and the columns correspond to sample paths, independent realizations, or individual time series. In any given column, the first row contains the oldest observation and the last row contains the most recent observation.
- Time series vectors and matrices are time-tagged series of asset returns. If you have a price series, the GARCH Toolbox lets you convert it to a return series using either continuous compounding or periodic compounding.
- Continuous compounding is the default compounding method of the GARCH Toolbox. The results of some GARCH Toolbox functions are approximate for periodic compounding, but exact for continuous compounding. Using continuous compounding when moving between prices and returns, ensures exact results regardless of the function.
- The GARCH Toolbox assumes that return series are stationary processes. The price-to-return transformation generally guarantees a stable data set for GARCH modeling.
- The term *conditional* implies explicit dependence on a past sequence of observations. The term *unconditional* is more concerned with long-term behavior of a time series and assumes no explicit knowledge of the past.

See the Glossary for general term definitions.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention to Use	Example
Example code	Monospace font	To assign the value 5 to A, enter `A = 5`
Function names/syntax	Monospace font	The cos function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	Monospace font	MATLAB responds with
		`A =`
		`5`

Menu names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.
String variables (from a finite list)	*Monospace italics*	`sysc = d2c(sysd,'method')`

1

Tutorial

GARCH Overview

This section:

- Introduces GARCH
- Introduces the characteristics of GARCH models that are commonly associated with financial time series

Introducing GARCH

GARCH stands for Generalized Autoregressive Conditional Heteroskedasticity. Loosely speaking, you can think of heteroskedasticity as time-varying variance (i.e., volatility). Conditional implies a dependence on the observations of the immediate past, and autoregressive describes a feedback mechanism that incorporates past observations into the present. GARCH then is a mechanism that includes past variances in the explanation of future variances. More specifically, GARCH is a time series modeling technique that uses past variances and past variance forecasts to forecast future variances.

In this manual, whenever a time series is said to have GARCH effects, the series is heteroskedastic, i.e., its variances vary with time. If its variances remain constant with time, the series is homoskedastic.

Why Use GARCH?

GARCH modeling builds on advances in the understanding and modeling of volatility in the last decade. It takes into account excess kurtosis (i.e. fat tail behavior) and volatility clustering, two important characteristics of financial time series. It provides accurate forecasts of variances and covariances of asset returns through its ability to model time-varying conditional variances. As a consequence, you can apply GARCH models to such diverse fields as risk management, portfolio management and asset allocation, option pricing, foreign exchange, and the term structure of interest rates.

You can find highly significant GARCH effects in equity markets, not only for individual stocks, but for stock portfolios and indices, and equity futures markets as well [5]. These effects are important in such areas as value-at-risk (VaR) and other risk management applications that concern the efficient allocation of capital. You can use GARCH models to examine the relationship between long- and short-term interest rates. As the uncertainty for rates over various horizons changes through time, you can also apply GARCH models in

the analysis of time-varying risk premiums [5]. Foreign exchange markets, which couple highly persistent periods of volatility and tranquility with significant fat tail behavior [5], are particularly well suited for GARCH modeling.

Note Bollerslev [4] developed GARCH as a generalization of Engle's [8] original ARCH volatility modeling technique. Bollerslev designed GARCH to offer a more parsimonious model (i.e., using fewer parameters) that lessens the computational burden.

GARCH Limitations

Although GARCH models are useful across a wide range of applications, they do have limitations:

- GARCH models are only part of a solution. Although GARCH models are usually applied to return series, financial decisions are rarely based solely on expected returns and volatilities.
- GARCH models are parametric specifications that operate best under relatively stable market conditions [9]. Although GARCH is explicitly designed to model time-varying conditional variances, GARCH models often fail to capture highly irregular phenomena, including wild market fluctuations (e.g., crashes and subsequent rebounds), and other highly unanticipated events that can lead to significant structural change.
- GARCH models often fail to fully capture the fat tails observed in asset return series. Heteroskedasticity explains some of the fat tail behavior, but typically not all of it. Fat tail distributions, such as student-t, have been applied in GARCH modeling, but often the choice of distribution is a matter of trial and error.

Using GARCH to Model Financial Time Series

GARCH models account for certain characteristics that are commonly associated with financial time series:

- Fat tails
- Volatility clustering

Probability distributions for asset returns often exhibit fatter tails than the standard normal, or Gaussian, distribution. The fat tail phenomenon is known as excess kurtosis. Time series that exhibit a fat tail distribution are often referred to as leptokurtic. The blue (or dashed) line in Figure 1-1, A Plot Showing Excess Kurtosis illustrates excess kurtosis. The red (or solid) line illustrates a Gaussian distribution.

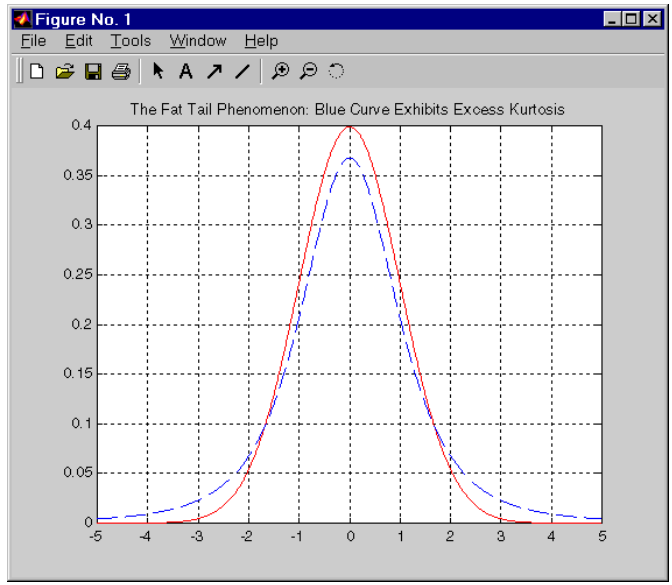


Figure 1-1: A Plot Showing Excess Kurtosis

In addition, financial time series usually exhibit a characteristic known as volatility clustering, in which large changes tend to follow large changes, and small changes tend to follow small changes (see Figure 1-2, A Plot Showing Volatility Clustering). In either case, the changes from one period to the next are typically of unpredictable sign. Volatility clustering, or persistence, suggests a time series model in which successive disturbances, although uncorrelated, are nonetheless serially dependent.

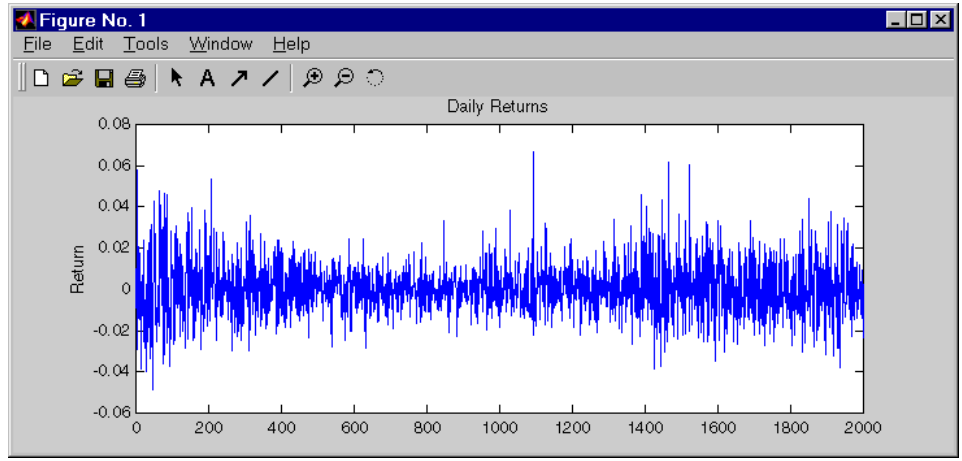


Figure 1-2: A Plot Showing Volatility Clustering

Volatility clustering (a type of heteroskedasticity) accounts for some but not all of the fat tail effect (or excess kurtosis) typically observed in financial data. A part of the fat tail effect can also result from the presence of non-Gaussian asset return distributions that just happen to have fat tails.

This section also discusses:

- Correlation in Financial Time Series
- Conditional Variances
- Serial Dependence in Innovations
- Homoskedasticity of the Unconditional Variance

Correlation in Financial Time Series

If you treat a financial time series as a sequence of random observations, this random sequence, or stochastic process, may exhibit some degree of correlation from one observation to the next. You can use this correlation structure to predict future values of the process based on the past history of observations. Exploiting the correlation structure, if any, allows you to decompose the time series into a deterministic component (i.e., the forecast), and a random component (i.e., the error, or uncertainty, associated with the forecast).

Eq. (1-1) uses these components to represent a univariate model of an observed time series y_t.

$$y_t = f(t-1, X) + \varepsilon_t \quad \textbf{(1-1)}$$

In this equation:

- $f(t-1, X)$ represents the deterministic component of the current return as a function of any information known at time $t-1$, including past innovations $\{\varepsilon_{t-1}, \varepsilon_{t-2}, \ldots\}$, past observations $\{y_{t-1}, y_{t-2}, \ldots\}$, and any other relevant explanatory time series data, X.
- ε_t is the random component. It represents the innovation in the mean of y_t. Note that you can also interpret the random disturbance, or shock, ε_t, as the single-period-ahead forecast error.

Conditional Variances

The key insight of GARCH lies in the distinction between conditional and unconditional variances of the innovations process $\{\varepsilon_t\}$. The term *conditional* implies explicit dependence on a past sequence of observations. The term *unconditional* is more concerned with long-term behavior of a time series and assumes no explicit knowledge of the past.

GARCH models characterize the conditional distribution of ε_t by imposing serial dependence on the conditional variance of the innovations. Specifically, the variance model imposed by GARCH, conditional on the past, is given by

$$Var_{t-1}(y_t) = E_{t-1}(\varepsilon_t^2) = \sigma_t^2 \quad \textbf{(1-2)}$$

where

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2 \quad \textbf{(1-3)}$$

Given the form of Eq. (1-2) and Eq. (1-3), you can see that σ_t^2 is the forecast of the next period's variance, given the past sequence of variance forecasts, σ_{t-i}^2, and past realizations of the variance itself, ε_{t-j}^2.

When $P = 0$, the GARCH(0,Q) model of Eq. (1-3) becomes Eq. (1-4), the original ARCH(Q) model introduced by Engle [8].

$$\sigma_t^2 = \kappa + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2 \qquad \textbf{(1-4)}$$

Eq. (1-3) and Eq. (1-4) are referred to as GARCH(P,Q) and ARCH(Q) variance models, respectively. Note that when $P = Q = 0$, the variance process is simply white noise with variance κ.

Parsimonious Parameterization. In practice, you often need a large lag Q for ARCH modeling, and this requires that you estimate a large number of parameters. To reduce the computational burden, Bollerslev [4] extended Engle's ARCH model by including past conditional variances. This results in a more parsimonious representation of the conditional variance process.

Volatility Clustering. Eq. (1-3) and Eq. (1-4) mimic the volatility clustering phenomenon. Large disturbances, positive or negative, become part of the information set used to construct the variance forecast of the next period's disturbance. In this manner, large shocks of either sign are allowed to persist, and can influence the volatility forecasts for several periods. The lag lengths P and Q, as well the magnitudes of the coefficients G_i and A_j, determine the degree of persistence. Note that the basic GARCH(P,Q) model of Eq. (1-3) is a symmetric variance process, in that the sign of the disturbance is ignored.

Serial Dependence in Innovations

A common assumption when modeling financial time series is that the forecast errors (i.e., the innovations) are zero-mean random disturbances uncorrelated from one period to the next.

$$\begin{aligned} E\{\varepsilon_t \varepsilon_T\} &= 0 \qquad t \neq T \\ &= \sigma_t^2 \qquad t = T \end{aligned}$$

As mentioned above, although successive innovations are uncorrelated, they are not independent. In fact, an explicit generating mechanism for a GARCH(P,Q) innovations process, $\{\varepsilon_t\}$, is

$$\varepsilon_t = \sigma_t z_t \qquad \textbf{(1-5)}$$

where σ_t is the conditional standard deviation given by the square root of Eq. (1-3), and z_t is a standardized, independent, identically distributed (i.i.d.)

random draw from some specified probability distribution. The GARCH literature uses several distributions to model GARCH processes, but the vast majority of research assumes the standard normal (i.e., Gaussian) density such that $\varepsilon_t \sim N(0, \sigma_t^2)$. Reflecting this, Eq. (1-5) illustrates that a GARCH innovations process $\{\varepsilon_t\}$ simply rescales an i.i.d process $\{z_t\}$ such that the conditional standard deviation incorporates the serial dependence of Eq. (1-3). Equivalently, Eq. (1-5) also states that a standardized GARCH disturbance, ε_t / σ_t, is itself an i.i.d. random variable z_t.

Notice that GARCH models are consistent with various forms of efficient market theory, which state that asset returns observed in the past cannot improve the forecasts of asset returns in the future. Since GARCH innovations $\{\varepsilon_t\}$ are serially uncorrelated, GARCH modeling does not violate efficient market theory.

Homoskedasticity of the Unconditional Variance

The GARCH Toolbox imposes the following parameter constraints on the conditional variance parameters.

$$\sum_{i=1}^{P} G_i + \sum_{j=1}^{Q} A_j < 1$$

$$\begin{aligned} &\kappa > 0 \\ &G_i \geq 0 \qquad i = 1, 2, \ldots, P \\ &A_j \geq 0 \qquad j = 1, 2, \ldots, Q \end{aligned} \tag{1-6}$$

The first constraint, a stationarity constraint, is necessary and sufficient for the existence of a finite, time-independent variance of the innovations process $\{\varepsilon_t\}$. The remaining constraints are sufficient to ensure that the conditional variance $\{\sigma_t^2\}$ is strictly positive.

When the conditional variance parameters satisfy the inequalities in Eq. (1-6), the unconditional variance (i.e., time-independent, or long-run variance expectation) of the innovations process $\{\varepsilon_t\}$ is

$$\sigma^2 = E(\varepsilon_t^2) = \frac{\kappa}{1 - \sum_{i=1}^{P} G_i - \sum_{j=1}^{Q} A_j} \quad \textbf{(1-7)}$$

Although Eq. (1-3) shows that the conditional variance of ε_t changes with time, Eq. (1-7) shows that the unconditional variance is constant (i.e., homoskedastic).

GARCH Toolbox Overview

This section discusses:

- Models for the Conditional Mean and Variance

  Allowable models for describing conditional mean and variance to the GRACH Toolbox
- Conventions and Clarifications

  MATLAB constructs and financial concepts as they are used in this manual
- The Default Model

  The default model that is used as the basis of discussion in this manual

Models for the Conditional Mean and Variance

The GARCH Toolbox allows a flexible model description of the conditional mean, using a general ARMAX form. ARMAX models encompass autoregressive (AR), moving average (MA), and regression (X) models, in any combination. Specifically, the toolbox allows a general ARMAX(R,M,Nx) form for the conditional mean

$$y_t = C + \sum_{i=1}^{R} AR_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{M} MA_j \varepsilon_{t-j} + \sum_{k=1}^{Nx} \beta_k X(t,k) \qquad \textbf{(1-8)}$$

where X is an explanatory regression matrix in which each column is a time series and $X(t,k)$ denotes the tth row and kth column.

The GARCH Toolbox models the conditional variance as a standard GARCH process with Gaussian innovations. It allows a general GARCH(P,Q) form with Gaussian innovations for the conditional variance

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2 \qquad \textbf{(1-9)}$$

Note This GARCH model is based on Bollerslev's original paper [4], and also includes Engle's original ARCH model [8] as a special case.

Conventions and Clarifications

Rows, Columns, Length, and Size

MATLAB operates as a large-scale, array-based processor, which makes it ideally suited for time series modeling and analysis. This manual imparts specific meanings to the words *length* and *size* in discussing arrays.

Matrices. A matrix is an m-by-n array in which m is the number of rows and n is the number of columns. By convention, the rows (i.e., the m-dimension) of a time series matrix correspond to a time index. In any given column, the first row contains the oldest observation and the last row contains the most recent observation. Columns (i.e., the n-dimension) correspond to sample paths, independent realizations, or individual time series.

Let *A* be a 100-by-5 time series matrix generated to support a Monte Carlo simulation experiment. In this case, *A* has 100 observations for each of five independent sample paths (or equivalently, five realizations of some underlying univariate random process in which each column is a realization of an individual time series). In this case, the *size* of *A* is 100-by-5. If some other matrix, *B*, is the same size as *A*, then *B* is also a 100-by-5 matrix.

Since the current release of the GARCH Toolbox addresses univariate models only, matrices usually represent multiple realizations of a univariate time series (as opposed to a single realization of a multivariate time series). Whenever a GARCH Toolbox function detects the presence of an input matrix of size m-by-n, it assumes that m is the number of time-tagged observations and n is the number of realizations.

Vectors. The *length* of a time series vector represents only the number of observations the vector contains. It does not indicate whether the vector is a row or column vector, i.e. it does not indicate the vector's size. For example, a time series vector of length 10 can be a row vector (i.e., a 1-by-10 matrix) or a column vector (i.e., a 10-by-1 matrix).

When a function detects a time series vector, row or column, it assumes that the vector represents a single realization of a univariate time series, and the length of the vector is the number of observations.

Note Although most functions can process either row or column vectors, you can avoid confusing input/output arguments if you format single realizations of a univariate time series as column vectors. Using column vectors also makes it easier for you to display data in the MATLAB command window.

Precision

The GARCH Toolbox performs all its calculations in double precision. Select **File** > **Preferences...** > **General** > **Numeric Format** to set the numeric format for your displays. The default is **Short**.

Prices, Returns, and Compounding

The GARCH Toolbox assumes that time series vectors and matrices are time-tagged series of observations. If you have a price series, the toolbox lets you convert it to a return series using either continuous compounding or periodic compounding in accord with Eq. (1-10) and Eq. (1-11).

If you denote successive price observations made at time t and t+1 as P_t and P_{t+1}, respectively, continuous compounding transforms a price series $\{P_t\}$ into a return series $\{y_t\}$ as

$$y_t = \log\frac{P_{t+1}}{P_t} = \log P_{t+1} - \log P_t \qquad \textbf{(1-10)}$$

Periodic compounding defines the transformation as

$$y_t = \frac{P_{t+1} - P_t}{P_t} = \frac{P_{t+1}}{P_t} - 1 \qquad \textbf{(1-11)}$$

Continuous compounding is the default compounding method of the GARCH Toolbox, and is the preferred method for most of continuous-time finance. Since GARCH modeling is typically based on relatively high frequency data (i.e., daily or weekly observations), the difference between the two methods is usually small. However, there are some toolbox functions whose results are

approximations for periodic compounding, but exact for continuous compounding. If you adopt the continuous compounding default convention when moving between prices and returns, all toolbox functions produce exact results.

Stationary and Nonstationary Time Series

Figure 1-3, Typical Equity Price Series illustrates a typical equity price series. Notice that there appears to be no long-run average level about which the series evolves. This is evidence of a nonstationary time series.

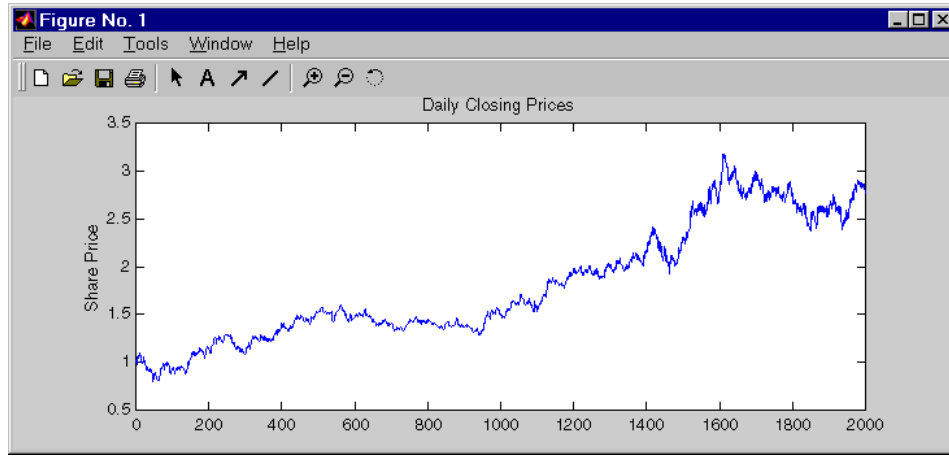


Figure 1-3: Typical Equity Price Series

Figure 1-4, Continuously Compounded Returns Associated with the Price Series, however, illustrates the continuously compounded returns associated with the same price series. In contrast, the returns appear to be quite stable over time, and the transformation from prices to returns has produced a stationary time series.

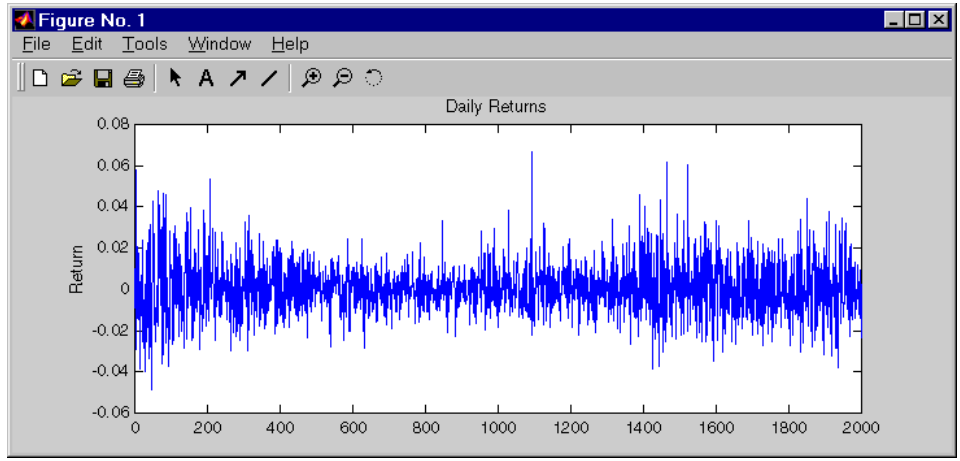


Figure 1-4: Continuously Compounded Returns Associated with the Price Series

The GARCH Toolbox assumes that return series are stationary processes. This may seem limiting, but the price-to-return transformation is common and generally guarantees a stable data set for GARCH modeling.

The Default Model

The GARCH Toolbox default model is the simple (yet common) conditional mean model with GARCH(1,1) Gaussian innovations, based on Eq. (1-8) and Eq. (1-9).

$$y_t = C + \varepsilon_t \quad \textbf{(1-12)}$$

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + A_1\varepsilon_{t-1}^2 \quad \textbf{(1-13)}$$

In the conditional mean model, Eq. (1-12), the returns, y_t, consist of a simple constant, plus an uncorrelated, white noise disturbance, ε_t. This model is often sufficient to describe the conditional mean in a financial return series. Most financial return series do not require the comprehensiveness that an ARMAX model provides.

In the conditional variance model, Eq. (1-13), the variance forecast, σ_t^2, consists of a constant plus a weighted average of last period's forecast, σ_{t-1}^2,

and last period's squared disturbance, ε_{t-1}^2. Although financial return series, as defined in Eq. (1-10) and Eq. (1-11), typically exhibit little correlation, the squared returns often indicate significant correlation and persistence. This implies correlation in the variance process, and is an indication that the data is a candidate for GARCH modeling.

Although simplistic, the default model shown in Eq. (1-12) and Eq. (1-13) has several benefits:

- It represents a parsimonious model that requires you to estimate only four parameters (C, κ, G_1, and A_1). According to Box and Jenkins [7], the fewer parameters to estimate, the less that can go wrong. Elaborate models often fail to offer real benefits when forecasting (see Hamilton [10], page 109).
- The simple GARCH(1,1) model captures most of the variability in most return series. Small lags for P and Q are common in empirical applications. Typically, GARCH(1,1), GARCH(2,1), or GARCH(1,2) models are adequate for modeling volatilities even over long sample periods (see Bollerslev, Chou, and Kroner [5], pages 10 and 22).

Analysis and Estimation Example Using the Default Model

The example in this section uses the GARCH Toolbox default model to examine the equity series of a hypothetical company, the XYZ Corporation. It uses the default model to estimate the parameters needed to model the series. Specifically, the example:

1. Performs a pre-estimation analysis to determine if the data is heteroskedastic and can be modeled using GARCH
2. Estimates the parameters for the default model
3. Performs a post-estimation analysis to confirm that the default model explains the heteroskedasticity present in the data

---

Note Due to platform differences, the estimation results you obtain when you recreate the examples in this chapter may differ from those shown in the text. These differences will propagate through any subsequent examples that use the estimation results as input, and may cause the numerical output of some examples to differ markedly from the text. These differences, however, do not affect the outcome of the examples.

Pre-Estimation Analysis

The pre-estimation analysis:

1. Loads the raw data: daily closing prices
2. Converts the prices to a return series
3. Checks for correlation
4. Quantifies the correlation

Load the Raw Data: Daily Closing Prices

Start by loading the MATLAB binary file `xyz.mat`, and examining its contents using the `whos` command.

```
load xyz
whos
  Name          Size          Bytes  Class

  prices     2001x1           16008  double array

Grand total is 2001 elements using 16008 bytes
```

The `whos` command lists all the variables in the current workspace, together with information about their size, bytes, and class.

The data you loaded from `xyz.mat` consists of a single column vector, `prices`, of length 2001. This vector contains the daily closing prices of the XYZ Corporation. Use the MATLAB `plot` function to examine the data (see `plot` in the online MATLAB Function Reference).

```
plot([0:2000], prices)
ylabel('Share Price')
title('Daily Closing Prices')
```

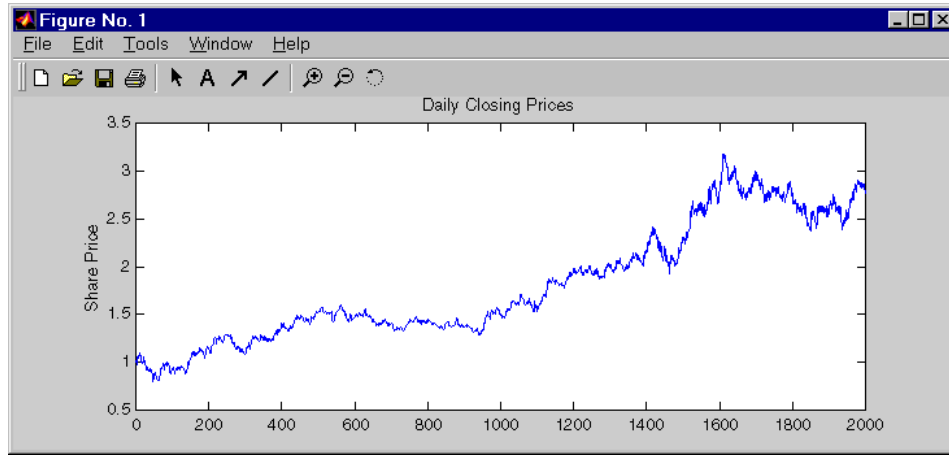


Figure 1-5: Daily Closing Prices of the XYZ Corporation

The plot shown in Figure 1-5, Daily Closing Prices of the XYZ Corporation is the same as the one shown in Figure 1-3, Typical Equity Price Series.

Convert the Prices to a Return Series

Because GARCH modeling assumes a return series, you need to convert the prices to returns. Use the utility function `price2ret`, and then examine the result.

```
xyz = price2ret(prices);
whos
  Name         Size         Bytes  Class

  prices    2001x1          16008  double array
  xyz       2000x1          16000  double array

Grand total is 4001 elements using 32008 bytes
```

The workspace information shows both the 2001-point price series and the 2000-point return series derived from it.

Now, use the MATLAB `plot` function to see the return series.

```
plot(xyz)
ylabel('Return')
title('Daily Returns')
```

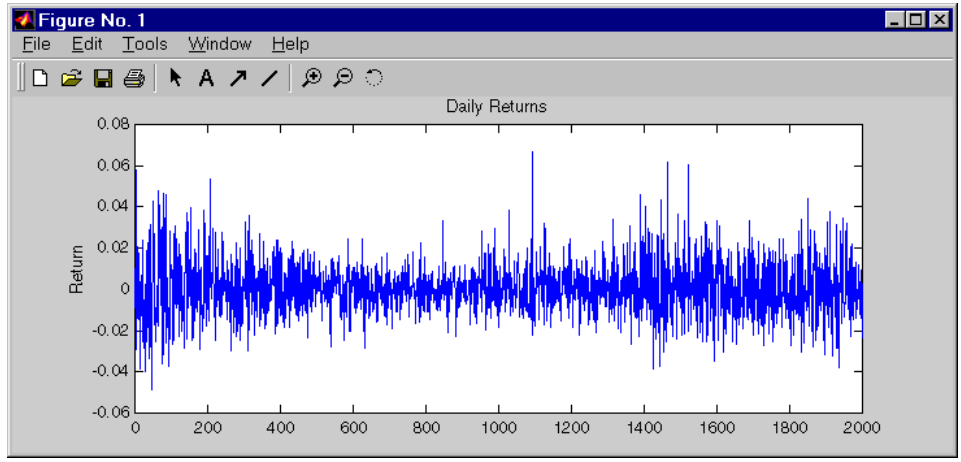


Figure 1-6: Raw Return Series Based on Daily Closing Prices

The results, shown in Figure 1-6, Raw Return Series Based on Daily Closing Prices, are the same as those shown in Figure 1-4, Continuously Compounded Returns Associated with the Price Series. Notice the presence of volatility clustering in the raw return series.

Check for Correlation

In the Return Series. You can check qualitatively for correlation in the raw return series by calling the functions `autocorr` and `parcorr` to examine the sample autocorrelation function (ACF) and partial-autocorrelation (PACF) function, respectively.

```
autocorr(xyz)
```

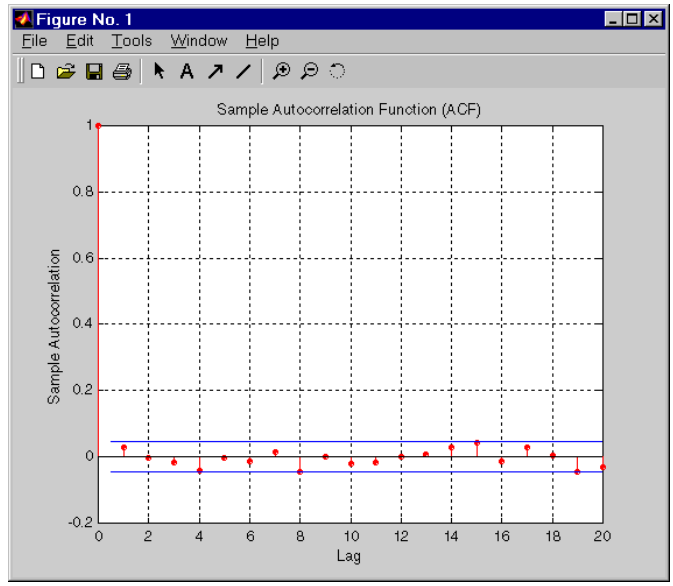


Figure 1-7: ACF with Bounds for the Raw Return Series

The `autocorr` function computes and displays the sample ACF of the returns, along with the upper and lower standard deviation confidence bounds, based on the assumption that all autocorrelations are zero beyond lag zero.

```
parcorr(xyz)
```

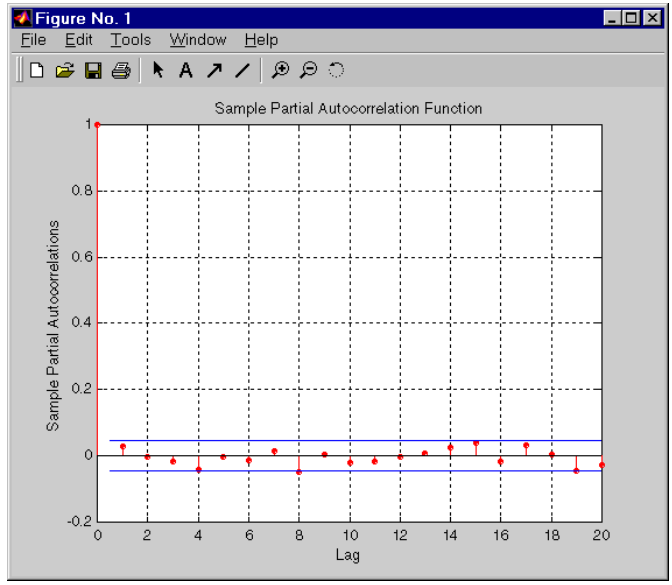


Figure 1-8: PACF with Bounds for the Raw Return Series

Similarly, the `parcorr` function displays the sample PACF with upper and lower confidence bounds.

Since the individual ACF values can have large variances and can also be autocorrelated, you should view the sample ACF and PACF with care (see Box, Jenkins, Reinsel [7], pages 34 and 186). However, as preliminary identification tools, the ACF and PACF provide some indication of the broad correlation characteristics of the returns. From Figure 1-7, ACF with Bounds for the Raw Return Series and Figure 1-8, PACF with Bounds for the Raw Return Series, there is no real indication that you need to use any correlation structure in the conditional mean. Also, notice the similarity between the graphs.

In the Squared Returns. Although the ACF of the observed returns exhibits little correlation, the ACF of the squared returns may still indicate significant correlation and persistence in the second-order moments. Check this by plotting the ACF of the squared returns.

```
autocorr(xyz.^2)
```

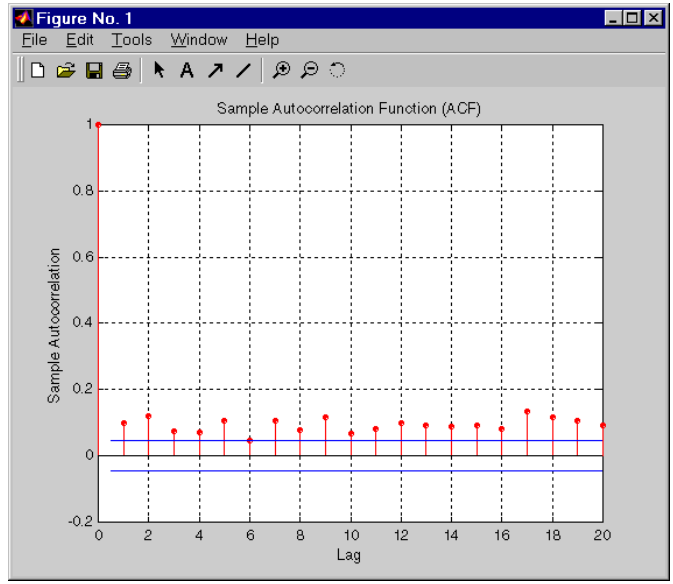


Figure 1-9: ACF of the Squared Returns

Figure 1-9, ACF of the Squared Returns shows that, although the returns themselves are largely uncorrelated, the variance process exhibits some correlation. This is consistent with the earlier discussion in the section, "The Default Model" on page 1-14. Note that the ACF shown in Figure 1-9, ACF of the Squared Returns appears to die out slowly, indicating the possibility of a variance process close to being nonstationary.

Note The syntax in the preceding command, an operator preceded by the dot operator (.), indicates that the operation is performed on an element-by-element basis. In the preceding command, `xyz.^2` indicates that each element of the vector `xyz` is squared.

Quantify the Correlation

You can quantify the preceding qualitative checks for correlation using formal hypothesis tests, such as the Ljung-Box-Pierce Q-test and Engle's ARCH test.

The function `lbqtest` implements the Ljung-Box-Pierce Q-test for a departure from randomness based on the ACF of the data. The Q-test is most often used as a post-estimation lack-of-fit test applied to the fitted innovations (i.e., residuals). In this case, however, you can also use it as part of the pre-fit analysis because the default model assumes that returns are just a simple constant plus a pure innovations process. Under the null hypothesis of no serial correlation, the Q-test statistic is asymptotically Chi-Square distributed (see Box, Jenkins, Reinsel [7], page 314).

The function `archtest` implements Engle's test for the presence of ARCH effects. Under the null hypothesis that a time series is a random sequence of Gaussian disturbances (i.e., no ARCH effects exist), this test statistic is also asymptotically Chi-Square distributed (see Engle [8], pages 999-1000).

Both functions return identical outputs. The first output, `H`, is a Boolean decision flag. `H = 0` implies that no significant correlation exists (i.e., do not reject the null hypothesis). `H = 1` means that significant correlation exists (i.e., reject the null hypothesis). The remaining outputs are the P-value (`pValue`), the test statistic (`Stat`), and the critical value of the Chi-Square distribution (`CriticalValue`).

Ljung-Box-Pierce Q-Test. Using `lbqtest`, you can verify, at least approximately, that no significant correlation is present in the raw returns when tested for up to 10, 15, and 20 lags of the ACF at the 0.05 level of significance.

```
[H, pValue, Stat, CriticalValue] = lbqtest(xyz-mean(xyz), [10 15 20]', 0.05);
[H  pValue  Stat  CriticalValue]
ans =
        0    0.2996   11.7869   18.3070
        0    0.2791   17.6943   24.9958
        0    0.1808   25.5616   31.4104
```

However, there is significant serial correlation in the squared returns when you test them with the same inputs.

```
[H, pValue, Stat, CriticalValue] = lbqtest((xyz-mean(xyz)).^2, [10 15 20]', 0.05);
[H  pValue  Stat  CriticalValue]
ans =
    1.0000         0  177.5937   18.3070
    1.0000         0  263.9325   24.9958
    1.0000         0  385.6907   31.4104
```

Engle's ARCH Test. You can also perform Engle's ARCH test using the function `archtest`. This test also shows significant evidence in support of GARCH effects (i.e heteroskedasticity).

```
[H, pValue, Stat, CriticalValue] = archtest(xyz-mean(xyz), [10 15 20]', 0.05);
[H  pValue  Stat  CriticalValue]
ans =
    1.0000         0  107.6171   18.3070
    1.0000         0  127.1083   24.9958
    1.0000         0  159.6543   31.4104
```

Each of these examples extracts the sample mean from the actual returns. This is consistent with the definition of the conditional mean equation of the default model, in which the innovations process is $\varepsilon_t = y_t - C$, and C is the mean of y_t.

Parameter Estimation

The parameter estimation:

1. Estimates the model parameters
2. Examines the estimated GARCH model

Estimate the Model Parameters

The presence of heteroskedasticity, shown in the previous analysis, indicates that GARCH modeling is appropriate. Use the estimation function `garchfit` to estimate the model parameters. Assume the default GARCH model described in the section "The Default Model" on page 1-14. This only requires that you specify the return series of interest as an argument to the function `garchfit`.

Note Because the default value of the `Display` parameter in the specification structure is **on**, `garchfit` prints diagnostic, optimization, and summary information to the MATLAB command window in the example below. (See `fmincon` in the Optimization Toolbox for information about the optimization information.)

```
[coeff, errors, LLF, innovations, sigma, summary] = garchfit(xyz);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Diagnostic Information
```

```
Number of variables: 4

Functions
 Objective:                              garchllfn
 Gradient:                               finite-differencing
 Hessian:                                finite-differencing (or Quasi-Newton)
 Nonlinear constraints:                  garchnlc
 Gradient of nonlinear constraints:      finite-differencing

Constraints
 Number of nonlinear inequality constraints: 0
 Number of nonlinear equality constraints:   0

 Number of linear inequality constraints:    1
 Number of linear equality constraints:      0
 Number of lower bound constraints:          4
 Number of upper bound constraints:          0

Algorithm selected
   medium-scale

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 End diagnostic information

                                  max                        Directional
 Iter  F-count   f(x)         constraint    Step-size      derivative   Procedure
    1      5    -5921.94  -1.684e-005            1      -7.92e+004
    2     34    -5921.94  -1.684e-005    1.19e-007            -553
    3     43    -5924.42  -1.474e-005        0.125           -31.2
    4     49    -5936.16  -6.996e-021            1            -288
    5     57    -5960.62            0         0.25            -649
    6     68    -5961.45  -4.723e-006       0.0313           -17.3
    7     75    -5963.18  -2.361e-006          0.5           -28.6
    8     81    -5968.24            0            1             -55
    9     90    -5970.54  -6.016e-007        0.125            -196
   10    103    -5970.84  -1.244e-006      0.00781           -16.1
   11    110    -5972.77  -9.096e-007          0.5           -34.4
   12    126    -5972.77  -9.354e-007     0.000977           -24.5
   13    134    -5973.29   -1.05e-006         0.25           -4.97
   14    141    -5973.95  -6.234e-007          0.5           -1.99
   15    147    -5974.21  -1.002e-006            1          -0.641
   16    153    -5974.57  -9.028e-007            1         -0.0803
   17    159    -5974.59  -8.054e-007            1         -0.0293
   18    165     -5974.6  -8.305e-007            1         -0.0039
   19    172     -5974.6  -8.355e-007          0.5        -0.000964
   20    192     -5974.6  -8.355e-007    -6.1e-005        -0.000646
   21    212     -5974.6  -8.355e-007    -6.1e-005        -0.000996    Hessian modified twice
```

```
   22   219     -5974.6  -8.361e-007          0.5       -0.000184
   23   239     -5974.6  -8.361e-007    -6.1e-005        -0.00441    Hessian modified twice
Optimization terminated successfully:
 Search direction less than 2*options.TolX and
  maximum constraint violation is less than options.TolCon
 No Active Constraints
```

Examine the Estimated GARCH Model

Now that the estimation is complete, you can display the parameter estimates and their standard errors using the function `garchdisp`,

```
garchdisp(coeff, errors)

  Number of Parameters Estimated: 4

                                Standard          T
  Parameter       Value           Error       Statistic
 -----------   -----------   ------------   -----------
          C    0.00049183     0.00025585       1.9223
          K    8.2736e-007    2.7446e-007      3.0145
   GARCH(1)    0.96283        0.0051557      186.7500
    ARCH(1)    0.03178        0.004416         7.1965
```

If you substitute these estimates in the definition of the default model, Eq. (1-12) and Eq. (1-13), the estimation process implies that the constant conditional mean/GARCH(1,1) conditional variance model that best fits the observed data is

$$y_t = 0.00049183 + \varepsilon_t$$

$$\sigma_t^2 = 8.2736\text{e}{-}007 + 0.96283\sigma_{t-1}^2 + 0.03178\varepsilon_{t-1}^2$$

where G_1 = `GARCH(1)` = `0.96283` and A_1 = `ARCH(1)` = `0.03178`. In addition, C = `C` = `0.00049183` and κ = `K` = `8.2736e-007`.

Figure 1-10, GARCH(1,1) Log-Likelihood Contours for the XYZ Corporation shows the log-likelihood contours of the default GARCH(1,1) model fit to the returns of the XYZ Corporation. The contour data is generated by the GARCH Toolbox demonstration function `garch11grid`. This function evaluates the log-likelihood function on a grid in the G_1-A_1 plane, holding the parameters C and κ fixed at their maximum likelihood estimates of `0.00049183` and `8.2736e-007`, respectively.

The contours confirm the printed `garchfit` results above. The maximum log-likelihood value, `LLF = 5974.6`, occurs at the coordinates G_1 = `GARCH(1) = 0.96283` and A_1 `= ARCH(1) = 0.03178`.

The figure also reveals a highly negative correlation between the estimates of the G_1 and A_1 parameters of the GARCH(1,1) model. This implies that a small change in the estimate of the G_1 parameter is nearly compensated for by a corresponding change of opposite sign in the A_1 parameter.

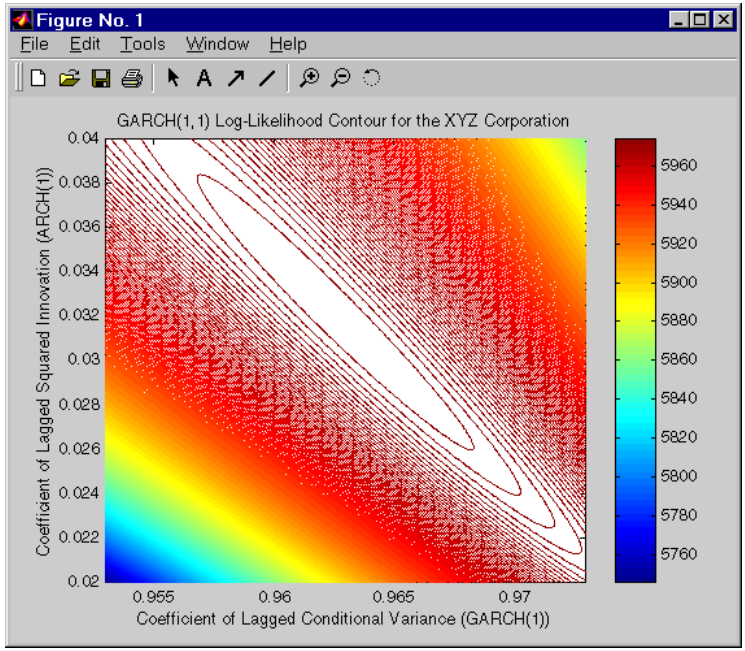


Figure 1-10: GARCH(1,1) Log-Likelihood Contours for the XYZ Corporation

---

Note If you view this manual on the Web, the color-coded bar at the right of the figure indicates the height of the log-likelihood surface above the GARCH(1,1) plane.

Post-Estimation Analysis

The post_estimation analysis:

1. Compares the residuals, conditional standard deviations, and returns
2. Plots and compares correlation of the standardized innovations
3. Quantifies and compares correlation of the standardized innovations

Compare the Residuals, Conditional Standard Deviations, and Returns

In addition to the parameter estimates and standard errors, `garchfit` also returns the optimized log-likelihood function value (`LLF`), the residuals (`innovations`), and conditional standard deviations (`sigma`). Use the function `garchplot` to inspect the relationship between the innovations (i.e., residuals) derived from the fitted model, the corresponding conditional standard deviations, and the observed returns. `garchplot` displays the tiered plot shown in Figure 1-11, Comparison of Innovations, Conditional Standard Deviations and Observed Returns.

```
garchplot(innovations, sigma, xyz)
```

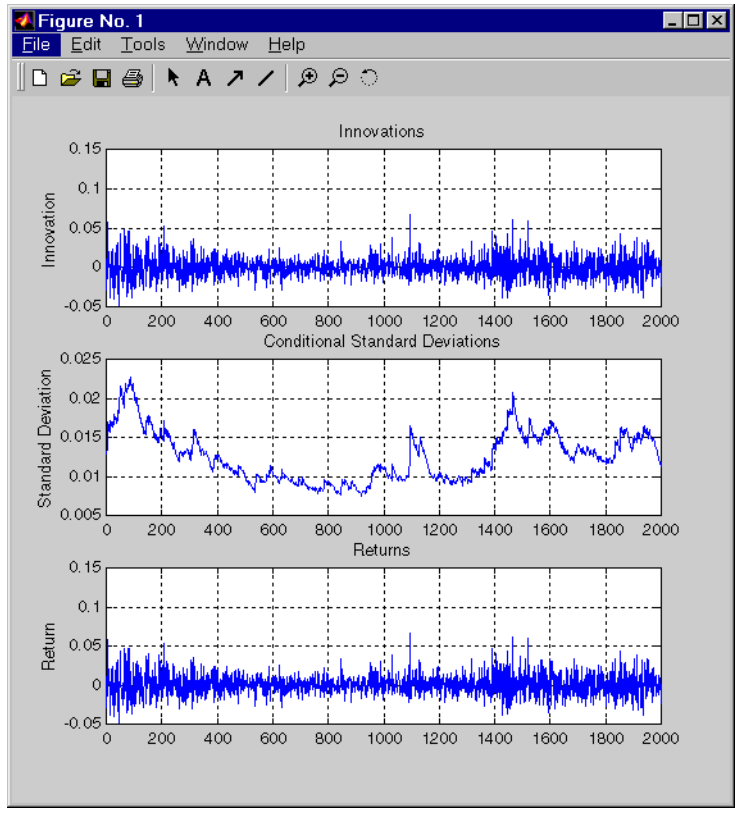


Figure 1-11: Comparison of Innovations, Conditional Standard Deviations and Observed Returns

Notice in Figure 1-11, Comparison of Innovations, Conditional Standard Deviations and Observed Returns that both the innovations (top plot) and the returns (bottom plot) exhibit volatility clustering. Also, notice that the sum, $G_1 + A_1 = 0.96283 + 0.03178$, is 0.99461, which is close to the integrated, nonstationary boundary given by Eq. (1-6).

Plot and Compare Correlation of the Standardized Innovations

Although the fitted innovations exhibit volatility clustering (Figure 1-11, Comparison of Innovations, Conditional Standard Deviations and Observed Returns), if you plot of the standardized innovations (the innovations divided

by their conditional standard deviation), they appear generally stable with little clustering.

```
plot(innovations./sigma)
ylabel('Innovation')
title('Standardized Innovations')
```

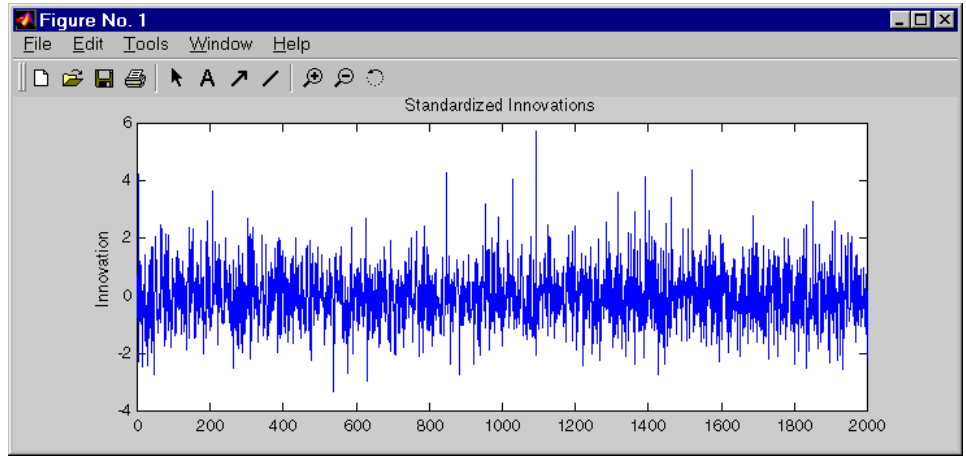


Figure 1-12: Standardized Innovations

If you plot the ACF of the squared standardized innovations (Figure 1-13, ACF of the Squared Standardized Innovations), they also show no correlation.

```
autocorr((innovations./sigma).^2)
```

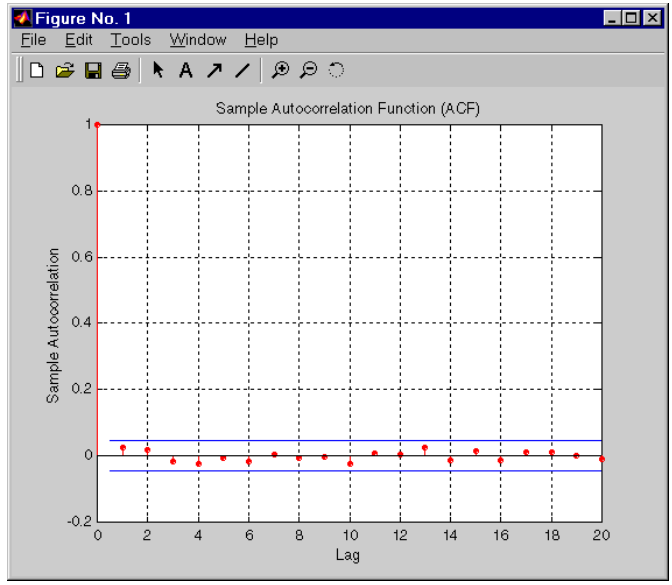


Figure 1-13: ACF of the Squared Standardized Innovations

Now compare the ACF of the squared standardized innovations (Figure 1-13, ACF of the Squared Standardized Innovations) to the ACF of the squared returns prior to fitting the default model (Figure 1-9, ACF of the Squared Returns). The comparison shows that the default model explains sufficiently the heteroskedasticity in the raw returns.

Quantify and Compare Correlation of the Standardized Innovations

Compare the results below of the Q-test and the ARCH test with the results of these same tests in the pre-estimation analysis. In the pre-estimation analysis, both the Q-test and the ARCH test indicate a rejection (`H = 1` with `pValue = 0`) of their respective null hypotheses, showing significant evidence in support of GARCH effects. In the post-estimate analysis, using standardized innovations based on the estimated model, these same tests indicate acceptance (`H = 0` with highly significant `pValues`) of their respective null hypotheses and confirm the explanatory power of the default model.

```
[H, pValue, Stat, CriticalValue] = lbqtest((innovations./sigma).^2,[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]
ans =
         0    0.8632    5.3966   18.3070
         0    0.9328    7.7677   24.9958
         0    0.9819    9.0843   31.4104


[H, pValue, Stat, CriticalValue] = archtest(innovations./sigma,[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]
ans =
         0    0.8883    5.0428   18.3070
         0    0.8765    9.0200   24.9958
         0    0.9521   10.7657   31.4104
```

The GARCH Specification Structure

This section discusses:

- Purpose of the Specification Structure
- Contents of the Specification Structure
- Valid Model Specifications
- Accessing Specification Structures
- Using the Specification Structure for Estimation, Simulation, and Forecasting

Purpose of the Specification Structure

Situations may arise in which you need more direct control of the analysis than is provided by the default model,

$$y_t = C + \varepsilon_t$$

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + A_1\varepsilon_{t-1}^2$$

(See “Analysis and Estimation Example Using the Default Model” on page 1-16.) For example, you may want to estimate the parameters of more elaborate conditional mean or variance models, perform Monte Carlo simulation, perform what-if analyses, or forecast time series.

The GARCH Toolbox maintains the parameters that define a model in a GARCH specification structure. In the default model example, `garchfit` creates the specification structure, `coeff`, and stores the model orders and estimated parameters of the default model in it. For more complex models, however, such as those required for the tasks listed above, you must specify the necessary parameters and store them in a specification structure.

The specification structure, `coeff` (from the default model example) represents the following default model estimated by `garchfit`.

$$y_t = 0.00049183 + \varepsilon_t$$

$$\sigma_t^2 = 8.2736\text{e-}007 + 0.96283\sigma_{t-1}^2 + 0.03178\varepsilon_{t-1}^2$$

"Contents of the Specification Structure" on page 1-33 shows the specification structure for the default model.

Contents of the Specification Structure

This example shows the contents of the specification structure. It is the specification structure, `coeff`, for the default model. The term to the left of the colon (`:`) is the parameter name.

```
coeff
coeff =
         Comment: 'Mean: ARMAX(0,0,0); Variance: GARCH(1,1)'
               R: 0
               M: 0
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 4.9183e-004
              AR: []
              MA: []
         Regress: []
               K: 8.2736e-007
          GARCH: 0.9628
           ARCH: 0.0318
           FixC: []
          FixAR: []
          FixMA: []
      FixRegress: []
            FixK: []
       FixGARCH: []
        FixARCH: []
    Optimization: [1x1 struct]
```

The specification structure parameters of interest in this discussion are `Comment`, `R`, `M`, `P`, `Q`, `C`, `AR`, `MA`, `Regress`, `K`, `GARCH`, and `ARCH`. (See the `garchset` reference page for a complete description of the GARCH specification structure parameters.) This section discusses:

- The Comment Field
- Equation Variables and Parameter Names
- Interpreting the Specification Structure

The Comment Field

The `Comment` field summarizes the ARMAX and GARCH models used for the conditional mean and variance equations in the default model example. The `Comment` value `'Mean: ARMAX(0,0,0); Variance: GARCH(1,1)'` describes the default model in terms of the general ARMAX(R,M,Nx) form for the conditional mean, where $R = M = Nx = 0$

$$y_t = C + \sum_{i=1}^{R} AR_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{M} MA_j \varepsilon_{t-j} + \sum_{k=1}^{Nx} \beta_k X(t,k) \qquad \textbf{(1-14)}$$

and the general GARCH(P,Q) form with Gaussian innovations for the conditional variance, where $P = Q = 1$.

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2 \qquad \textbf{(1-15)}$$

By default, `garchfit` and `garchset` generate the `Comment` field automatically Although you can set the value of the `Comment` field, it offers such a convenient summary that The MathWorks discourages your doing so. However, if you do specify your own comment, the GARCH Toolbox recognizes this and does not override your comment.

Equation Variables and Parameter Names

For the most part, the names of specification structure parameters that define the ARMAX/GARCH models reflect the variable names of their corresponding components in Eq. (1-14) and Eq. (1-15):

- `R` and `M` represent the order of the ARMA(R,M) model.
- `P` and `Q` represent the order of the GARCH(P,Q) model.
- `AR` represents the coefficient vector AR_i.
- `MA` represents the coefficient vector MA_j.
- `GARCH` represents the coefficient vector G_i.
- `ARCH` represents the coefficient vector A_j.
- `C` and `K` represent the constants C and κ, respectively.

Unlike the other components of these equations, X has no representation in the GARCH specification structure. X is an optional matrix of returns that some toolbox functions use as explanatory variables in the regression component of the conditional mean. For example, X could contain return series of a suitable market index collected over the same period as y. Toolbox functions that allow the use of a regression matrix provide a separate argument by which you can specify it. In the specification structure, `Regress` represents the coefficient vector of X, β_k.

Interpreting the Specification Structure

In the specification structure, `coeff`, for the default model example, the `AR`, `MA`, and `Regress` fields are empty matrices (`[]`). This is because the default mean equation is an ARMAX(0,0,0) model, where $R = M = Nx = 0$, and `AR`, `MA`, and `Regress` are R-, M-, and Nx-element vectors, respectively.

The `GARCH` and `ARCH` fields are both scalars set to their respective estimated values. They are scalars because the default variance equation is a GARCH(1,1) model, where $P = 1$ lag of the past conditional variance and $Q = 1$ lag of the past squared innovations.

`C` and `K` are the constants of the mean and variance equations, respectively. Their values were estimated by `garchfit`.

Valid Model Specifications

The specification structure you provide as input to all functions except `garchfit` must contain a complete model specification. That is, the orders of the ARMA and GARCH models must agree with the lengths of their corresponding coefficient vectors. Specifically, the value of `R` must be the same as the length of the vector `AR`, and `M` must be the same as the length of `MA`. The value of `P` must be the same as the length of the vector `GARCH`, and `Q` must be the same as the length of `ARCH`.

Only `garchfit` can accept as input a specification structure in which some or all of the model orders (`R`, `M`, `P`, or `Q`) are greater than 0 and the coefficient vectors are empty (`[]`). During the estimation process, `garchfit` creates appropriate coefficient vectors whose lengths correspond to the specified model orders.

Accessing Specification Structures

This section discusses:

- Using garchset to Create a Specification Structure
- Retrieving Specification Structure Values
- Accessing Fields Directly

Using garchset to Create a Specification Structure

The function `garchset` provides various options for creating and modifying a specification structure. Each of the following commands uses a different `garchset` syntax to create identical specification structures for the default model.

```
spec = garchset('R', 0, 'm', 0, 'P', 1, 'Q', 1);
spec = garchset('p', 1, 'Q', 1);
spec = garchset('R', 0, 'M', 0); spec = garchset(spec, 'P', 1, 'Q', 1);
spec = garchset;
```

The first command explicitly sets all model orders: `R`, `M`, `P`, and `Q`. This command illustrates the most basic `garchset` calling syntax. It specifies the structure fields as parameter/value pairs, in which the parameter name is a MATLAB character string enclosed in single quotes, followed by its corresponding value. When calling `garchset`, you only need to type the leading characters that uniquely identify the parameter. As illustrated here, case is ignored for parameter names.

The second command sets model orders for a GARCH(1,1) variance process only, and relies on the ARMAX(0,0,?) default for the mean. The third command creates an initial structure, and then updates the existing structure with additional parameter/value pairs. The last command, with no input arguments, creates a structure for the default model. The last command also implies that the following commands produce exactly the same estimation results.

```
[coeff, errors, LLF, innovations, sigma] = garchfit(xyz);
[coeff, errors, LLF, innovations, sigma] = garchfit(garchset, xyz);
```

Retrieving Specification Structure Values

The function `garchget` retrieves the values contained in specification structure fields.

Use garchget to retrieve the estimated coefficients from coeff. Then use garchset to write those coefficients to a new specification structure, spec, that is almost identical to coeff. For both garchget and garchset, you only need to type the leading characters that uniquely identify the parameter. Case is ignored for parameter names.

```
C = garchget(coeff, 'C')   % Use a separate garchget call to
                            % get each estimated coefficient.
C =
  4.9183e-004

K = garchget(coeff, 'K')

K =
  8.2736e-007

G = garchget(coeff, 'GARCH')

G =
    0.9628

A = garchget(coeff, 'ARCH')

A =
    0.0318
                            % Use garchset to create a new
                            % structure, spec.
spec = garchset('C', C, 'K', K, 'GARCH', G, 'ARCH', A)

spec =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(1,1)'
               R: 0
               M: 0
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 4.9183e-004
              AR: []
              MA: []
         Regress: []
```

```
                K: 8.2736e-007
            GARCH: 0.9628
             ARCH: 0.0318
             FixC: []
            FixAR: []
            FixMA: []
        FixRegress: []
              FixK: []
          FixGARCH: []
           FixARCH: []
     Optimization: [1x1 struct]
```

In this example, `garchset` automatically generates the first six fields (i.e., `Comment`, `R`, `M`, `P`, `Q`, and `Distribution`). Specifically, `garchset` infers the comment and model orders (`R`, `M`, `P`, `Q`) from the corresponding coefficient vectors (`AR`, `MA`, `ARCH`, `GARCH`). The converse is not true. If you specify only the model orders, `garchset` creates the coefficient vectors as empty matrices (`[]`). If you later call `garchfit`, it estimates the coefficient vectors for models of the order you specify, and updates the `AR`, `MA`, `ARCH`, and `GARCH` fields with these values.

Note The only difference between the `coeff` and `spec` structures above lies in their `Comment` fields. In `coeff`, `garchfit` explicitly sets the number of explanatory (i.e., regression) variables in the `Comment` field of `coeff` to `0`. This is because `coeff` represents an actual model whose conditional mean has no regression component. On the other hand, `garchset` inserts a '?' because it has no knowledge when it creates `spec`, whether you will include a regression component when you call `garchfit` to estimate the model parameters.

Accessing Fields Directly

In addition to using `garchset` and `garchget` to access the values of specification structure fields, you can also manipulate the fields of a structure directly. For example, the commands

```
garchget(spec, 'P')
spec.P
```

both retrieve the order `P` in the structure `spec`. Similarly, the commands

```
spec = garchset(spec, 'P', 3)
```

```
spec.P = 3
```

both write the order `P = 3`.

The first command in each case uses a GARCH Toolbox function to retrieve or write the value of a field. In this case the toolbox performs error checking (e.g., to ensure compatibility between inputs and guarantee that ARMA models are stationary/invertible). You also have the convenience of partial field names and case insensitivity.

In each case, the second command manipulates the structure directly. Although this approach does not support partial field names and case insensitivity, it can be convenient when you work interactively at the MATLAB command line. However, it does *not* provide error checking. For this reason, you should avoid manipulating a specification structure directly when writing code.

Note that the call to `garchset` above fails in your example workspace because the corresponding coefficient vector, `GARCH`, has only one element. Setting `spec.P = 3` directly succeeds but leaves you with an inconsistent specification structure.

Using the Specification Structure for Estimation, Simulation, and Forecasting

The three functions, `garchfit`, `garchpred`, and `garchsim`, comprise the core analysis and modeling routines of the GARCH Toolbox. These three functions operate on the GARCH specification structure. Table 1-1, GARCH Specification Structure Use describes each function's use of the GARCH specification structure.

Table 1-1: GARCH Specification Structure Use

Function	Description	Use of GARCH Specification Structure
`garchfit`	Estimates the parameters of a conditional mean specification of ARMAX form and a conditional variance specification of GARCH form.	**Input** Optionally accepts a GARCH specification structure as input. If the structure contains the model orders but no coefficient vectors (`C`, `AR`, `MA`, `Regress`, `K`, `ARCH`, `GARCH`), `garchfit` uses maximum likelihood to estimate the coefficients for an ARMAX/GARCH model of the specified orders. If the structure contains coefficient vectors, `garchfit` uses them as initial estimates for further refinement. If you provide no specification structure, `garchfit` assumes, and returns, a specification structure for the default model. **Output** Returns a specification structure that contains a fully specified ARMAX/GARCH model.
`garchpred`	Provides minimum-mean-square-error (MMSE) forecasts of the conditional mean and standard deviation of a return series, for a specified number of periods into the future.	**Input** Requires a GARCH specification structure that contains the coefficient vectors for the model for which `garchpred` is to forecast the conditional mean and standard deviation. **Output** `garchpred` does not modify or return the specification structure.
`garchsim`	Uses Monte Carlo methods to simulates sample paths for return series, innovations, and conditional standard deviation processes.	**Input** Requires a GARCH specification structure that contains the coefficient vectors for the model for which `garchsim` is to simulate sample paths. **Output** `garchsim` does not modify or return the specification structure.

Simulation

This section shows you how to:

- Simulate sample paths, using the simulation function `garchsim`, for return series, innovations, and conditional standard deviation processes
- Examine transient effects in the simulation process

It also provides a general simulation example.

Simulating Sample Paths

The section "Analysis and Estimation Example Using the Default Model" on page 1-16 models the equity series of a hypothetical company, the XYZ Corporation, using the default model. This section uses the resulting model

$$y_t = 0.00049183 + \varepsilon_t$$

$$\sigma_t^2 = 8.2736\text{e}{-007} + 0.96283\sigma_{t-1}^2 + 0.03178\varepsilon_{t-1}^2$$

to simulate sample paths, using the simulation function `garchsim`, for return series, innovations, and conditional standard deviation processes. You can think of `garchsim` as a filter that you can use to generate a (possibly) correlated return series $\{y_t\}$ from a white noise input series $\{\varepsilon_t\}$.

Use the following commands to restore your workspace if necessary. This example omits the estimation output to save space.

```
load xyz
xyz = price2ret(prices);
[coeff, errors, LLF, innovations, sigma] = garchfit(xyz);
coeff
coeff =
         Comment: 'Mean: ARMAX(0,0,0); Variance: GARCH(1,1)'
               R: 0
               M: 0
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 4.9183e-004
              AR: []
```

```
              MA: []
         Regress: []
               K: 8.2736e-007
           GARCH: 0.9628
            ARCH: 0.0318
            FixC: []
           FixAR: []
           FixMA: []
      FixRegress: []
            FixK: []
         FixGARCH: []
          FixARCH: []
     Optimization: [1x1 struct]
```

Using Default Inputs

Now call `garchsim` to simulate sample paths using the model in `coeff`. This command accepts `garchsim` defaults for:

- The number of sample paths (i.e. realizations) to generate: `1`
- The number of observations to generate for each path: `100`
- The random number generator seed: `0`

```
[e,s,y] =  garchsim(coeff);

whos e s y
  Name      Size         Bytes  Class

  e       100x1            800  double array
  s       100x1            800  double array
  y       100x1            800  double array

Grand total is 300 elements using 2400 bytes
```

The result is a single realization (i.e., one sample path) of 100 observations each for the innovations $\{\varepsilon_t\}$, conditional standard deviations $\{\sigma_t\}$, and returns $\{y_t\}$ processes. These processes are designated by the output variables `e`, `s`, and `y`, respectively.

Simulating a Much Longer Path

However, accurate GARCH modeling typically requires a few years worth of data. If there are 250 trading days per year, 1000 observations would be a more useful sample.

```
[e,s,y] =  garchsim(coeff, 1000);
whos e s y
  Name      Size         Bytes  Class

  e      1000x1           8000  double array
  s      1000x1           8000  double array
  y      1000x1           8000  double array

Grand total is 3000 elements using 24000 bytes
```

The result is a single realization of 1000 observations (roughly four years of data) for each of $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$. Plot the `garchsim` output data to see what it looks like.

```
garchplot(e,s,y)
```

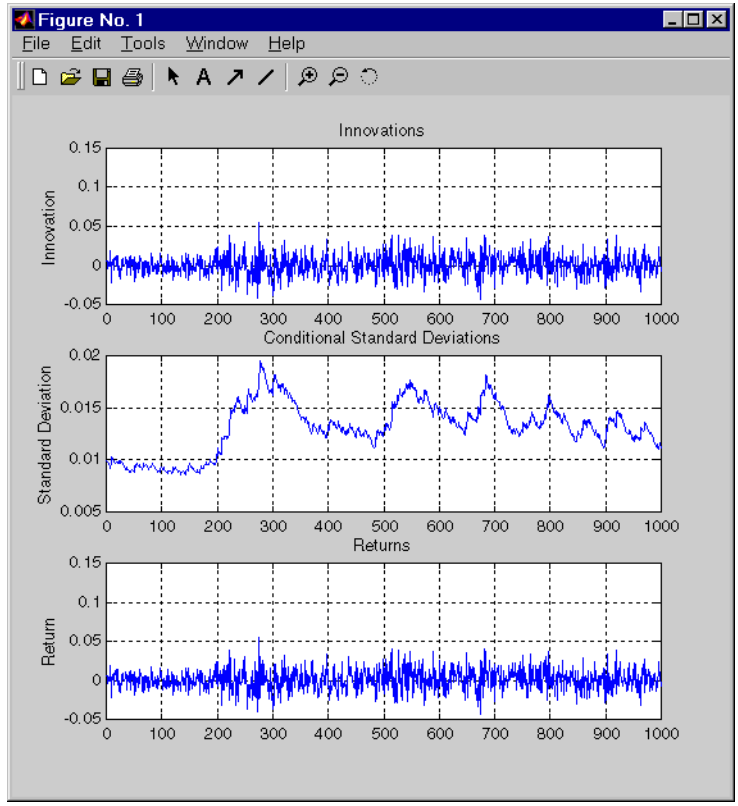


Figure 1-14: A Single Realization of 1000 Observations

Simulating Multiple Paths

However, Monte Carlo simulation requires multiple independent paths. Use the same model to simulate 1000 paths of 200 observations each.

```
[e,s,y] =  garchsim(coeff, 200, 1000);
whos e s y
  Name      Size         Bytes  Class

  e       200x1000     1600000  double array
  s       200x1000     1600000  double array
  y       200x1000     1600000  double array

Grand total is 600000 elements using 4800000 bytes
```

In this example, $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$ are 200-by-1000 element matrices. These are relatively large arrays, and demand large chunks of memory. In fact, because of the way the GARCH Toolbox manages transients, simulating this data requires more memory than the 4800000 bytes indicated above.

Transients in the Simulation Process

Automatic Minimization of Transient Effects

The function `garchsim` generates stable output processes in (approximately) steady-state by attempting to eliminate transients in the data it simulates. `garchsim` first estimates the number of observations needed for the transients to decay to some arbitrarily small value, and then generates a number of observations equal to the sum of this estimated value and the number you request. `garchsim` then ignores the estimated number of initial samples needed for the transients to decay sufficiently and returns only the requested number of later observations.

To do this, `garchsim` interprets the simulated GARCH(P,Q) conditional variance process as an ARMA(max(P,Q),P) model for the squared innovations (see, for example, Bollerslev [4], p.310). It then interprets this ARMA(max(P,Q),P) model as the correlated output of a linear filter and estimates its impulse response by finding the magnitude of the largest eigenvalue of its auto-regressive polynomial. Based on this eigenvalue, `garchsim` estimates the number of observations needed for the magnitude of the impulse response (which begins at 1) to decay below `0.01` (i.e., `1` percent). If the conditional mean has an ARMA(R,M) component, then `garchsim` also estimates its impulse response.

Depending on the values of the parameters in the simulated conditional mean and variance models, you may need long pre-sample periods for the transients

to die out. Although the simulation outputs may be relatively small matrices, the initial computation of these transients can result in a large memory burden and seriously impact performance. In the previous example, which simulates three 200-by-1000 element arrays, intermediate storage is required for far more than 200 observations.

Further Minimization of Transient Effects

If you suspect transients persist in the simulated data `garchsim` returns, you can oversample and delete observations from the beginning of each series. For example, suppose you would like to simulate 10 independent paths of 1000 observations each for $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$ starting from a known scalar random number seed (`12345`). Start by generating 1200 observations, then further minimize the effect of transients by retaining only the last 1000 observations of interest.

```
[e,s,y] =  garchsim(coeff, 1200, 10, 12345);
whos e s y
  Name      Size         Bytes  Class

  e      1200x10         96000  double array
  s      1200x10         96000  double array
  y      1200x10         96000  double array

Grand total is 36000 elements using 288000 bytes

e = e(end-999:end, :);
s = s(end-999:end, :);
y = y(end-999:end, :);

whos e s y
  Name      Size         Bytes  Class

  e      1000x10         80000  double array
  s      1000x10         80000  double array
  y      1000x10         80000  double array

Grand total is 30000 elements using 240000 bytes
```

Note The example above also illustrates how to specify a random number generator seed. If you do not specify a seed, as in the example in "Simulating Multiple Paths" on page 1-44, the default seed is `0` (the MATLAB initial state).

Understanding Transient Effects

The example in this section builds on the example in the section "Further Minimization of Transient Effects" on page 1-46. The previous example simulated 10 independent paths of 1000 observations each for $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$ and returned its outputs in the variables `e`, `s`, and `y` respectively. This example uses the GARCH Toolbox inference function `garchinfer` to infer $\{\varepsilon_t\}$ and $\{\sigma_t\}$ from the simulated return series `y`. It then compares the steady-state simulated innovations and conditional standard deviation processes with the inferred innovations and conditional standard deviation processes.

Essentially, `garchsim` uses an ARMA model as a linear filter to transform an uncorrelated input innovations process $\{\varepsilon_t\}$ into a correlated output returns process $\{y_t\}$. Use the function `garchinfer` to reverse this process by inferring innovations $\{\varepsilon_t\}$ and standard deviation $\{\sigma_t\}$ processes from the observations in $\{y_t\}$

```
[eInferred, sInferred] = garchinfer(coeff, y);
```

where `eInferred` and `sInferred` are the inferred innovations and conditional standard deviations, respectively. Notice that when you query the workspace, `eInferred` and `sInferred` are the same size as the simulated returns matrix `y`

```
whos eInferred  sInferred y
  Name            Size         Bytes  Class

  eInferred    1000x10         80000  double array
  sInferred    1000x10         80000  double array
  y            1000x10         80000  double array

Grand total is 30000 elements using 240000 bytes
```

Now compare the steady-state, simulated processes with their inferred counterparts by examining the third trial (i.e., the third column of each matrix). Note that there is nothing special about the third column, and the following comparisons hold for all columns.

First, create two matrices, `eData` and `sData`, to store the row numbers, the simulated and inferred series, and the difference between the two.

```
eData = [[1:length(e)]' e(:,3)  eInferred(:,3)  [e(:,3)-eInferred(:,3)]];
sData = [[1:length(s)]' s(:,3)  sInferred(:,3)  [s(:,3)-sInferred(:,3)]];

whos eData sData
  Name        Size         Bytes  Class

  eData    1000x4          32000  double array
  sData    1000x4          32000  double array

Grand total is 8000 elements using 64000 bytes
```

Now, print the first 10 rows of `eData` and `sData`, using the `fprintf` command to format the printed output, and examine the observations.

Note Depending on your platform, the innovations (`e`, `eInferred`) and standard deviations (`s`, `sInferred`) may differ in value from those shown below. This has little effect on the calculated differences, which continue to demonstrate the convergence shown in the last column.

```
fprintf('%4d %12.8f %12.8f %12.8f\n', eData(1:10,:)')
   1  -0.00111887  -0.00111887   0.00000000
   2  -0.01022535  -0.01022535   0.00000000
   3  -0.01391679  -0.01391679   0.00000000
   4   0.00769383   0.00769383   0.00000000
   5   0.00284161   0.00284161   0.00000000
   6   0.00837156   0.00837156   0.00000000
   7  -0.01022153  -0.01022153   0.00000000
   8  -0.00064348  -0.00064348   0.00000000
   9   0.00769471   0.00769471   0.00000000
  10  -0.00011629  -0.00011629  -0.00000000

fprintf('%4d %12.8f %12.8f %12.8f\n', sData(1:10,:)')
   1   0.01176309   0.01532522  -0.00356213
   2   0.01157993   0.01506653  -0.00348661
   3   0.01154388   0.01492360  -0.00337972
   4   0.01163145   0.01488014  -0.00324869
   5   0.01153130   0.01469346  -0.00316216
```

```
     6   0.01136278   0.01445537  -0.00309258
     7   0.01128578   0.01429146  -0.00300569
     8   0.01125978   0.01417048  -0.00291070
     9   0.01108652   0.01393485  -0.00284832
    10   0.01100236   0.01377214  -0.00276979
```

Notice that the difference between the simulated and inferred innovations is effectively zero immediately, whereas the standard deviations appear to converge slowly. If you examine every 25th observation of the standard deviations, through the 400th observation, the convergence is more obvious.

```
fprintf('%4d %12.8f %12.8f %12.8f\n', sData(25:25:400,:)')
    25   0.01060556   0.01230273  -0.00169717
    50   0.01167755   0.01230644  -0.00062889
    75   0.01290505   0.01312981  -0.00022476
   100   0.01228385   0.01237591  -0.00009206
   125   0.01256986   0.01260484  -0.00003498
   150   0.01292421   0.01293742  -0.00001321
   175   0.01212655   0.01213201  -0.00000546
   200   0.01155697   0.01155919  -0.00000222
   225   0.01409612   0.01409683  -0.00000071
   250   0.01468410   0.01468437  -0.00000026
   275   0.01336617   0.01336628  -0.00000011
   300   0.01138117   0.01138123  -0.00000005
   325   0.01414220   0.01414222  -0.00000002
   350   0.01312882   0.01312883  -0.00000001
   375   0.01494447   0.01494447  -0.00000000
   400   0.01704352   0.01704352  -0.00000000
```

The innovations processes of the default model converge immediately because the default model assumes a simple constant in the conditional mean equation (i.e., there is no correlation in the conditional mean). However, the GARCH(1,1) default conditional variance equation is highly persistent (recall that the GARCH and ARCH parameter estimates are 0.9628 and 0.0318, respectively).

Note The `fprintf` function lets you control the specific formatting of printed data. This example uses it to print the first 10 rows of `eData` and `sData`. It prints each innovation and difference value in fixed point notation, in a field of at least 12 digits, with 8 digits of precision. (See `fprintf` in the online MATLAB Function Reference.)

A General Simulation Example

This simulation example is more general than the previous one that used the default model, GARCH(1,1). It uses an ARMA(2,1) model to express correlation in the conditional mean. The example:

1. Defines an ARMA(2,1)/GARCH(1,1) model
2. Uses the model to simulate 2000 observations for return series, innovations, and conditional standard deviation processes
3. Infers the innovations and standard deviations for the simulated return series
4. Compares the simulated and inferred innovations for the first 20 observations

Create the Model

Start by creating an ARMA(2,1)/GARCH(1,1) composite model with repeated calls to `garchset`.

```
spec = garchset;
spec = garchset(spec, 'R', 2, 'M', 1);
spec = garchset(spec, 'C', 0, 'AR', [0.6 0.2], 'MA', 0.4);
spec = garchset(spec, 'K', 0.00001, 'GARCH', 0.8, 'ARCH', 0.1)
spec =
         Comment: 'Mean: ARMAX(2,1,?); Variance: GARCH(1,1)'
               R: 2
               M: 1
               P: 1
               Q: 1
    Distribution: 'Gaussian'
```

```
                C: 0
               AR: [0.6000 0.2000]
               MA: 0.4000
          Regress: []
                K: 1.0000e-005
            GARCH: 0.8000
             ARCH: 0.1000
             FixC: []
            FixAR: []
            FixMA: []
        FixRegress: []
              FixK: []
          FixGARCH: []
           FixARCH: []
    Optimization: [1x1 struct]
```

If you substitute the coefficient vectors from this specification structure in Eq. (1-8) and Eq. (1-9) you get the following ARMA(2,1) and GARCH(1,1) models. These are the models this example simulates.

$$y_t = 0 + 0.6y_{t-1} + 0.2y_{t-2} + \varepsilon_t + 0.4\varepsilon_{t-1}$$

$$\sigma_t^2 = 0.00001 + 0.8\sigma_{t-1}^2 + 0.1\varepsilon_{t-1}^2$$

Simulate and Infer the Innovations

Use this model to simulate 2000 observations in a return series, and their corresponding innovations and standard deviations. Then use `garchinfer` to infer innovations and standard deviations from the simulated return series.

```
[e,s,y] =  garchsim(spec, 2000);
[eInferred, sInferred] = garchinfer(spec, y);
```

You can think of the simulation engine `garchsim` as a filter that generates a (possibly) correlated return series $\{y_t\}$ from a white noise input series $\{\varepsilon_t\}$. `garchinfer` reverses this process by inferring innovations $\{\varepsilon_t\}$ and standard deviation $\{\sigma_t\}$ processes from the observations in $\{y_t\}$.

However, `garchinfer` is a convenience function that only provides a user-friendly interface to the log-likelihood objective function, `garchllfn`. So, in fact, `garchllfn` is the inverse (i.e., whitening) filter associated with the

simulation engine, because it infers the white noise process from the observed return series. (See the section “Understanding Transient Effects” on page 1-47.)

Note `garchfit` also calls the log-likelihood objective function, `garchllfn`, to infer the innovations and standard deviations.

Compare Simulated and Inferred Innovations

Now compare the simulated and inferred innovations for the first 20 observations. Notice that, after a few observations, the difference between the simulated and inferred innovations is insignificant.

```
eData = [[1:length(e)]'  e  eInferred  [e-eInferred]];
fprintf('%4d %12.8f %12.8f %12.8f\n', eData(1:20,:)')
   1  -0.00836573   0.00000000  -0.00836573
   2  -0.01976087   0.00000000  -0.01976087
   3  -0.00063568  -0.00854003   0.00790435
   4  -0.01022288  -0.00706114  -0.00316174
   5   0.00621509   0.00495039   0.00126470
   6   0.00496725   0.00547313  -0.00050588
   7   0.01596937   0.01576702   0.00020235
   8   0.00610852   0.00618946  -0.00008094
   9  -0.00640740  -0.00643977   0.00003238
  10   0.00367566   0.00368861  -0.00001295
  11  -0.00936189  -0.00936707   0.00000518
  12  -0.00018263  -0.00018056  -0.00000207
  13  -0.00043157  -0.00043240   0.00000083
  14   0.00000037   0.00000070  -0.00000033
  15  -0.00264566  -0.00264579   0.00000013
  16   0.00890411   0.00890416  -0.00000005
  17  -0.01577120  -0.01577122   0.00000002
  18   0.00409658   0.00409659  -0.00000001
  19   0.00825279   0.00825278   0.00000000
  20   0.00672859   0.00672859  -0.00000000
```

In the example above, the difference between the simulated and inferred innovations (`e - eInferred`) illustrates the transient effects introduced by the inference. When `garchsim` generates data, it generates sufficient initial data,

which it then discards, to allow transients to decay to some arbitrarily small value (see "Automatic Minimization of Transient Effects" on page 1-45). However, the inference function `garchinfer` (an interface to the log-likelihood objective function, `garchllfn`) must infer the innovations and conditional standard deviations directly from the observed returns. This can introduce transient effects.

That the first $R = 2$ rows of the inferred innovations are `0`, illustrates the link between simulation, inference, and estimation in the GARCH Toolbox. This fact is also directly related to the manner in which maximum likelihood estimation is performed.

Maximum Likelihood Estimation

Forming the log-likelihood objective function involves a two-step process:

1. `garchllfn` uses the conditional mean specification of ARMAX form shown in Eq. (1-16) to infer the innovations from the observed returns. This equation is derived from Eq. (1-8) by solving for ε_t.

$$\varepsilon_t = -C + y_t - \sum_{i=1}^{R} AR_i y_{t-i} - \sum_{j=1}^{M} MA_j \varepsilon_{t-j} - \sum_{k=1}^{Nx} \beta_k X(t,k) \qquad \textbf{(1-16)}$$

   To infer the innovations, `garchllfn` uses the Box and Jenkins conditional approach, which conditions the recursion on the initial R observations of y_t, setting the initial values of ε_t to `0` (see Hamilton [10], page 132, or Box, Jenkins, and Reinsel [7], page 237). Note that for the default model, $R = M = 0$, and no transients are induced due to this initialization.

2. `garchllfn` must then infer the conditional variances from the squared innovations as illustrated in Eq. (1-9), which is replicated here.

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2$$

   This step initializes the recursion by setting the first max(P,Q) observations of both σ_t^2 and ε_t^2 to the sample variance of the innovations inferred from the first step (see Hamilton [10], pages 666-667, or Bollerslev [4], page 316).

Forecasting

This section uses the estimated default model and the XYZ Corporation, from the section "Simulation" on page 1-41, to demonstrate the use of the forecasting function `garchpred`.

$$y_t = 0.00049183 + \varepsilon_t$$

$$\sigma_t^2 = 8.2736e{-}007 + 0.96283\sigma_{t-1}^2 + 0.03178\varepsilon_{t-1}^2$$

`garchpred` computes minimum-mean-square-error (MMSE) forecasts of the conditional mean and conditional standard deviation of the returns $\{y_t\}$ in each period over a user-specified forecast horizon.

Specifically, this section discusses:

- Computing a Forecast
- Computing Root Mean Square Errors (RMSE)
- Asymptotic Behavior for Long-Range Forecast Horizons

Note Example results in this section are printed in **Short E** numeric format for readability. Select **File** > **Preferences...** > **General** > **Short E** before starting the example to duplicate these printed results.

Computing a Forecast

This section discusses:

- Using Default Inputs
- Forecasting Over a Longer Horizon
- Long-Range Forecasting
- Forecasting Returns Over Multiple Periods

If the variables for the estimated model no longer exist in your workspace, then use the following commands to load the data and regenerate the estimation results of the default model. This example omits the estimation output to save space.

```
load xyz
xyz = price2ret(prices);
[coeff, errors, LLF, innovations, sigma] = garchfit(xyz);
```

Using Default Inputs

Now call `garchpred` to compute the conditional mean and standard deviation return forecasts for the XYZ Corporation using the default model parameter estimates. Provide the specification structure `coeff` (the output of `garchfit`) and the XYZ Corporation return series `xyz`, as input. Accept the `garchpred` default (`1`) for the number of forecast periods.

```
[sFcast, yFcast]  = garchpred(coeff, xyz);
[sFcast, yFcast]

ans =

  1.1670e-002  4.9183e-004
```

The result consists of the MMSE forecasts of the conditional standard deviations and the conditional mean of the return series `xyz` for a one-period default horizon.

Note `garchpred` allows the use of a time series regression matrix and an associated time series matrix of forecasted explanatory data. If you specify no regression matrix, the conditional mean has no regression component. See the section “Conditional Mean Models with Regression Components” on page 1-62 for information about using regression models.

Forecasting Over a Longer Horizon

To obtain information about asymptotic behavior, you need to forecast for more more than a single period. Use the following command to forecast the conditional mean and standard deviation in each period of a 10-period forecast horizon.

```
[sFcast, yFcast]  = garchpred(coeff, xyz, 10);
[sFcast, yFcast]
ans =
  1.1670e-002  4.9183e-004
```

```
    1.1674e-002  4.9183e-004
    1.1678e-002  4.9183e-004
    1.1682e-002  4.9183e-004
    1.1686e-002  4.9183e-004
    1.1690e-002  4.9183e-004
    1.1694e-002  4.9183e-004
    1.1697e-002  4.9183e-004
    1.1701e-002  4.9183e-004
    1.1705e-002  4.9183e-004
```

The results show that the default model forecast of the conditional mean is always C = 0.00049183. This is true for any forecast horizon because the expected value of any innovation, ε_t, is `0`.

In contrast, the conditional standard deviation forecast changes from period to period and approaches the unconditional standard deviation of $\{\varepsilon_t\}$, given by the square root of Eq. (1-7).

$$\sigma = \sqrt{\frac{\kappa}{1 - \sum_{i=1}^{P} G_i - \sum_{j=1}^{Q} A_j}} \qquad \textbf{(1-17)}$$

For this example, you can calculate the unconditional standard deviation of $\{\varepsilon_t\}$ as

```
s0 = sqrt(coeff.K/(1 - sum([coeff.GARCH(:) ; coeff.ARCH(:)])))
s0 =
  1.2393e-002
```

Plot the conditional standard deviations, `sigma`, derived from the fitted returns. The plot reveals that the most recent values of σ_t fall below this long-run, asymptotic value.

```
plot(sigma)
title('Fitted Conditional Standard Deviations: XYZ Corporation')
```

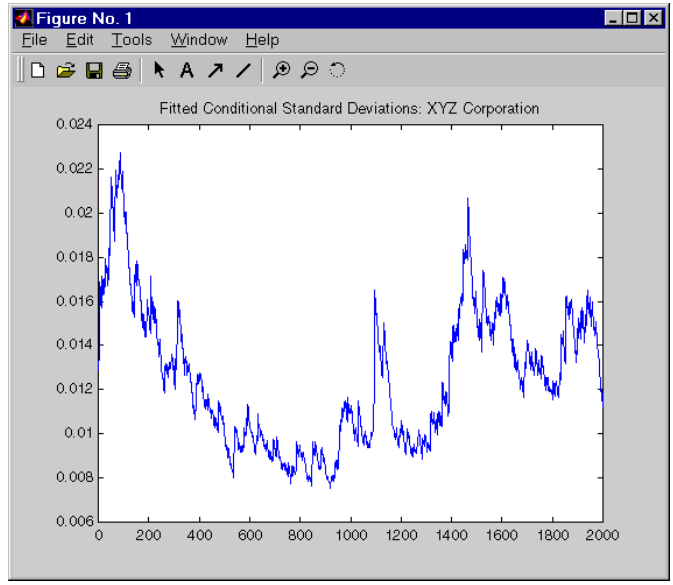


Figure 1-15: Fitted Conditional Standard Deviations

Long-Range Forecasting

That the most recent values of σ_t fall below `1.2393e-002` indicates that the long-range forecast of σ_t approaches this value from below. Confirm this by forecasting the standard deviations out 1000 periods, then plotting the forecasts (blue, dashed) and asymptotic value (red, solid) on the same graph.

```
sFcast = garchpred(coeff, xyz, 1000);
plot(sFcast, 'blue--')
hold('on')
plot([0 length(sFcast)], [s0 s0], 'red')
title('Standard Deviation Forecasts and Asymptotic Value: XYZ Corporation')
```

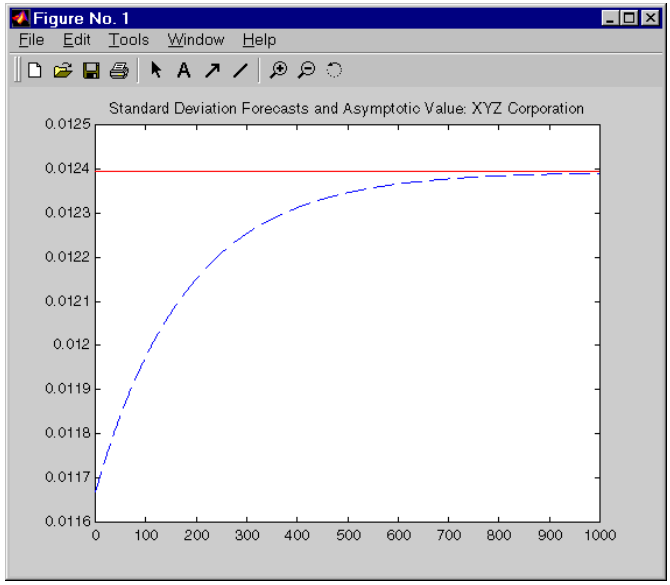


Figure 1-16: Standard Deviation Forecasts and Asymptotic Value

You can see from Figure 1-16, Standard Deviation Forecasts and Asymptotic Value that it takes a very long time for the forecast to reach its steady-state value. This is consistent with the high degree of persistence in the volatility process for the XYZ Corporation (see Figure 1-9, ACF of the Squared Returns).

Forecasting Returns Over Multiple Periods

In addition to computing conditional mean and volatility forecasts on a per-period basis, `garchpred` also computes volatility forecasts of returns for assets held for multiple periods. For example, to forecast the standard deviation of the return you would obtain if you purchased XYZ stock today and sold it 10 days from now,

```
[sFcast, yFcast, sTotal]  = garchpred(coeff, xyz, 10);
[sFcast, sTotal]
ans =
  1.1670e-002  1.1670e-002
  1.1674e-002  1.6506e-002
  1.1678e-002  2.0220e-002
  1.1682e-002  2.3352e-002
```

```
    1.1686e-002  2.6112e-002
    1.1690e-002  2.8609e-002
    1.1694e-002  3.0907e-002
    1.1697e-002  3.3047e-002
    1.1701e-002  3.5057e-002
    1.1705e-002  3.6959e-002
```

The vector `sTotal` (the second column above) represents the standard deviation forecasts of returns when the asset is held for multiple periods. The first element contains the standard deviation of the return expected if XYZ stock were held for one period, the second element contains the standard deviation of the return expected if XYZ stock were held for two periods, and so on. The last element contains the volatility forecast of the expected return if XYZ were purchased today and held for 10 periods.

If you convert the standard deviations `sFcast` and `sTotal` to variances by squaring each element, you can see an interesting relationship between the cumulative sum of `sFcast.^2` and `sTotal.^2`.

```
[cumsum(sFcast.^2)  sTotal.^2]
ans =
  1.3618e-004  1.3618e-004
  2.7246e-004  2.7246e-004
  4.0883e-004  4.0883e-004
  5.4530e-004  5.4530e-004
  6.8185e-004  6.8185e-004
  8.1850e-004  8.1850e-004
  9.5524e-004  9.5524e-004
  1.0921e-003  1.0921e-003
  1.2290e-003  1.2290e-003
  1.3660e-003  1.3660e-003
```

Although not exactly equivalent, this relationship in the presence of heteroskedasticity is similar to the familiar square-root-of-time rule for converting constant variances of uncorrelated returns expressed on a per-period basis to a variance over multiple periods. This relationship between `sFcast` and `sTotal` holds for the default conditional mean model only (i.e., the relationship is valid for uncorrelated returns).

Note that the calculation of `sTotal` is strictly correct for continuously compounded returns only, and is an approximation for periodically compounded returns.

Note The `sTotal` output of `garchpred` is not available for conditional mean models with regression components.

Computing Root Mean Square Errors (RMSE)

You can also use `garchpred` to calculate the root mean square errors (RMSE) associated with the conditional mean forecasts in `yFcast`.

```
[sFcast, yFcast, sTotal, yRMSE]  = garchpred(coeff, xyz, 10);
[yFcast, yRMSE]
ans =
  4.9183e-004  1.1670e-002
  4.9183e-004  1.1674e-002
  4.9183e-004  1.1678e-002
  4.9183e-004  1.1682e-002
  4.9183e-004  1.1686e-002
  4.9183e-004  1.1690e-002
  4.9183e-004  1.1694e-002
  4.9183e-004  1.1697e-002
  4.9183e-004  1.1701e-002
  4.9183e-004  1.1705e-002
```

The first column above contains the minimum mean square error (MMSE) forecasts of the conditional mean of the returns in each of the first 10 periods (from the section “Forecasting Over a Longer Horizon” on page 1-55). The second column contains the standard error of the corresponding forecast (see Baillie & Bollerslev [1], equation 19, page 96). You can use these results to construct approximate confidence intervals for conditional mean forecasts, with the approximation becoming more accurate during periods of relatively stable volatility (see Baillie & Bollerslev [1], and Bollerslev, Engle, and Nelson [6]). As heteroskedasticity in returns disappears (i.e., as the returns approach the homoskedastic, or constant variance, limit), the approximation is exact and you can apply the Box & Jenkins confidence bounds (see Box, Jenkins, and Reinsel [7], pages 133-145).

Note The `yRMSE` output of `garchpred` is not available for conditional mean models with regression components.

Asymptotic Behavior for Long-Range Forecast Horizons

If you are working with long-range forecast horizons, the following asymptotic behaviors hold for the outputs of `garchpred`:

- As mentioned earlier in this section, the conditional standard deviation forecast (i.e., the first `garchpred` output, `sFcast`) approaches the unconditional standard deviation of $\{\varepsilon_t\}$ given by the square root of Eq. (1-7).
- GARCH effects do not affect the MMSE forecast of the conditional mean (i.e., the second `garchpred` output, `yFcast`). The forecast approaches the unconditional mean of $\{y_t\}$ as in the constant variance case. That is, the presence of GARCH effects introduces dependence in the variance process, and only affects the uncertainty of the mean forecast, leaving the mean forecast itself unchanged.
- The mean-square-error of the conditional mean (i.e., the square of the fourth `garchpred` output, `yRMSE.^2`) approaches the unconditional variance of $\{y_t\}$.

Conditional Mean Models with Regression Components

The GARCH Toolbox allows conditional mean models with regression components, i.e., of general ARMAX(R,M,Nx) form.

$$y_t = C + \sum_{i=1}^{R} AR_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{M} MA_j \varepsilon_{t-j} + \sum_{k=1}^{Nx} \beta_k X(t,k)$$

Conditional mean models with a regression component introduce additional complexity in the sense that the GARCH Toolbox has no way of knowing what the explanatory data represents or how it was generated. This is in contrast to ARMA models, which have an explicit forecasting mechanism and well-defined stationarity/invertibility requirements.

All the primary functions in the GARCH Toolbox (i.e., `garchfit`, `garchinfer`, `garchpred`, and `garchsim`) accept an optional regression matrix `X`, which represents *X* in the equation above. You must ensure that the regression matrix you provide is valid and you must:

- Collect and format the past history of explanatory data you include in `X`
- As needed, forecast `X` into the future to form `XF`

This section discusses:

- Incorporating a Regression Model in an Estimation
- Simulation and Inference Using a Regression Component
- Forecasting Using a Regression Component
- Regression in a Monte Carlo Framework

Incorporating a Regression Model in an Estimation

This section uses the asymptotic equivalence of auto-regressive models and linear regression models to illustrate the use of a regression component in the GARCH Toolbox. The example is presented in two parts:

- Fitting an AR/GARCH Model to a Simulated Return Series
- Fitting a Regression Model to the Same Return Series

Fitting an AR/GARCH Model to a Simulated Return Series

This section defines a specification structure for an AR/GARCH model, and then uses that model to fit a simulated return series to the defined model.

Define the AR/GARCH Model. Start by creating a specification structure for an AR(2)/GARCH(1,1) composite model with successive calls to `garchset`. Set the `Display` flag to **`off`** to suppress the optimization details that `garchfit` normally prints to the screen.

```
spec = garchset('K', 0.005, 'GARCH', 0.7, 'ARCH', 0.1);
spec = garchset(spec, 'C', 0);
spec = garchset(spec, 'R', 2, 'AR', [0.5 -0.8]);
spec = garchset(spec, 'Regress', [0.5  -0.8])
spec = garchset(spec, 'Display', 'off');

spec =
         Comment: 'Mean: ARMAX(2,0,?); Variance: GARCH(1,1)'
               R: 2
               M: 0
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 0
              AR: [0.5000 -0.8000]
              MA: []
         Regress: [0.5000 -0.8000]
               K: 0.0050
           GARCH: 0.7000
            ARCH: 0.1000
            FixC: []
           FixAR: []
           FixMA: []
      FixRegress: []
            FixK: []
         FixGARCH: []
          FixARCH: []
    Optimization: [1x1 struct]
```

Notice that in this specification structure, spec:

- The model order fields `R`, `M`, `P`, and `Q` are consistent with the number of coefficients in the `AR`, `MA`, `GARCH`, and `ARCH` vectors, respectively.
- Although the `Regress` field indicates two regression coefficients, the `Comment` field still contains a question mark as a placeholder for the number of explanatory variables.
- There is no model order field for the `Regress` vector, analogous to the `R`, `M`, `P`, and `Q` orders of an ARMA(R,M)/GARCH(P,Q) model.

Fit the Model to a Simulated Return Series. Simulate 2000 observations of the innovations, conditional standard deviations, and returns for the AR(2)/GARCH(1,1) process defined in `spec`. Use the model defined in `spec` to estimate the parameters of the simulated return series and then compare the parameter estimates to the original coefficients in `spec`.

```
[e,s,y] = garchsim(spec, 2000);
[coeff, errors] = garchfit(spec, y);
garchdisp(coeff, errors)
```

```
  Number of Parameters Estimated: 6

                              Standard          T
  Parameter       Value         Error       Statistic
 -----------   -----------   ------------   -----------
          C    -0.00045653    0.0034627       -0.1318
      AR(1)    0.50256        0.013926        36.0875
      AR(2)    -0.80022       0.013987       -57.2134
          K    0.0049947      0.0019528        2.5577
   GARCH(1)    0.71232        0.094514         7.5366
    ARCH(1)    0.082964       0.022582         3.6740
```

The estimated parameters, shown in the `Value` column, are quite close to the original coefficients in `spec`.

Because you specified no explanatory regression matrix as input to `garchsim` and `garchfit`, these functions ignore the regression coefficients (`Regress`). Display the `Comment` field of the resulting `garchfit` output structure. It shows a `0` for the order of the regression component.

```
comment = garchget(coeff, 'Comment')
comment =
```

```
Mean: ARMAX(2,0,0); Variance: GARCH(1,1)
```

Fitting a Regression Model to the Same Return Series

To illustrate the use of a regression matrix, fit the return series y, an AR(2) process in the mean, to a regression model with two explanatory variables. The regression matrix consists of the first- and second-order lags of the simulated return series y.

Remove AR Component. First, remove the AR component from the specification structure.

```
spec = garchset(spec, 'R', 0, 'AR', [])
spec =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(1,1)'
               R: 0
               M: 0
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 0
              AR: []
              MA: []
         Regress: [0.5000 -0.8000]
               K: 0.0050
           GARCH: 0.7000
            ARCH: 0.1000
            FixC: []
           FixAR: []
           FixMA: []
      FixRegress: []
            FixK: []
         FixGARCH: []
          FixARCH: []
    Optimization: [1x1 struct]
```

Create the Regression Matrix. Create a regression matrix of first- and second-order lags using the simulated returns vector y as input. Examine the first 10 rows of y and the corresponding rows of the lags.

```
X = lagmatrix(y, [1 2]);
[y(1:10)  X(1:10,:)]
```

```
ans =
    0.0562       NaN       NaN
    0.0183    0.0562       NaN
   -0.0024    0.0183    0.0562
   -0.1506   -0.0024    0.0183
   -0.3937   -0.1506   -0.0024
   -0.0867   -0.3937   -0.1506
    0.1075   -0.0867   -0.3937
    0.2225    0.1075   -0.0867
    0.1044    0.2225    0.1075
    0.1288    0.1044    0.2225
```

A `NaN` (an IEEE arithmetic standard for Not-a-Number) in the resulting matrix `X` indicates the presence of a missing observation. If you use `X` to fit a regression model to `y`, `garchfit` produces an error.

```
[coeff, errors] = garchfit(spec, y, X);
??? Error using ==> garchfit
 Regression matrix 'X' has insufficient number of observations.
```

The error occurs because there are fewer valid rows (i.e., those rows without a `NaN`) in the regression matrix `X` than there are observations in `y`. The returns vector `y` has 2000 observations but the most recent number of valid observations in `X` is only 1998.

You can do one of two things to enable you to proceed. For a return series of this size it makes little difference which option you choose:

- Strip off the first two observations in `y`
- Replace all `NaN`s in `X` with some reasonable value

This example continues by replacing all `NaN`s with the sample mean of `y`. Use the MATLAB function `isnan` to identify `NaN`s and the function `mean` to compute the mean of `y`.

```
X(isnan(X)) = mean(y);
[y(1:10), X(1:10,:)]
ans =
    0.0562    0.0004    0.0004
    0.0183    0.0562    0.0004
   -0.0024    0.0183    0.0562
   -0.1506   -0.0024    0.0183
```

```
   -0.3937   -0.1506   -0.0024
   -0.0867   -0.3937   -0.1506
    0.1075   -0.0867   -0.3937
    0.2225    0.1075   -0.0867
    0.1044    0.2225    0.1075
    0.1288    0.1044    0.2225
```

Note If the number of valid rows in `X` exceeds the number of observations in `y`, then `garchfit` includes in the estimation only the most recent rows of `X`, equal to the number of observations in `y`.

Fit the Regression Model. Now that the explanatory regression matrix `X` is compatible with the return series vector `y`, use `garchfit` to estimate the model coefficients for the return series using the regression matrix and display the results.

```
[coeffX, errorsX] = garchfit(spec, y, X);
garchdisp (coeffX, errorsX)
```

```
  Number of Parameters Estimated: 6

                              Standard          T
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
         C    -0.00044818    0.0034618       -0.1295
Regress(1)    0.50257        0.01392          36.1049
Regress(2)    -0.8002        0.013981        -57.2344
         K    0.0050529      0.0019709         2.5637
  GARCH(1)    0.70955        0.095315          7.4443
   ARCH(1)    0.083293       0.022664          3.6751
```

These estimation results are similar to those shown for the AR model in the section "Fitting an AR/GARCH Model to a Simulated Return Series" on page 1-63. This similarity illustrates the asymptotic equivalence of auto-regressive models and linear regression models.

By illustrating the extra steps involved in formatting the explanatory matrix, this part of the example also highlights the additional complexity involved in modeling conditional means with regression components.

Simulation and Inference Using a Regression Component

Including a regression component with `garchsim` and `garchinfer` is similar to including one with `garchfit`.

For example, the following command simulates a single realization of 2000 observations of the innovations, conditional standard deviations, and returns. It uses the initial MATLAB default state as a random number generator seed, and incorporates the regression matrix `X`.

```
[e,s,y] = garchsim(spec, 2000, 1, [], X);
```

You can also use the same regression matrix `X` to infer the innovations and conditional standard deviations from the returns.

```
[eInfer, sInfer] = garchinfer(spec, y, X);
```

Forecasting Using a Regression Component

Inclusion of a regression component in forecasting is also similar to including one in an estimation. However, in addition to the explanatory data, you must consider the use of forecasted explanatory data.

This section discusses:

- Forecasted Explanatory Data
- Generating the Forecasted Explanatory Data
- Ordinary Least Squares Regression

Forecasted Explanatory Data

If you want to forecast the conditional mean of a return series `y` in each period of a 10-period forecast horizon, the correct calling syntax for `garchpred` is

```
[sFcast, yFcast] = garchpred(spec, y, 10, X, XF);
```

where `X` is the same regression matrix shown above, and `XF` is a regression matrix of forecasted explanatory data. In fact, `XF` represents a projection into the future of the same explanatory data found in `X`. Note that the command above produces an error if you execute it in your current workspace because `XF` is missing.

XF must have the same number of columns as X. In each column of XF, the first row contains the one-period-ahead forecast, the second row the two-period-ahead forecast, and so on. If you specify XF, the number of rows (forecasts) in each column of must equal or exceed the forecast horizon. When the number of forecasts in XF exceeds the 10-period forecast horizon, garchpred uses only the first 10 forecasts. If XF is empty ([]) or missing, the conditional mean forecast has no regression component.

You should use the same regression matrix X when calling garchpred that you used for simulation and/or estimation. This is because garchpred requires a complete conditional mean specification to correctly infer the innovations $\{\varepsilon_t\}$ from the observed return series $\{y_t\}$.

Forecasting the Conditional Standard Deviation. If you only need to forecast the conditional standard deviation (i.e., sFcast), XF is unnecessary. This is true even if you included the matrix X in the simulation and/or estimation process.

For example, you would use the following syntax to forecast only the conditional standard deviation of the return series y over a 10-period forecast horizon

```
sFcast = garchpred(spec, y, 10, X);
```

Forecasting the Conditional Mean. If you specify X, you must also specify XF to forecast the conditional mean (i.e., yFcast).

For example, to forecast the conditional mean of the return series y over a 10-period forecast horizon,

```
[sFcast yFcast] = garchpred(spec, y, 10, X, XF);
```

The forecasted explanatory data, XF, does not affect the standard deviation forecast. Note that this command produces an error if you execute it in your current workspace because XF is missing.

Generating the Forecasted Explanatory Data

Typically, the regression matrix X contains the observed returns of a suitable market index, collected over the same time interval as the observed data of interest. In this case, X is most likely a vector, corresponding to a single explanatory variable, and you must devise some way of generating the forecast of X (i.e., XF).

One approach, using the GARCH Toolbox, is to first use `garchfit` to fit a suitable ARMA(R,M) model to the returns in `X`, then use `garchpred` to forecast the market index returns into the future. Specifically, since you're not interested in fitting the volatility of `X`, you can simplify the estimation process by assuming a constant conditional variance model, i.e. ARMA(R,M)/ GARCH(0,0).

Ordinary Least Squares Regression

The following example illustrates an ordinary least squares regression by simulating a return series that scales the returns of the XYZ Corporation. It also provides an example of a constant conditional variance model. A model like this might, for example, represent a leveraged position in the common stock of the XYZ Corporation.

First, create a specification structure. Set the `Display` flag to **`off`** to suppress the optimization details that `garchfit` normally prints to the screen.

```
spec = garchset('Display', 'off');
spec = garchset(spec, 'P', 0, 'Q', 0);
spec = garchset(spec, 'C', 0, 'Regress', 1.2, 'K', 0.00015)
spec =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(0,0)'
               R: 0
               M: 0
               P: 0
               Q: 0
    Distribution: 'Gaussian'
               C: 0
              AR: []
              MA: []
         Regress: 1.2000
               K: 1.5000e-004
           GARCH: []
            ARCH: []
            FixC: []
           FixAR: []
           FixMA: []
      FixRegress: []
            FixK: []
        FixGARCH: []
```

```
        FixARCH: []
   Optimization: [1x1 struct]
```

Now, simulate a single realization of 2000 observations, fit the model, and examine the results

```
[e, s, y] = garchsim(spec, 2000, 1, [], xyz);
[coeff, errors] = garchfit(spec, y, xyz);
garchdisp(coeff, errors)
```

```
  Number of Parameters Estimated: 3

                              Standard          T
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
         C    -5.5043e-006   0.0002711       -0.0203
Regress(1)    1.2402         0.020454        60.6304
         K    0.0001464      4.6871e-006     31.2345
```

These estimation results are just the ordinary least squares (OLS) regression results. In fact, in the absence of GARCH effects and assuming Gaussian innovations, maximum likelihood estimation and least squares regression are the same thing.

Note This example is shown purely for illustration purposes. Although you can use the GARCH Toolbox to perform OLS, it is computationally inefficient and is not recommended.

Regression in a Monte Carlo Framework

In the general case, the functions `garchsim`, `garchinfer`, and `garchpred` process multiple realizations (i.e., sample paths) of univariate time series. That is, the outputs of `garchsim`, as well as the observed return series input to `garchpred` and `garchinfer`, can be matrices in which each column represents an independent realization. `garchfit` is different, in that the input observed return series of interest must be a vector (i.e., a single realization).

When simulating, inferring, and forecasting multiple realizations, the appropriate toolbox function applies a given regression matrix `X` to each

realization of a univariate time series. For example, in the following command, `garchsim` applies a given `X` matrix to all 10 columns of the output series $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$.

```
[e,s,y] = garchsim(spec, 100, 10, [], X);
```

In a true Monte Carlo simulation of the above process, including a regression component, you would call `garchsim` inside a loop 10 times, once for each path. Each iteration would pass in a unique realization of `X` and produce single-column outputs.

Model Selection and Analysis

The GARCH Toolbox offers a number of model selection tools.

The section "Analysis and Estimation Example Using the Default Model" on page 1-16 illustrates the use of the autocorrelation (`autocorr`) and partial autocorrelation (`parcorr`) functions as qualitative guides in the process of model selection and assessment. It also introduces the `archtest` and `lbqtest` hypothesis testing functions.

This section discusses:

- Likelihood Ratio Tests (`lratiotest`)
- Akaike and Bayesian Information Criteria (`aicbic`)
- Equality Constraints and Parameter Significance
- Equality Constraints and Initial Parameter Estimates

The examples that follow again rely on the daily returns of the XYZ Corporation. If the variables no longer exist in your MATLAB workspace, you can recreate them with the commands,

```
load xyz
xyz = price2ret(prices);
```

Likelihood Ratio Tests

The section "Analysis and Estimation Example Using the Default Model" on page 1-16 demonstrates that the default GARCH(1,1) model explains most of the variability of the returns of the XYZ Corporation. This example uses the function `lratiotest` to determine if evidence exists to support the use of a GARCH(2,1) model.

The example first fits the return series of the XYZ Corporation to the default GARCH(1,1) model. It then overfits the same series using the following, more elaborate, GARCH(2,1) model.

$$y_t = C + \varepsilon_t$$

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + G_2\sigma_{t-2}^2 + A_1\varepsilon_{t-1}^2$$

The example is presented in two parts:

- Estimate Parameters for the GARCH(1,1) and GARCH(2,1) Models
- Perform the Likelihood Ratio Test

Estimate Parameters for the GARCH(1,1) and GARCH(2,1) Models

The GARCH(1,1) Model. First, create a GARCH(1,1) default model with the `Display` flag set to **`off`**. Then, estimate the model and display the results, including the maximized log-likelihood function value.

```
spec11 = garchset('Display', 'off', 'P', 1, 'Q', 1);
[coeff11, errors11, LLF11, innovations11, sigma11, summary11] = garchfit(spec11, xyz);
garchdisp(coeff11, errors11)

  Number of Parameters Estimated: 4

                              Standard          T
  Parameter       Value          Error       Statistic
 -----------   -----------   ------------   -----------
          C    0.00049183     0.00025585       1.9223
          K    8.2736e-007    2.7446e-007      3.0145
   GARCH(1)    0.96283        0.0051557      186.7500
    ARCH(1)    0.03178        0.004416         7.1965

LLF11
LLF11 =
  5.9746e+003
```

Note that a more accurate value of `LLF11` is `5974.6025`.

The GARCH(2,1) Model. Create a GARCH(2,1) specification structure. Again, set the `Display` flag to **`off`**.

```
spec21 = garchset('Display', 'off', 'P', 2, 'Q', 1)
spec21 =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(2,1)'
               R: 0
               M: 0
               P: 2
               Q: 1
```

```
      Distribution: 'Gaussian'
                 C: []
                AR: []
                MA: []
           Regress: []
                 K: []
             GARCH: []
              ARCH: []
              FixC: []
             FixAR: []
             FixMA: []
         FixRegress: []
               FixK: []
           FixGARCH: []
            FixARCH: []
      Optimization: [1x1 struct]
```

Now estimate the GARCH(2,1) model and display the results, including the maximized log-likelihood function value.

```
[coeff21,errors21,LLF21,innovations21,sigma21,summary21] = garchfit(spec21, xyz);
garchdisp(coeff21, errors21)

  Number of Parameters Estimated: 5

                              Standard          T
  Parameter       Value          Error       Statistic
 -----------   -----------   ------------   -----------
          C    0.00049584     0.000256         1.9369
          K    1.3645e-006    4.6186e-007      2.9545
   GARCH(1)    0.0358         0.028327         1.2638
   GARCH(2)    0.90149        0.029642        30.4131
    ARCH(1)    0.05379        0.0073393        7.3291

LLF21
LLF21 =
  5.9759e+003
```

A more accurate value of `LLF21` is `5975.8927`.

Perform the Likelihood Ratio Test

Of the two models associated with the same return series:

- The default GARCH(1,1) model is a restricted model. That is, you can interpret a GARCH(1,1) model as a GARCH(2,1) model with the restriction that $G_2 = 0$.
- The more elaborate GARCH(2,1) model is an unrestricted model.

Since `garchfit` enforces no boundary constraints during either of the two estimations, you can apply a likelihood ratio test (LRT) (see Hamilton [10], pages 142-144).

In this context, the unrestricted GARCH(2,1) model serves as the alternative hypothesis (i.e., the hypothesis the example gathers evidence to support), while the restricted GARCH(1,1) model serves as the null hypothesis (i.e., the hypothesis the example assumes is true, lacking any evidence to support the alternative).

The LRT statistic is asymptotically Chi-Square distributed with degrees-of-freedom equal to the number of restrictions imposed. Since the GARCH(1,1) model imposes one restriction, specify one degrees-of-freedom in your call to `lratiotest`. Test the models at the 0.05 significance level.

```
[H, pValue, Stat, CriticalValue] = lratiotest(LLF21, LLF11, 1, 0.05);
[H  pValue  Stat  CriticalValue]
ans =
         0    0.1082    2.5806    3.8415
```

`H = 0` indicates that there is insufficient statistical evidence in support of the GARCH(2,1) model. The conclusion is that the default GARCH(1,1) model adequately explains the variability in the return series when compared to a more elaborate GARCH(2,1) model.

Akaike and Bayesian Information Criteria

You can also use Akaike (AIC) and Bayesian (BIC) information criteria to compare alternative models. Since information criteria penalize models with additional parameters, the AIC and BIC model-order-selection criteria are based on parsimony (see Box, Jenkins, and Reinsel [7], pages 200-201).

The following example uses the default GARCH(1,1) and GARCH(2,1) models developed in the previous section, “Likelihood Ratio Tests” on page 1-73. It is presented in two parts:

- Counting Estimated Parameters
- Computing the AIC and BIC Criteria

Counting Estimated Parameters

For both AIC and BIC, you need to provide the number of parameters estimated in the model. For the relatively simple models in the previous example, you can just count the number of parameters. The GARCH(2,1) model estimated five parameters (C, κ, G_1, G_2, and A_1), and GARCH(1,1) model estimated four parameters (C, κ, G_1, and A_1).

Use the function `garchcount` for more elaborate models. `garchcount` accepts the output specification structure created by `garchfit` and returns the number of parameters in the model defined in that structure.

```
n21 = garchcount(coeff21)
n21 =
     5
n11 = garchcount(coeff11)
n11 =
     4
```

Computing the AIC and BIC Criteria

Now use the function `aicbic` to compute the AIC and BIC statistics for the GARCH(2,1) model and the GARCH(1,1) model. Note that for the BIC statistic, you must also specify the number of observations in the return series.

```
[AIC, BIC] = aicbic(LLF21, n21, 2000);
[AIC   BIC]
ans =
  1.0e+004 *
   -1.1942   -1.1914
```

More accurate values are `AIC = -11941.7855` and `BIC = -11913.7810`.

```
[AIC, BIC] = aicbic(LLF11, n11, 2000);
[AIC   BIC]
ans =
  1.0e+004 *
   -1.1941   -1.1919
```

More accurate values are `AIC = -11941.2049` and `BIC = -11918.8013`

You can use the relative values of the AIC and BIC statistics as guides in the model selection process. In this example, the AIC criterion favors the

GARCH(2,1) model, while the BIC criterion favors the GARCH(1,1) default model with fewer parameters. Notice that since BIC imposes a greater penalty for additional parameters than does AIC, BIC always provides a model with a number of parameters no greater than that chosen by AIC.

Equality Constraints and Parameter Significance

The GARCH Toolbox lets you set and constrain model parameters as a way of assessing the parameters' significance.

This section:

- Shows you how to use the specification structure to fix individual parameters.
- Provides an example that demonstrates the use of equality constraints.

The Specification Structure Fix Fields

Each of the coefficient fields `C`, `AR`, `MA`, `Regress`, `K`, `GARCH`, and `ARCH`, in the specification structure, has a corresponding Boolean field that lets you hold any individual parameter fixed. These fix fields are `FixC`, `FixAR`, `FixMA`, `FixRegress`, `FixK`, `FixGARCH`, and `FixARCH`. For example, look at the output structure from the GARCH(2,1) estimation in the section "Likelihood Ratio Tests" on page 1-73.

```
coeff21
coeff21 =
         Comment: 'Mean: ARMAX(0,0,0); Variance: GARCH(2,1)'
               R: 0
               M: 0
               P: 2
               Q: 1
    Distribution: 'Gaussian'
               C: 4.9584e-004
              AR: []
              MA: []
         Regress: []
               K: 1.3645e-006
           GARCH: [0.0358 0.9015]
            ARCH: 0.0538
             FixC: []
            FixAR: []
```

```
          FixMA: []
     FixRegress: []
           FixK: []
       FixGARCH: []
        FixARCH: []
   Optimization: [1x1 struct]
```

Each fix field, when not empty ([]), is the same size as the corresponding coefficient field. A `0` in a particular element of a fix field indicates that the corresponding element of its companion value field is an initial parameter guess that `garchfit` refines during the estimation process. A `1` indicates that `garchfit` holds the corresponding element of its value field fixed during the estimation process (i.e., an equality constraint).

The GARCH(2,1) Model as an Example

This example uses the GARCH(2,1) model above to demonstrate the use of equality constraints. First, display the estimation results for the model.

```
garchdisp(coeff21, errors21)

  Number of Parameters Estimated: 5

                            Standard          T
 Parameter       Value        Error       Statistic
-----------   -----------   ------------   -----------
         C    0.00049584     0.000256        1.9369
         K    1.3645e-006    4.6186e-007     2.9545
  GARCH(1)    0.0358         0.028327        1.2638
  GARCH(2)    0.90149        0.029642       30.4131
   ARCH(1)    0.05379        0.0073393       7.3291
```

The T-statistic column is the parameter value divided by the standard error, and is normally distributed for large samples. The T-statistic measures the number of standard deviations the parameter estimate is away from zero, and as a general rule, a T-statistic greater than 2 in magnitude corresponds to approximately a 95 percent confidence interval. The T-statistics in the table above imply that the conditional mean constant (`C`) is on the edge of significance. They also imply that the `GARCH(1)` parameter adds little if any explanatory power to the model.

The GARCH(1) Parameter. Constrain the GARCH(1) parameter at 0 to assess its significance.

```
specG1 = garchset(coeff21, 'GARCH', [0 0.9], 'FixGARCH', [1 0])
specG1 =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(2,1)'
               R: 0
               M: 0
               P: 2
               Q: 1
    Distribution: 'Gaussian'
               C: 4.9584e-004
              AR: []
              MA: []
         Regress: []
               K: 1.3645e-006
           GARCH: [0 0.9000]
            ARCH: 0.0538
            FixC: []
           FixAR: []
           FixMA: []
     FixRegress: []
           FixK: []
        FixGARCH: [1 0]
         FixARCH: []
    Optimization: [1x1 struct]
```

The specG1 structure field FixGARCH indicates that garchfit holds GARCH(1) fixed at 0, and refines GARCH(2) from an initial value of 0.9 during the estimation process. In other words, the specG1 specification structure tests the composite model,

$$y_t = C + \varepsilon_t$$

$$\sigma_t^2 = \kappa + G_2\sigma_{t-2}^2 + A_1\varepsilon_{t-1}^2$$

Now estimate the model subject to the equality constraint and display the results.

```
[coeffG1,errorsG1,LLFG1,innovationsG1,sigmaG1] = garchfit(specG1, xyz);
garchdisp(coeffG1, errorsG1)
```

```
 Number of Parameters Estimated: 4

                             Standard          T
 Parameter       Value         Error       Statistic
-----------   -----------   ------------   -----------
         C    0.00052356     0.00025499       2.0532
         K    1.6865e-006    4.6547e-007      3.6231
  GARCH(1)    0              Fixed            Fixed
  GARCH(2)    0.93442        0.0085294      109.5531
   ARCH(1)    0.054718       0.0072265        7.5719

LLFG1
LLFG1 =
  5.9738e+003
```

A more accurate value of `LLFG1` is `5973.7872`.

Notice that the standard error and T-statistic columns for the first GARCH parameter indicate that `garchfit` held the GARCH(1) parameter fixed. The number of estimated parameters also decreased from 5 in the original, unrestricted GARCH(2,1) model to 4 in this restricted GARCH(2,1) model.

Apply the likelihood ratio test as before.

```
[H, pValue, Stat, CriticalValue] = lratiotest(LLF21, LLFG1, 1, 0.05);
[H  pValue  Stat  CriticalValue]
ans =
    1.0000    0.0402    4.2112    3.8415
```

The results support rejection of the simpler, restricted model at the 0.05 significance level, but just barely. The P-value indicates that had you tested at a significance level of 0.04 or less, the restricted model would have been accepted.

The GARCH(2) Parameter. As a second example, assess the significance of the GARCH(2) parameter by setting it to `0`.

```
specG2 = garchset(coeff21, 'GARCH', [0.9 0], 'FixGARCH', [0 1])
specG2 =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(2,1)'
               R: 0
               M: 0
               P: 2
               Q: 1
    Distribution: 'Gaussian'
```

```
                C: 4.9584e-004
               AR: []
               MA: []
          Regress: []
                K: 1.3645e-006
           GARCH: [0.9000 0]
            ARCH: 0.0538
             FixC: []
            FixAR: []
            FixMA: []
       FixRegress: []
             FixK: []
         FixGARCH: [0 1]
          FixARCH: []
     Optimization: [1x1 struct]
```

The specG2 structure field FixGARCH indicates that garchfit holds GARCH(2) fixed at 0, and refines GARCH(1) from an initial value of 0.9 during the estimation process. In other words, the specG2 specification structure tests the composite model,

$$y_t = C + \varepsilon_t$$

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + A_1\varepsilon_{t-1}^2$$

which is really the GARCH(1,1) default model.

Now, estimate the model subject to the equality constraint and display the results.

```
[coeffG2,errorsG2,LLFG2,innovationsG2,sigmaG2] = garchfit(specG2, xyz);
garchdisp(coeffG2, errorsG2)
```

```
  Number of Parameters Estimated: 4

                              Standard          T
  Parameter       Value         Error       Statistic
 -----------   -----------   ------------   -----------
          C    0.00048996     0.00025618       1.9126
          K    7.9828e-007    2.6908e-007      2.9667
   GARCH(1)    0.9636         0.0050784      189.7460
   GARCH(2)    0              Fixed            Fixed
    ARCH(1)    0.031239       0.0043564        7.1709
```

```
LLFG2
LLFG2 =
  5.9746e+003
```

Note that a more accurate value of `LLFG2` is `5974.6058`.

Finally, apply the likelihood ratio test again,

```
[H, pValue, Stat, CriticalValue] = lratiotest(LLF21, LLFG2, 1, 0.05);
[H  pValue  Stat  CriticalValue]
ans =
         0    0.1086    2.5738    3.8415
```

In this case, the results support acceptance of the restricted model at the 0.05 significance level. In fact, the P-value indicates that the test would support acceptance at the 0.10 significance level as well. This result again emphasizes that the default GARCH(1,1) model adequately explains the variation in the observed returns. A close examination reveals that the log-likelihood function values of the two models are nearly identical (i.e., `LLFG2 = 5974.6058`, `LLF11 = 5974.6025`).

Equality Constraints and Initial Parameter Estimates

This section highlights some important points regarding equality constraints and initial parameter estimates in the GARCH Toolbox. It discusses:

- Complete Model Specification
- Empty Fix Fields
- Number of Equality Constraints

Complete Model Specification

To set equality constraints during estimation, you must provide a complete model specification. The only flexibility in this regard is that you can decouple the model specification for the conditional mean from the model specification for the conditional variance.

The following example demonstrates an attempt to set equality constraints for an incomplete conditional mean model and a complete variance model. Create an ARMA(1,1)/GARCH(1,1) specification structure for conditional mean and variance models, respectively.

```
spec = garchset('R', 1, 'M', 1, 'P', 1, 'Q', 1);
spec = garchset(spec, 'C', 0, 'AR', 0.5, 'FixAR', 1);
```

```
spec = garchset(spec, 'K', 0.0005, 'GARCH', 0.8, 'ARCH', 0.1, 'FixGARCH', 1)
spec =
         Comment: 'Mean: ARMAX(1,1,?); Variance: GARCH(1,1)'
               R: 1
               M: 1
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 0
              AR: 0.5000
              MA: []
         Regress: []
               K: 5.0000e-004
           GARCH: 0.8000
            ARCH: 0.1000
            FixC: []
           FixAR: 1
           FixMA: []
      FixRegress: []
            FixK: []
         FixGARCH: 1
          FixARCH: []
    Optimization: [1x1 struct]
```

The conditional mean model is incomplete because the `MA` field is still empty. Since the requested ARMA(1,1) model is an incomplete conditional mean specification, `garchfit` ignores the `C`, `AR`, and `FixAR` fields, computes initial parameter estimates, and overwrites any existing parameters in the incomplete conditional mean specification. It also estimates all conditional mean parameters (i.e., `C`, `AR`, and `MA`) and ignores the request to constrain the `AR` parameter.

However, since the structure explicitly sets all fields in the conditional variance model, `garchfit` uses the specified values of `K` and `ARCH` as initial estimates subject to further refinement, and holds the `GARCH` parameter at 0.8 throughout the optimization process.

Empty Fix Fields

Any fix field that you leave empty (`[]`), is equivalent to a vector of zeros of compatible length. That is, when `garchfit` encounters an empty fix field, it automatically estimates the corresponding parameter. For example, the following specification structures produce the same GARCH(1,1) estimation results.

```
spec1 = garchset('K', 0.005, 'GARCH', 0.8, 'ARCH', 0.1, 'FixGARCH', 0, 'FixARCH', 0)
```

```
spec2 = garchset('K', 0.005, 'GARCH', 0.8, 'ARCH', 0.1)
```

Number of Equality Constraints

Avoid setting several equality constraints simultaneously. Although the ability to set equality constraints is both convenient and useful, equality constraints complicate the estimation process. For example, if you really want to estimate a GARCH(1,1) model, then specify a GARCH(1,1) model instead of a more elaborate model with numerous constraints.

Recommendations and Suggestions

This final section of the tutorial highlights some general recommendations to make it easier for you to use the GARCH Toolbox. It discusses:

- Simplicity/Parsimony
- Convergence Issues
- Initial Parameter Estimates
- Boundary Constraints and Statistical Inferences
- Data Size and Quality

Simplicity/Parsimony

Specify the smallest, most simplistic models that adequately describe your data. This is especially relevant for estimation. Simple models are easier to estimate, easier to forecast, and easier to analyze. In fact, certain model selection criteria, such as the AIC/BIC discussed in the section "Model Selection and Analysis" on page 1-73, penalize models for their complexity.

The section "Analysis and Estimation Example Using the Default Model" on page 1-16, examines the autocorrelation function (ACF) and partial autocorrelation function (PACF) of the XYZ Corporation. The results support the use of a simple constant for the conditional mean model as adequate to describe the data.

The following example illustrates an unnecessarily complicated model specification. It uses an ARMA(1,1)/GARCH(1,1) composite model, rather than a simple constant with GARCH(1,1) innovations, to estimate the model parameters for the returns of the XYZ Corporation.

Create a specification structure for an ARMA(1,1)/GARCH(1,1) model. Set the `Display` flag to **`off`** to suppress the optimization details that `garchfit` normally prints to the screen.

```
spec = garchset;
spec = garchset(spec, 'Display', 'off', 'R', 1, 'M', 1)
spec =
         Comment: 'Mean: ARMAX(1,1,?); Variance: GARCH(1,1)'
               R: 1
               M: 1
               P: 1
```

```
              Q: 1
   Distribution: 'Gaussian'
              C: []
             AR: []
             MA: []
        Regress: []
              K: []
          GARCH: []
           ARCH: []
           FixC: []
          FixAR: []
          FixMA: []
     FixRegress: []
           FixK: []
       FixGARCH: []
        FixARCH: []
   Optimization: [1x1 struct]
```

Now, estimate the model and examine the results.

```
[coeff,errors,LLF,innovations,sigma,summary] = garchfit(spec, xyz);
garchdisp(coeff, errors)
```

```
  Number of Parameters Estimated: 6

                              Standard          T
  Parameter       Value         Error       Statistic
 -----------   -----------   ------------   -----------
          C    0.00088504     0.00046465       1.9048
     AR(1)    -0.76595       0.098721        -7.7587
     MA(1)    0.80041        0.09305          8.6020
          K    7.9417e-007    2.7078e-007      2.9329
   GARCH(1)    0.96313        0.0051048      188.6716
    ARCH(1)    0.031735       0.0043606        7.2775
```

These results imply that the ARMA(1,1)/GARCH(1,1) composite model that best fits the observed data is

$$y_t = 0.00088504 - 0.76595\,y_{t-1} + \varepsilon_t + 0.80041\varepsilon_{t-1}$$

$$\sigma_t^2 = 7.9417e{-}007 + 0.96313\sigma_{t-1}^2 + 0.031735\varepsilon_{t-1}^2$$

However, close examination of the conditional mean equation reveals that the AR(1) and MA(1) parameters are almost identical. In fact, rewriting the mean equation in backshift (i.e., lag) operator notation, where $By_t = y_{t-1}$,

$$(1 + 0.76595B)y_t = 0.00088504 + (1 + 0.80041B)\varepsilon_t$$

the auto-regressive and moving-average polynomials come close to canceling each other (see Box, Jenkins, Reinsel [7], pages 263-267). This is an example of parameter redundancy, or pole-zero cancellation. It implies that you can use the default model simple white noise process to approximate the conditional mean model.

In fact, from the section "Analysis and Estimation Example Using the Default Model" on page 1-16, the default model that best fits the observed data is

$$y_t = 0.00049183 + \varepsilon_t$$

$$\sigma_t^2 = 8.2736e{-}007 + 0.96283\sigma_{t-1}^2 + 0.03178\varepsilon_{t-1}^2$$

Note that the long-run (i.e., unconditional) mean and variance forecasts of each model are in very close agreement.

However, notice that the AR(1) and MA(1) T-statistics provide a misleading impression, implying that the parameters are highly significant. In fact, the more elaborate ARMA(1,1) model only complicates the analysis by requiring the estimation of two additional parameters. If you evaluate the information criteria, both AIC and BIC favor the default model (BIC is more decisive), and the LRT with two degrees-of-freedom fails to reject the default model.

Convergence Issues

When estimating the parameters of a composite conditional mean/variance model, you may occasionally encounter convergence problems. For example, the estimation may appear to stall, showing little or no progress. It may terminate prematurely prior to convergence. Or, it may converge to an unexpected, suboptimal solution.

You can avoid many of these difficulties by performing a sound, pre-fit analysis as outlined in the section "Analysis and Estimation Example Using the Default

Model" on page 1-16. That section discusses graphical techniques (plotting the return series, examining the ACF and PACF), as well as some preliminary tests, including Engle's ARCH test and the Q-test. In addition, the section "GARCH Limitations" on page 1-3 mentions some of the limitations of GARCH models. In particular, it notes that GARCH techniques do not easily capture wild, spurious swings in a return series.

The most effective way of avoiding convergence problems is to select the most simplistic model that adequately describes your data. In fact, extreme difficulty in convergence is an indication that the model you chose does not describe your data well.

Specification Structure Fields That Affect Convergence

If you believe that your model is appropriate, and you still experience convergence problems during estimation, there are several fields in the specification structure that you can modify. The specification structure fields that affect convergence for the estimation function are `MaxIter`, `MaxFunEvals`, `TolCon`, `TolFun`, and `TolX`.

MaxIter and MaxFunEvals. `MaxIter` is the maximum number of iterations allowed in the estimation process. Each iteration involves an optimization phase in which garchfit suitably modifies calculations such as line search, gradient, and step size. The default value of `MaxIter` is 400. Although an estimation rarely exceeds `MaxIter`, you can increase the value if you suspect the estimation terminated prematurely.

`MaxFunEvals`, a field closely related to `MaxIter`, specifies the maximum number of log-likelihood objective function evaluations. The default value is 100 times the number of parameters estimated in the model. For example, the default model has four parameters, and so the default value of `MaxFunEvals` is 400. When the estimation process terminates prematurely, it is usually because `MaxFunEvals`, rather than `MaxIter`, is exceeded. You can increase `MaxFunEvals` if you suspect the estimation terminated prematurely.

TolCon, TolFun, and TolX. The fields `TolCon`, `TolFun`, and `TolX` are tolerance-related parameters that directly influence how and when convergence is achieved.

`TolCon` is the termination tolerance placed on violations of the stationarity and positivity constraints, and represents the maximum value by which parameter estimates can violate a constraint and still allow successful convergence. See

Eq. (1-6) in the section “Homoskedasticity of the Unconditional Variance” on page 1-8 for information about these constraints.

`TolFun` is the termination tolerance placed on the log-likelihood objective function. Successful convergence occurs when the log-likelihood function value changes by less than `TolFun`.

`TolX` is the termination tolerance placed on the estimated parameter values. Similar to `TolFun`, successful convergence occurs when the parameter values change by less than `TolX`.

`TolCon`, `TolFun`, and `TolX` have the same default value, `1e-006`. If you experience extreme difficulty in convergence (e.g., the estimation shows little or no progress, or shows progress but stops early), then increasing one or more of these parameter values (e.g., from `1e-006` to `1e-004`) may allow the estimation to converge. If the estimation appears to converge to a suboptimal solution, then decreasing one or more of these parameter values (e.g., from `1e-006` to `1e-007`) may provide more accurate parameter estimates.

Determining Convergence Status

There are two ways to determine whether an estimation achieves convergence. The first, and easiest, is to examine the optimization details of the estimation. By default, `garchfit` displays this information in the MATLAB command window. The second way to determine convergence status is to request the `garchfit` optional summary output.

To illustrate these methods, revisit the default model for the XYZ Corporation.

```
[coeff, errors, LLF, innovations, sigma, summary] = garchfit(xyz);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Diagnostic Information

Number of variables: 4

Functions
 Objective:                            garchllfn
 Gradient:                             finite-differencing
 Hessian:                              finite-differencing (or Quasi-Newton)
 Nonlinear constraints:                garchnlc
 Gradient of nonlinear constraints:    finite-differencing

Constraints
 Number of nonlinear inequality constraints: 0
 Number of nonlinear equality constraints:   0
```

```
 Number of linear inequality constraints:    1
 Number of linear equality constraints:      0
 Number of lower bound constraints:          4
 Number of upper bound constraints:          0

Algorithm selected
   medium-scale

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 End diagnostic information

                                max                     Directional
 Iter  F-count    f(x)         constraint    Step-size     derivative   Procedure
    1      5    -5921.94  -1.684e-005            1      -7.92e+004
    2     34    -5921.94  -1.684e-005    1.19e-007            -553
    3     43    -5924.42  -1.474e-005        0.125           -31.2
    4     49    -5936.16  -6.996e-021            1            -288
    5     57    -5960.62            0         0.25            -649
    6     68    -5961.45  -4.723e-006       0.0313           -17.3
    7     75    -5963.18  -2.361e-006          0.5           -28.6
    8     81    -5968.24            0            1             -55
    9     90    -5970.54  -6.016e-007        0.125            -196
   10    103    -5970.84  -1.244e-006      0.00781           -16.1
   11    110    -5972.77  -9.096e-007          0.5           -34.4
   12    126    -5972.77  -9.354e-007     0.000977           -24.5
   13    134    -5973.29   -1.05e-006         0.25           -4.97
   14    141    -5973.95  -6.234e-007          0.5           -1.99
   15    147    -5974.21  -1.002e-006            1          -0.641
   16    153    -5974.57  -9.028e-007            1         -0.0803
   17    159    -5974.59  -8.054e-007            1         -0.0293
   18    165     -5974.6  -8.305e-007            1         -0.0039
   19    172     -5974.6  -8.355e-007          0.5       -0.000964
   20    192     -5974.6  -8.355e-007    -6.1e-005       -0.000646
   21    212     -5974.6  -8.355e-007    -6.1e-005       -0.000996    Hessian modified twice
   22    219     -5974.6  -8.361e-007          0.5       -0.000184
   23    239     -5974.6  -8.361e-007    -6.1e-005        -0.00441    Hessian modified twice
Optimization terminated successfully:
 Search direction less than 2*options.TolX and
  maximum constraint violation is less than options.TolCon
 No Active Constraints
```

Notice that the optimization details indicate successful termination. Now, examine the summary output structure.

```
summary
summary =
          warning: 'No Warnings'
```

```
          converge: 'Function Converged to a Solution'
         covMatrix: [4x4 double]
        iterations: 23
     functionCalls: 241
       constraints: 'No Boundary Constraints'
```

The `converge` field indicates successful convergence. If the estimation failed to converge, the `converge` field would contain the message, `'Function Did NOT Converge'`. If the number of iterations or function evaluations exceeded its specified limits, the `converge` field would contain the message, `'Maximum Function Evaluations or Iterations Reached'`. The summary structure also contains fields that indicate the number of iterations (`iterations`) and log-likelihood function evaluations (`functionCalls`).

These are generic suggestions. The default values of `MaxIter`, `MaxFunEvals`, `TolCon`, `TolFun`, and `TolX` typically provide acceptable estimation results. For additional details, see the *Optimization Toolbox User's Guide* and, in particular, the reference section for the function `fmincon`.

Initial Parameter Estimates

Although `garchfit` computes initial parameter estimates if you provide none, at times it may be helpful to compute and specify your own initial guesses to avoid convergence problems.

Note If you specify initial estimates, you must provide complete conditional mean and/or variance model specifications. See the section “Equality Constraints and Initial Parameter Estimates” on page 1-83 for more information.

Partial Estimation

An important property of a conditionally Gaussian innovations process is that the parameters of the conditional mean and the conditional variance are asymptotically uncorrelated (see Bollerslev [4], pages 315-317, Engle [8], pages 994-997, and Gourieroux, [9], pages 43-51). You can estimate initial parameter estimates of the mean separately from those of the variance, breaking the composite estimation process into two parts.

For example, if the conditional mean is an ARMAX model, you can first estimate the ARMAX parameters assuming a constant variance innovations process (i.e., a GARCH(0,0) conditional variance model). The sample variance of the estimated residuals is then an approximation of the unconditional variance of the innovations process $\{\varepsilon_t\}$. Finally, based on reasonable parameter values of the GARCH and ARCH parameters of the conditional variance model, you can apply Eq. (1-7) to estimate the conditional variance constant κ.

For the common GARCH(1,1) model with Gaussian innovations,

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + A_1\varepsilon_{t-1}^2$$

it often turns out that you can obtain reasonable initial estimates by assuming G_1 is approximately 0.8 to 0.9, and A_1 is approximately 0.05 to 0.10.

Iterative Estimation

Another approach is to estimate the complete model, examine the results, then modify the parameter estimates as initial guesses for another round of estimation. For example, suppose you have already estimated a composite ARMA(1,1)/GARCH(1,1) model.

```
coeff
coeff =
         Comment: 'Mean: ARMAX(1,1,0); Variance: GARCH(1,1)'
               R: 1
               M: 1
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: 1.0000e-004
              AR: 0.5000
              MA: 0.4000
         Regress: []
               K: 5.0000e-006
           GARCH: 0.4000
            ARCH: 0.5000
            FixC: []
           FixAR: []
           FixMA: []
```

```
      FixRegress: []
            FixK: []
         FixGARCH: []
          FixARCH: []
    Optimization: [1x1 struct]
```

As you examine the above `coeff` structure (i.e., the first output of `garchfit`), you may feel that the parameters of the ARMA(1,1) model appear reasonable. However, you suspect the GARCH(1,1) results may be stuck at a local maximum. You can modify the conditional variance parameters.

```
coeff = garchset(coeff, 'K', 6.25e-6, 'GARCH', 0.85, 'ARCH', 0.05)
coeff =
          Comment: 'Mean: ARMAX(1,1,?); Variance: GARCH(1,1)'
                R: 1
                M: 1
                P: 1
                Q: 1
     Distribution: 'Gaussian'
                C: 1.0000e-004
               AR: 0.5000
               MA: 0.4000
          Regress: []
                K: 6.2500e-006
           GARCH: 0.8500
            ARCH: 0.0500
             FixC: []
            FixAR: []
            FixMA: []
      FixRegress: []
            FixK: []
         FixGARCH: []
          FixARCH: []
    Optimization: [1x1 struct]
```

You can then use this updated `coeff` specification structure as input to another round of optimization.

```
[coeff, errors, LLF, innovations, sigma, summary] = garchfit(coeff, xyz);
```

Compare the log-likelihood function values (i.e., `LLF`) to assess the various alternatives. This example illustrates the convenience of the shared specification structure.

Boundary Constraints and Statistical Inferences

The estimation process places stationarity and positivity constraints on the parameters (see Eq. (1-6) in the section "Homoskedasticity of the Unconditional Variance" on page 1-8).

Whenever `garchfit` actively imposes parameter constraints (other than user-specified equality constraints) during the estimation process, the statistical results based on the maximum likelihood parameter estimates are invalid (see Hamilton [10], page 142). This is because statistical inference relies on the log-likelihood function being approximately quadratic in the neighborhood of the maximum likelihood parameter estimates. This cannot be the case when the estimates fail to fall in the interior of the parameter space.

As an example of an actively imposed parameter constraint, fit a GARCH(1,2) model to the returns of the XYZ Corporation. This model is intentionally misspecified and estimations for such models often have difficulty converging. You can increase the likelihood of convergence by making the requirement for convergence less stringent. To do this increase the termination tolerance parameter `TolCon` from `1e-6` (the default) to `1e-5`.

```
spec = garchset('P', 1, 'Q', 2, 'TolCon', 1e-5);
[coeff, errors, LLF, innovations, sigma, summary] = garchfit(spec, xyz);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Diagnostic Information

Number of variables:5

Functions
 Objective:                            garchlllfn
 Gradient:                             finite-differencing
 Hessian:                              finite-differencing (or Quasi-Newton)
 Nonlinear constraints:                garchnlc
 Gradient of nonlinear constraints:    finite-differencing

Constraints
 Number of nonlinear inequality constraints:0
 Number of nonlinear equality constraints:  0
```

```
 Number of linear inequality constraints:   1
 Number of linear equality constraints:     0
 Number of lower bound constraints:         5
 Number of upper bound constraints:         0

Algorithm selected
   medium-scale

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 End diagnostic information

                                  max                   Directional
Iter   F-count     f(x)         constraint   Step-size   derivative  Procedure
   1       6     -5922.27  -1.684e-005           1    -3.34e+004
   2      36     -5922.27  -1.684e-005   1.19e-007          -578
   3      46     -5926.29  -1.474e-005       0.125           -60
   4      60     -5926.45  -1.558e-005     0.00781         -51.6
   5      68      -5952.6   -7.79e-006         0.5         -27.5
   6      76     -5964.39  -3.895e-006         0.5         -12.4
   7      84     -5964.42  -1.947e-006         0.5         -95.4
   8      98     -5964.43  -2.084e-006     0.00781         -27.4
   9     106     -5971.69  -1.552e-006         0.5          -7.6
  10     114     -5974.09  -7.762e-007         0.5         -97.8
  11     129     -5974.17  -9.254e-007     0.00391        -0.556
  12     136     -5974.59   4.337e-019           1       -0.0767
  13     145      -5974.6   5.421e-019        0.25       -0.0075
  14     152      -5974.6   1.084e-018           1      -0.00322
  15     159      -5974.6   2.168e-018           1      -0.00152
  16     166      -5974.6   4.337e-018           1      -0.00084
  17     173      -5974.6   8.674e-018           1     -0.000282
  18     183      -5974.6   9.758e-018       0.125     -6.16e-005
  19     191      -5974.6   1.464e-017         0.5      -0.000145   Hessian modified twice
  20     205      -5974.6   1.475e-017     0.00781     -1.94e-006
Optimization terminated successfully:
 Search direction less than 2*options.TolX and
  maximum constraint violation is less than options.TolCon
Active Constraints:
     5
Warning:Boundary Constraints Active; Standard Errors may be Inaccurate.
```

The warning message explicitly states that `garchfit` has imposed constraints. If you choose to suppress the estimations details (i.e., set the specification structure field `Display` to **`off`**), the same information is available from the `constraints` field of the `summary` output structure.

```
summary
summary =
        warning:'No Warnings'
       converge:'Function Converged to a Solution'
      covMatrix:[5x5 double]
     iterations:20
  functionCalls:208
    constraints:'Boundary Constraints Active; Errors may be Inaccurate'
```

Examine the estimation results to see exactly what happened.

```
garchdisp(coeff, errors)

 Number of Parameters Estimated: 5

                              Standard          T
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
         C    0.00048993     0.00025674       1.9083
         K    8.1018e-007    2.9827e-007      2.7163
  GARCH(1)    0.96327        0.0062937      153.0524
   ARCH(1)    0.031503       0.016075         1.9597
   ARCH(2)    0              0.018615         0.0000
```

The `0` value of `ARCH(2)` reveals that `garchfit` has enforced the variance positivity constraint of the second ARCH parameter. It indicates that the estimated GARCH(1,2) model is in fact a GARCH(1,1) model, and further emphasizes that the default model is well suited for the returns of the XYZ Corporation.

Furthermore, since a parameter constraint has been actively imposed during the estimation process, the statistical results based on the maximum likelihood parameter estimates are invalid. These statistical results include the standard errors shown in column two, as well as any likelihood ratio tests based on the `lratiotest` function.

Data Size and Quality

The size and quality of your return series affect the validity of your results. Because of this, The MathWorks recommends that you carefully examine your data prior to estimation. In particular, you should consider altering any missing or anomalous data points. For example, you can fill in missing data

points, and remove or smooth anomalous ones. (See the section “GARCH Limitations” on page 1-3.)

In addition, GARCH volatility modeling typically requires at least a few hundred observations. Assuming daily data, one year's worth of data requires about 250 data points.

2

Function Reference

This section gives you access to the GARCH function reference pages:

- By category
- Alphabetically

Functions by Category

This section lists the GARCH Toolbox functions according to their purpose.

Univariate GARCH Modeling

Function	Purpose
`garchfit`	Univariate GARCH process parameter estimation.
`garchpred`	Univariate GARCH process forecasting.
`garchsim`	Univariate GARCH process simulation.

Univariate GARCH Innovations Inference

Function	Purpose
`garchinfer`	Inverse filter to infer GARCH innovations and conditional standard deviations from an observed return series.

Log-Likelihood Objective Functions

Function	Purpose
`garchllfn`	Univariate GARCH process objective function (Gaussian innovations).

Statistics and Tests

Function	Purpose
`aicbic`	Akaike and Bayesian information criteria for model order selection.
`archtest`	Engle's hypothesis test for the presence of ARCH/GARCH effects.
`autocorr`	Plot or return computed sample auto-correlation function.
`crosscorr`	Plot or return computed sample cross-correlation function.
`lbqtest`	Ljung-Box Q-statistic lack-of-fit hypothesis test.
`lratiotest`	Likelihood ratio hypothesis test.
`parcorr`	Plot or return computed sample partial auto-correlation function.

GARCH Specification Structure Interface Functions

Function	Purpose
`garchget`	Retrieve a GARCH specification structure parameter.
`garchset`	Create or modify a GARCH specification structure.

Helpers and Utilities

Function	Purpose
`garchar`	Convert finite-order ARMA models to infinite-order AR models.
`garchcount`	Count GARCH estimation coefficients.
`garchdisp`	Display GARCH process estimation results.
`garchma`	Convert finite-order ARMA models to infinite-order MA models.
`lagmatrix`	Create a lagged time series matrix.
`price2ret`	Convert price series to a return series.
`ret2price`	Convert return series to a price series.

Graphics

Function	Purpose
`garchplot`	Plot matched univariate innovations, volatility, and return series.

Purpose

Akaike (AIC) and Bayesian (BIC) information criteria for model order selection

Syntax

```
AIC = aicbic(LogLikelihood, NumParams)
[AIC, BIC] = aicbic(LogLikelihood, NumParams, NumObs)
```

Arguments

`LogLikelihood`	Vector of optimized log-likelihood objective function (LLF) values associated with parameter estimates of the models to be tested. `aicbic` assumes you obtained the LLF values from the estimation function `garchfit`, or the inference function `garchinfer`.
`NumParams`	Number of estimated parameters associated with each LLF value in `LogLikelihood`. `NumParams` can be a scalar applied to all values in `LogLikelihood`, or a vector the same length as `LogLikelihood`. All elements of `NumParams` must be positive integers. Use `garchcount` to compute `NumParams` values.
`NumObs`	Sample size of the observed return series you associate with each value of `LogLikelihood`. `NumObs` can be a scalar applied to all values in `LogLikelihood`, or a vector the same length as `LogLikelihood`. It is required to compute BIC. All elements of `NumObs` must be positive integers.

Description

`aicbic` computes the Akaike and Bayesian information criteria, using optimized log-likelihood objective function (LLF) values as input. You can obtain the LLF values by fitting models of the conditional mean and variance to a univariate return series.

`AIC = aicbic(LogLikelihood, NumParams)` computes only the Akaike (AIC) information criteria.

`[AIC, BIC] = aicbic(LogLikelihood, NumParams, NumObs)` computes both the Akaike (AIC) and Bayesian (BIC) information criteria.

Since information criteria penalize models with additional parameters, parsimony is the basis of the AIC and BIC model order selection criteria.

`AIC` Vector of AIC statistics associated witin this “Arguments” sectionh each `LogLikelihood` objective function value. The AIC statistic is defined as:

$$AIC = (-2 \times LogLikelihood) + (2 \times NumParams)$$

`BIC` Vector of BIC statistics associated with each `LogLikelihood` objective function value. The BIC statistic is defined as:

$$BIC = (-2 \times LogLikelihood) + (NumParams \times Log(NumObs))$$

See Also

`garchdisp`, `garchfit`, `garchinfer`

Reference

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Purpose

Engle's hypothesis test for the presence of ARCH/GARCH effects

Syntax

```
[H, pValue, ARCHstat, CriticalValue] = archtest(Residuals, Lags,
  Alpha)
```

Arguments

`Residuals` Time series vector of sample residuals obtained from a curve fit, which `archtest` examines for the presence of ARCH effects. The last element contains the most recent observation.

`Lags` (optional) Vector of positive integers indicating the lags of the squared sample residuals included in the ARCH test statistic. If specified, each lag should be significantly less than the length of `Residuals`. If `Lags = []` or is not specified, the default is `1` lag (i.e., first order ARCH).

`Alpha` (optional) Significance level(s) of the hypothesis test. `Alpha` can be a scalar applied to all lags in `Lags`, or a vector of significance levels the same length as `Lags`. If `Alpha = []` or is not specified, the default is `0.05`. For all elements, α, of `Alpha`, $0 < \alpha < 1$.

Description

`[H, pValue, ARCHstat, CriticalValue] = archtest(Residuals, Lags, Alpha)` tests the null hypothesis that a time series of sample residuals consists of independent identically distributed (i.i.d.) Gaussian disturbances, i.e., no ARCH effects exist.

Given sample residuals obtained from a curve fit (e.g., a regression model), `archtest` tests for the presence of Mth order ARCH effects by regressing the squared residuals on a constant and the lagged values of the previous M squared residuals. Under the null hypothesis, the asymptotic test statistic, $T(R^2)$, where T is the number of squared residuals included in the regression and R^2 is the sample multiple correlation coefficient, is asymptotically Chi-Square distributed with M degrees of freedom. When testing for ARCH effects, a GARCH(P,Q) process is locally equivalent to an ARCH(P+Q) process.

`H`	Boolean decision vector. `0` indicates acceptance of the null hypothesis that no ARCH effects exist, i.e., there is homoskedasticity at the corresponding element of `Lags`. `1` indicates rejection of the null hypothesis. The length of `H` is the same as the length of `Lags`.
`pValue`	Vector of P-values (significance levels) at which `archtest` rejects the null hypothesis of no ARCH effects at each lag in `Lags`.
`ARCHstat`	Vector of ARCH test statistics for each lag in `Lags`.
`CriticalValue`	Vector of critical values of the Chi-Square distribution for comparison with the corresponding element of `ARCHstat`.

Example

Create a vector of 100 (synthetic) residuals, then test for the 1st, 2nd, and 4th order ARCH effects at the 10 percent significance level.

```
randn('state',0)               % Start from a known state.
residuals     = randn(100,1);  % 100 Gaussian deviates ~ N(0,1)
[H,P,Stat,CV] = archtest(residuals, [1 2 4]', 0.10);
[H,P,Stat,CV]

ans =

         0    0.3925    0.7312    2.7055
         0    0.5061    1.3621    4.6052
         0    0.7895    1.7065    7.7794
```

See Also

`lbqtest`

References

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Engle, Robert, "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, Vol. 50, pp. 987-1007, 1982.

Gourieroux, C., *ARCH Models and Financial Applications*, Springer-Verlag, 1997.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

autocorr

Purpose

Plot or return computed sample auto-correlation function

Syntax

```
autocorr(Series, nLags, M, nSTDs)
[ACF, Lags, Bounds] = autocorr(Series, nLags, M, nSTDs)
```

Arguments

`Series` Vector of observations of a univariate time series for which `autocorr` computes or plots the sample auto-correlation function (ACF). The last element of `Series` contains the most recent observation of the stochastic sequence.

`nLags` (optional) Positive, scalar integer indicating the number of lags of the ACF to compute. If `nLags = []` or is not specified, the default is to compute the ACF at lags 0, 1, 2, ..., *T*, where *T* = `min([20, length(Series)-1])`.

`M` (optional) Nonnegative integer scalar indicating the number of lags beyond which the theoretical ACF is effectively `0`. `autocorr` assumes the underlying `Series` is an MA(M) process, and uses Bartlett's approximation to compute the large-lag standard error for lags > `M`. If `M = []` or is not specified, the default is `0`, and `autocorr` assumes that `Series` is Gaussian white noise. If `Series` is a Gaussian white noise process of length *N*, the standard error is approximately $1/\sqrt{N}$. `M` must be < `nLags`.

`nSTDs` (optional) Positive scalar indicating the number of standard deviations of the sample ACF estimation error to compute. `autocorr` assumes the theoretical ACF of `Series` is `0` beyond lag `M`. When `M = 0` and `Series` is a Gaussian white noise process of length *N*, specifying `nSTDs` results in confidence bounds at $\pm(\text{nSTDs}/\sqrt{N})$. If `nSTDs = []` or is not specified, the default is `2` (i.e., approximate 95 percent confidence interval).

Description

`autocorr(Series, nLags, M, nSTDs)` computes and plots the sample ACF of a univariate, stochastic time series with confidence bounds. To plot the ACF sequence without the confidence bounds, set `nSTDs = 0`.

`[ACF, Lags, Bounds] = autocorr(Series, nLags, M, nSTDs)` computes and returns the ACF sequence.

`ACF` Sample auto-correlation function of `Series`. `ACF` is a vector of length `nLags`+1 corresponding to lags 0, 1, 2, ..., `nLags`. The first element of `ACF` is unity, that is, `ACF(1) = 1 =` lag `0` correlation.

`Lags` Vector of lags corresponding to `ACF(0,1,2,...,nLags)`. Since an ACF is symmetric about `0` lag, `autocorr` ignores negative lags.

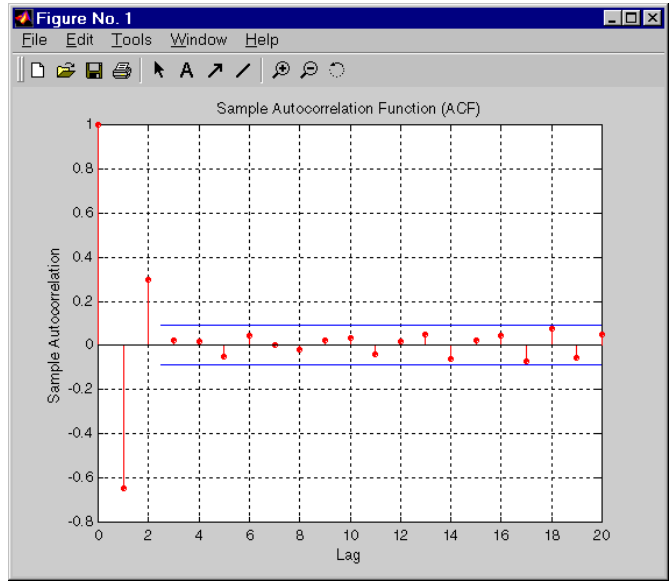
`Bounds` Two element vector indicating the approximate upper and lower confidence bounds, assuming that `Series` is an MA(M) process. Values of `ACF` beyond lag `M` that are effectively `0` lie within these bounds. Note that `autocorr` computes `Bounds` only for lags > `M`.

Example

Create an MA(2) process from a sequence of 1000 Gaussian deviates, and assess whether the ACF is effectively zero for lags > 2.

```
randn('state', 0)             % Start from a known state.
x = randn(1000, 1);           % 1000 Gaussian deviates ~ N(0,1).
y = filter([1 -1 1], 1, x);   % Create an MA(2) process.
[ACF, Lags, Bounds] = autocorr(y, [], 2);  % Compute the ACF
                                           % with 95 percent
                                           % confidence.

[Lags, ACF]

ans =

         0    1.0000
    1.0000   -0.6487
    2.0000    0.3001
    3.0000    0.0229
    4.0000    0.0196
    5.0000   -0.0489
    6.0000    0.0452
    7.0000    0.0012
    8.0000   -0.0214
    9.0000    0.0235
   10.0000    0.0340
   11.0000   -0.0392
   12.0000    0.0188
   13.0000    0.0504
```

```
   14.0000   -0.0600
   15.0000    0.0251
   16.0000    0.0441
   17.0000   -0.0732
   18.0000    0.0755
   19.0000   -0.0571
   20.0000    0.0485

Bounds =

    0.0899
   -0.0899

autocorr(y, [], 2)      % Use the same example, but plot the ACF
                        % sequence with confidence bounds.
```



See Also

`crosscorr`, `parcorr`
`filter` (in the online MATLAB Function Reference)

Reference

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

crosscorr

Purpose

Plot or return computed sample cross-correlation function

Syntax

```
crosscorr(Series1, Series2, nLags, nSTDs)
[XCF, Lags, Bounds] = crosscorr(Series1, Series2, nLags, nSTDs)
```

Arguments

`Series1` Vector of observations of the first univariate time series for which `crosscorr` computes or plots the sample cross-correlation function (XCF). The last element of `Series1` contains the most recent observation.

`Series2` Vector of observations of the second univariate time series for which `crosscorr` computes or plots the sample XCF. The last element of `Series2` contains the most recent observation.

`nLags` (optional) Positive, scalar integer indicating the number of lags of the XCF to compute. If `nLags = []` or is not specified, `crosscorr` computes the XCF at lags 0, 1, 2, ...,*T*, where T = `min([20, min([length(Series1), length(Series2)])-1])`.

`nSTDs` (optional) Positive scalar indicating the number of standard deviations of the sample XCF estimation error to compute, if `Series1` and `Series2` are uncorrelated. If `nSTDs = []` or is not specified, the default is `2` (i.e., approximate 95 percent confidence interval).

Description

`crosscorr(Series1, Series2, nLags, nSTDs)` computes and plots the sample cross-correlation function (XCF) between two univariate, stochastic time series. To plot the XCF sequence without the confidence bounds, set `nSTDs = 0`.

`[XCF, Lags, Bounds] = crosscorr(Series1, Series2, nLags, nSTDs)` computes and returns the XCF sequence.

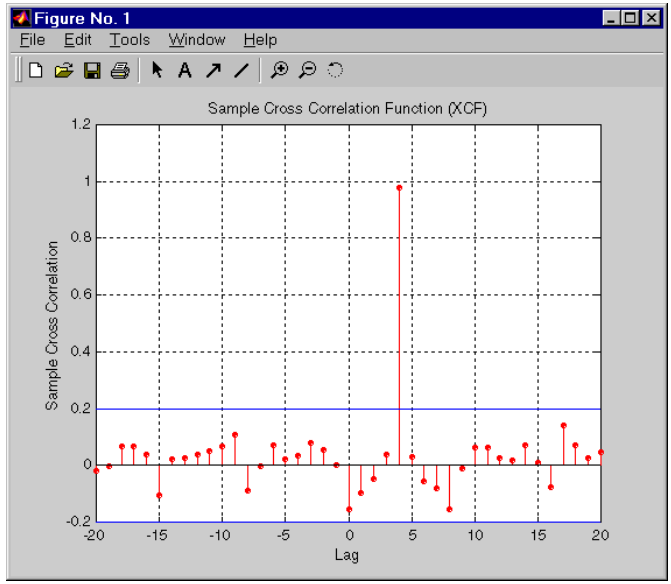
XCF	Sample cross-correlation function between Series1 and Series2. XCF is a vector of length 2(nLags)+1 corresponding to lags 0, 1, 2, ...,nLags. The center element of XCF contains the 0th lag cross correlation. XCF is a row (column) vector if Series1 is a row (column) vector.
Lags	Vector of lags corresponding to XCF(-nLags, ..., +nLags).
Bounds	Two-element vector indicating the approximate upper and lower confidence bounds assuming Series1 and Series2 are completely uncorrelated.

Example

Create a random sequence of 100 Gaussian deviates, and a delayed version lagged by four samples. Compute the XCF, and then plot it to see the XCF peak at the fourth lag.

```
randn('state',100)              % Start from a known state.
x           = randn(100,1);     % 100 Gaussian deviates, N(0,1).
y           = lagmatrix(x, 4);  % Delay it by 4 samples.
y(isnan(y)) = 0;                % Replace NaNs with zeros.
[XCF, Lags, Bounds] = crosscorr(x,y);  % Compute the XCF with
                                       % 95 percent confidence.
[Lags, XCF]

ans =

  -20.0000   -0.0210
  -19.0000   -0.0041
  -18.0000    0.0661
  -17.0000    0.0668
  -16.0000    0.0380
  -15.0000   -0.1060
  -14.0000    0.0235
  -13.0000    0.0240
  -12.0000    0.0366
  -11.0000    0.0505
  -10.0000    0.0661
   -9.0000    0.1072
   -8.0000   -0.0893
```

```
   -7.0000   -0.0018
   -6.0000    0.0730
   -5.0000    0.0204
   -4.0000    0.0352
   -3.0000    0.0792
   -2.0000    0.0550
   -1.0000    0.0004
         0   -0.1556
    1.0000   -0.0959
    2.0000   -0.0479
    3.0000    0.0361
    4.0000    0.9802
    5.0000    0.0304
    6.0000   -0.0566
    7.0000   -0.0793
    8.0000   -0.1557
    9.0000   -0.0128
   10.0000    0.0623
   11.0000    0.0625
   12.0000    0.0268
   13.0000    0.0158
   14.0000    0.0709
   15.0000    0.0102
   16.0000   -0.0769
   17.0000    0.1410
   18.0000    0.0714
   19.0000    0.0272
   20.0000    0.0473

Bounds =

    0.2000
   -0.2000

crosscorr(x,y)          % Use the same example, but plot the XCF
                        % sequence. Note the peak at the 4th lag.
```



See Also

`autocorr`, `parcorr`
`filter` (in the online MATLAB Function Reference)

garchar

Purpose

Convert finite-order ARMA models to infinite-order auto-regressive (AR) models

Syntax

```
InfiniteAR  = garchar(AR, MA, NumLags)
```

Arguments

`AR` — *R*-element vector of auto-regressive coefficients associated with the lagged observations of a univariate return series modeled as a finite order, stationary, invertible ARMA(R,M) model.

`MA` — *M*-element vector of moving-average coefficients associated with the lagged innovations of a finite-order, stationary, invertible univariate ARMA(R,M) model.

`NumLags` — (optional) Number of lagged AR coefficients that `garchar` includes in the approximation of the infinite-order AR representation. `NumLags` is an integer scalar and determines the length of the infinite-order AR output vector. If `NumLags = []` or is not specified, the default is `10`.

Description

`InfiniteAR = garchar(AR, MA, NumLags)` computes the coefficients of an infinite-order AR model, using the coefficients of the equivalent univariate, stationary, invertible, finite-order ARMA(R,M) model as input. `garchar` truncates the infinite-order AR coefficients to accommodate a user-specified number of lagged AR coefficients.

`InfiniteAR` — Vector of coefficients of the infinite-order AR representation associated with the finite-order ARMA model specified by the `AR` and `MA` input vectors. `InfiniteAR` is a vector of length `NumLags`. The *j*th element of `InfiniteAR` is the coefficient of the *j*th lag of the input series in an infinite-order AR representation. Note that Box, Jenkins, and Reinsel refer to the infinite-order AR coefficients as "π weights."

In the following ARMA(R,M) model, $\{y_t\}$ is the return series of interest and $\{\varepsilon_t\}$ the innovations noise process.

$$y_t = \sum_{i=1}^{R} AR_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{M} MA_j \varepsilon_{t-j}$$

If you write this model equation as

$$y_t = AR_1 y_{t-1} + \ldots + AR_R y_{t-R} + \varepsilon_t + MA_1 \varepsilon_{t-1} + \ldots + MA_M \varepsilon_{t-M}$$

you can specify the `garchar` input coefficient vectors, `AR` and `MA`, exactly as you read them from the model. In general, the *j*th elements of `AR` and `MA` are the coefficients of the *j*th lag of the return series and innovations processes y_{t-j} and ε_{t-j}, respectively. `garchar` assumes that the current-time-index coefficients of y_t and ε_t are `1` and are *not* part of `AR` and `MA`.

In theory, you can use the π weights returned in `InfiniteAR`, to approximate y_t as a pure AR process.

$$y_t = \sum_{i=1}^{\infty} \pi_i y_{t-i} + \varepsilon_t$$

Consistently, the *j*th element of the truncated infinite-order auto-regressive output vector, π_j or `InfiniteAR(j)`, is the coefficient of the *j*th lag of the observed return series, y_{t-j}, in this equation. See Box, Jenkins, and Reinsel [7], Section 4.2.3, pages 106-109.

Given the above discussion, the `AR` and `MA` vectors differ from the corresponding AR and MA polynomials formally presented in time series references such as Box, Jenkins, and Reinsel. The conversion from GARCH Toolbox vectors to the corresponding GARCH Toolbox polynomials is:

```
AR polynomial tested for stationarity = [1 ; -AR]
MA polynomial tested for invertibility = [1 ; MA]
```

Example

For the following ARMA(2,2) model, use `garchar` to obtain the first 20 weights of the infinite order AR approximation.

$$y_t = 0.5 y_{t-1} - 0.8 y_{t-2} + \varepsilon_t - 0.6 \varepsilon_{t-1} + 0.08 \varepsilon_{t-2}$$

From this model,

```
AR = [0.5 -0.8]
```

```
MA = [-0.6 0.08]
```

Since the current-time-index coefficients of y_t and ε_t are defined to be 1, the example omits them from AR and MA. This saves time and effort when you specify parameters using the garchset and garchget interfaces.

```
PI = garchar([0.5 -0.8], [-0.6 0.08], 20);

PI'

ans =

   -0.1000
   -0.7800
   -0.4600
   -0.2136
   -0.0914
   -0.0377
   -0.0153
   -0.0062
   -0.0025
   -0.0010
   -0.0004
   -0.0002
   -0.0001
   -0.0000
   -0.0000
   -0.0000
   -0.0000
   -0.0000
   -0.0000
   -0.0000
```

See Also

garchfit, garchma, garchpred

Reference

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Purpose

Count GARCH estimation coefficients

Syntax

```
NumParams = garchcount(Coeff)
```

Arguments

`Coeff` GARCH specification structure containing the estimated coefficients and equality constraints. `Coeff` is an output of the estimation function `garchfit`.

Description

`NumParams = garchcount(Coeff)` counts and returns the number of estimated coefficients from a GARCH specification structure containing coefficient estimates and equality constraint information. `garchcount` is a helper utility designed to support the GARCH Toolbox model selection function `aicbic`.

`NumParams` Number of estimated parameters (i.e., coefficients) included in the conditional mean and variance specifications, less any parameters held constant, as equality constraints, during the estimation. The `aicbic` function needs `NumParams` to calculate the Akaike (AIC) and Bayesian (BIC) statistics.

See Also

`aicbic`, `garchdisp`, `garchfit`

garchdisp

Purpose

Display GARCH process estimation results

Syntax

```
garchdisp(Coeff, Errors)
```

Arguments

`Coeff`	GARCH specification structure containing estimated coefficients and equality constraint information. `Coeff` is an output of the estimation function `garchfit`.
`Errors`	Structure containing the estimation errors (i.e., the standard errors) of the coefficients in `Coeff`. `Errors` is also an output of the estimation function `garchfit`.

Description

`garchdisp(Coeff, Errors)` displays coefficient estimates, standard errors, and T-statistics from a GARCH specification structure that was output by the estimation function `garchfit`.

This function displays matched GARCH Toolbox estimation results, and returns no output arguments. The tabular display includes parameter estimates, standard errors, and T-statistics for each parameter in the conditional mean and variance models. Parameters held fixed during the estimation process have the word `'Fixed'` printed in the standard error and T-statistic columns, indicating that the parameter was set as an equality constraint.

See Also

`garchcount`, `garchfit`

Purpose

Univariate GARCH process parameter estimation

Syntax

```
[Coeff, Errors, LLF, Innovations, Sigma, Summary] = garchfit(Series)
[Coeff, Errors, LLF, Innovations, Sigma, Summary] = garchfit(Spec,
  Series)
[Coeff, Errors, LLF, Innovations, Sigma, Summary] = garchfit(Spec,
  Series, X)
garchfit(...)
```

Arguments

`Series` Vector of observations of the underlying univariate return series for which `garchfit` estimates the parameters of the conditional mean and variance models. The last element of `Series` holds the most recent observation.

`Spec` (optional) GARCH specification structure that contains the conditional mean and variance models, and optimization parameters. You create the fields in this structure by calling the function `garchset`, or you can use the `Coeff` output structure from a previous call to `garchfit`.

`X` (optional) Time series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (e.g., the return series of an equity index), and represents the past history of the explanatory data. Each column of `X` is an individual time series used as an explanatory variable in the regression component of the conditional mean. In each column, the first row contains the oldest observation and the last row the most recent.

The number of valid (non-`NaN`) most recent observations in each column of `X` must equal or exceed the number of valid most recent observations in `Series`. If the number of valid observations in a column of `X` exceeds that of `Series`, `garchfit` uses only the most recent observations of `X`. If `X = []` or is not specified, the conditional mean has no regression component.

Description

`garchfit` estimates the parameters of a conditional mean specification of ARMAX form and a conditional variance specification of GARCH form. If the `Display` flag (see the function `garchset`) in the specification structure is set to **on** (the default), it also displays diagnostic and iterative optimization information in the MATLAB command window (see the function `fmincon` in the Optimization Toolbox).

`[Coeff, Errors, LLF, Innovations, Sigma, Summary] = garchfit(Series)` models an observed univariate return series as a constant, *C*, plus GARCH(1,1) conditionally Gaussian innovations. For models beyond this simplistic (yet common) model, you must provide model parameters in the specification structure, `Spec`. The *C* + GARCH(1,1) model is the default model of the GARCH Toolbox.

`[Coeff, Errors, LLF, Innovations, Sigma, Summary] = garchfit(Spec, Series)` infers the innovations from the return series and fits the model specification, contained in `Spec`, to the return series by maximum likelihood.

`[Coeff, Errors, LLF, Innovations, Sigma, Summary] = garchfit(Spec, Series, X)` provides a regression component for the conditional mean.

`garchfit(...)` (with input arguments as shown above but with no output arguments) prints the final parameter estimates and standard errors to the MATLAB command window. It also produces a tiered plot of the original return series, the inferred innovations (i.e., residuals), and the corresponding conditional standard deviations.

`Coeff`	GARCH specification structure containing the estimated coefficients. `Coeff` is of the same form as the `Spec` input structure. This allows other GARCH Toolbox functions, such as `garchset`, `garchget`, `garchsim`, `garchinfer`, and `garchpred`, to accept either `Spec` or `Coeff` seamlessly.
`Errors`	Structure containing the estimation errors (i.e., the standard errors) of the coefficients. The fields of `Errors` correspond to the coefficient fields (`C`, `AR`, `MA`, `Regress`, `K`, `GARCH`, `ARCH`) found in `Coeff` or `Spec`.

`LLF` Optimized log-likelihood objective function value associated with the parameter estimates found in `Coeff`. `garchfit` performs the optimization using the `fmincon` function of the Optimization Toolbox.

`Innovations` Innovations vector inferred from `Series`. The size of `Innovations` is the same as the size of `Series`.

`Sigma` Conditional standard deviation vector corresponding to `Innovations`. The size of `Sigma` is the same as the size of `Series`.

`Summary` Structure of summary information about the optimization process. The fields and their possible values are

- `warning` One of the following strings:
  ```
  'No Warnings'
  'ARMA Model Is Not Stationary/Invertible'
  ```
- `converge` One of the following strings:
  ```
  'Function Converged to a Solution'
  'Function Did NOT Converge'
  'Maximum Function Evaluations or Iterations
   Reached'
  ```
- `covMatrix` Covariance matrix of the parameter estimates
- `iterations` Number of iterations
- `functionCalls` Number of function evaluations
- `constraints` One of the following strings:
  ```
  'No Boundary Constraints'
  'Boundary Constraints Active; Errors May Be
   Inaccurate'
  ```

Note `garchfit` calculates the error covariance matrix of the parameter estimates, `Summary.covMatrix`, and the corresponding standard errors found in the `Errors` output structure, using finite difference approximation. In particular, it calculates the standard errors using the outer-product method (see Hamilton [10], section 5.8, bottom of page 143).

See Also

`garchllfn`, `garchpred`, `garchset`, `garchsim`, `fmincon` (in the Optimization Toolbox)

References

Bollerslev, T., "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of Econometrics*, Vol. 31, pp. 307-327, 1986.

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Engle, Robert, "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, vol. 50, pp. 987-1007, 1982.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

Purpose

Retrieve a GARCH specification structure parameter

Syntax

```
ParameterValue = garchget(Spec, 'ParameterName')
```

Arguments

`Spec` GARCH specification structure containing the orders, and coefficients, as well as the optimization constraints of the conditional mean and variance specifications of a GARCH model. You can create a GARCH specification structure as the output (`Spec`) of the companion function `garchset`, or the output (`Coeff`) of the estimation function `garchfit`.

`ParameterName` String indicating the name of the parameter whose value `garchget` extracts from `Spec`. You can specify only sufficient leading characters to uniquely identify the parameter. See `garchset` for a list of valid parameter names. `ParameterName` is case insensitive.

Description

`ParameterValue = garchget(Spec, 'ParameterName')` provides the preferred user-interface for retrieveing a model parameter from a GARCH specification structure.

`ParameterValue` Value of the named parameter, `ParameterName`, extracted from the structure `Spec`. `ParameterValue = []` if the parameter has no value.

Example

```
Spec = garchset('P',1,'Q',1)  % Create a GARCH(P=1, Q=1) model.
Spec =

          Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(1,1)'
                R: 0
                M: 0
                P: 1
                Q: 1
     Distribution: 'Gaussian'
                C: []
               AR: []
```

```
                MA: []
           Regress: []
                 K: []
             GARCH: []
              ARCH: []
              FixC: []
             FixAR: []
             FixMA: []
         FixRegress: []
               FixK: []
           FixGARCH: []
            FixARCH: []
      Optimization: [1x1 struct]

  P = garchget(Spec,'P')              % Extract the order P.

  P =
       1
```

See Also

`garchfit`, `garchpred`, `garchset`, `garchsim`
`optimget`, `optimset` (in the online MATLAB Function Reference)

Purpose

Inverse filter to infer GARCH innovations and conditional standard deviations from an observed return series

Syntax

```
[Innovations, Sigma, LogLikelihood] = garchinfer(Spec, Series, X)
```

Arguments

`Spec` GARCH specification structure that contains the conditional mean and variance specifications, as well as the optimization parameters of a GARCH model. You can create `Spec` by calling the function `garchset` or the estimation function `garchfit`.

`Series` Matrix of observations of the underlying univariate return series of interest for which `garchinfer` infers the innovations and corresponding conditional standard deviations.
Each column of `Series` is an independent realization (i.e., path). The last row of `Series` holds the most recent observation of each realization.

`X` (optional) Time series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (e.g., the return series of an equity index), and represents the past history of the explanatory data. Each column of `X` is an individual time series used as an explanatory variable in the regression component of the conditional mean. In each column, the first row contains the oldest observation and the last row the most recent.

The number of valid (non-`NaN`) most recent observations in each column of `X` must equal or exceed the number of valid most recent observations in `Series`. If the number of valid observations in a column of `X` exceeds that of `Series`, `garchinfer` uses only the most recent observations of `X`. If `X = []` or is not specified, the conditional mean has no regression component.

Description

`[Innovations, Sigma, LogLikelihood] = garchinfer(Spec, Series, X)` acts as an inverse, or whitening, filter to infer the innovations and conditional standard deviations from an observed return series, using a conditional mean specification of ARMAX form and a conditional variance specification of GARCH form as input. Since `garchinfer` provides an interface to the

appropriate log-likelihood objective function, it also computes the log-likelihood value as a convenience.

`Innovations`	Innovations matrix inferred from the input `Series` matrix. The size of `Innovations` is the same as the size of `Series`, and its organization is the same as `Series`.
`Sigma`	Conditional standard deviation matrix corresponding to `Innovations`. The size of `Sigma` is the same as the size of `Series`.
`LogLikelihood`	Vector of log-likelihood objective function values for each realization of `Series`. The length of `LogLikelihood` is the same as the number of columns in `Series`.

See Also

`garchfit`, `garchllfn`, `garchpred`, `garchset`, `garchsim`
`fmincon` (in the Optimization Toolbox)

References

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

Purpose

Univariate GARCH process objective function (Gaussian innovations)

Syntax

```
[LogLikelihood, G, H, Innovations, Sigma] = garchllfn(Parameters,
  Series, R, M, P, Q, X)
```

Arguments

`Parameters`	Column vector of process parameters associated with fitting conditional mean and variance specifications to the observed return series, `Series`. The conditional mean contributes the first (`1 + R + M + Nx`) parameters, where `Nx` is the number of explanatory variables you include in the regression component of the conditional mean (the number of columns in `X`). The conditional variance contributes the remaining (`1 + P + Q`) parameters. The resultant length of `Parameters` is (`2 + R + M + Nx + P + Q`). (See the "Formatting the Input Coefficient Vector" section below.)
`Series`	Matrix of observations of the underlying univariate return series of interest for which `garchllfn` estimates the parameters of the conditional mean and variance models. `Series` can have several columns, where each column is an independent realization (i.e., path). The last row of `Series` holds the most recent observation of each realization.
`R`	Nonnegative, scalar integer representing the AR-process order.
`M`	Nonnegative, scalar integer representing the MA-process order.
`P`	Nonnegative, scalar integer representing the number of lags of the conditional variance included in the GARCH process.

`Q`	Nonnegative, scalar integer representing the number of lags of the squared innovations included in the GARCH process.
`X`	(optional) Time series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (e.g., the return series of an equity index), and represents the past history of the explanatory data. Each column of `X` is an individual time series used as an explanatory variable in the regression component of the conditional mean. In each column, the first row contains the oldest observation and the last row the most recent. `X` must have the same number of rows as `Series`.

Description

`[LogLikelihood, G, H, Innovations, Sigma] = garchllfn(Parameters, Series, R, M, P, Q, X)` computes the log-likelihood objective function value suitable for maximum likelihood estimation (MLE).

For Gaussian innovations, `garchfit` uses `garchllfn` as the objective function to be optimized by `fmincon`. When `garchinfer` calls `garchllfn`, the primary outputs of `garchllfn` are the innovations and conditional standard deviations inferred from the input data. In either case, `garchllfn` must infer an uncorrelated white noise innovation process. In this sense, `garchllfn` is an inverse, or whitening, filter.

The use of `garchllfn` is specific to `Distribution = 'Gaussian'` in the GARCH specification structure.

Note Because `garchllfn` is performance sensitive and because `fmincon` calls it iteratively as the objective function, `garchllfn` performs no argument checking. Although you can call `garchllfn` directly, it is better to call it via `garchinfer`.

`LogLikelihood`	Vector of log-likelihood objective function values evaluated at the values in `Parameters`. The length of `LogLikelihood` is the same as the number of columns in `Series`. Because the `fmincon` function (of the Optimization Toolbox), which is used to optimize `garchllfn`, is a minimization routine, `LogLikelihood` is the negative of what is formally presented in most econometrics references
`G`	Reserved for future use. `G = []`.
`H`	Reserved for future use. `H = []`.
`Innovations`	Innovations matrix inferred from the input `Series` matrix.
`Sigma`	Conditional standard deviation matrix corresponding to `Innovations`.

Formatting the Input Coefficient Vector

Format the input coefficient vector `Parameters` exactly as you would read the coefficients from the recursive difference equations when solving for the current values of the y_t and σ_t^2 time series. Specifically, if:

- y_t = return series of interest (assumed stationary)
- ε_t = innovations of the model noise process (assumed invertible)
- σ_t^2 = conditional variance of the innovations process ε_t

then the following equations represent the general ARMAX(R,M,Nx)/ GARCH(P,Q) model.

$$y_t = C + \sum_{i=1}^{R} AR_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{M} MA_j \varepsilon_{t-j} + \sum_{k=1}^{Nx} \beta_k X(t,k)$$

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2$$

You can also write these equations as

$$y_t = C + AR_1 y_{t-1} + \ldots + AR_R y_{t-R} + \varepsilon_t + MA_1 \varepsilon_{t-1} + \ldots + MA_M \varepsilon_{t-M} + \beta_1 X(t,1) + \ldots + \beta_{Nx} X(t, Nx)$$

$$\sigma_t^2 = K + G_1 \sigma_{t-1}^2 + \ldots + G_P \sigma_{t-P}^2 + A_1 \varepsilon_{t-1}^2 + \ldots + A_Q \varepsilon_{t-Q}^2$$

Using this form, the following equations represent the conditional mean and variance of a specific ARMAX(R=2, M=2, Nx=1) / GARCH(P=2, Q=2) composite model.

$$y_t = 1.3 + 0.5 y_{t-1} - 0.8 y_{t-2} + \varepsilon_t - 0.6 \varepsilon_{t-1} + 0.08 \varepsilon_{t-2} + 1.2 X(t)$$

$$\sigma_t^2 = 0.5 + 0.2 \sigma_{t-1}^2 + 0.1 \sigma_{t-2}^2 + 0.3 \varepsilon_{t-1}^2 + 0.2 \varepsilon_{t-2}^2$$

In the MATLAB notation, and using specification structure parameter names, the coefficient vector, `Parameters`, that represents this model is

```
Parameters =
 [ C     AR(1:R)   MA(1:M)     B(1:Nx)  K     GARCH(1:P)  ARCH(1:Q) ]' =
 [ 1.3   0.5 -0.8  -0.6 0.08   1.2      0.5   0.2 0.1     0.3 0.2   ]'
```

Note that the coefficient of ε_t in the conditional mean equation is `1`. Since `garchfit` does not estimate the coefficient of ε_t, the coefficient vector does *not* include it.

Inferring the Innovations

`garchllfn` uses the following conditional mean specification of ARMAX form to infer the innovations, and then fits the conditional variance of the innovations to a GARCH model. It assumes Gaussian innovations.

$$\varepsilon_t = -C + y_t - \sum_{i=1}^{R} AR_i y_{t-i} - \sum_{j=1}^{M} MA_j \varepsilon_{t-j} - \sum_{k=1}^{Nx} \beta_k X(t,k)$$

You can derive this equation from the general conditional mean equation given above for y_t, by solving it for ε_t. Its coefficient vector, which `garchllfn` uses to infer the innovations, is the negation of `Parameters` with the insertion of the y_t coefficient.

See "Maximum Likelihood Estimation" in the "Tutorial" chapter for more information.

See Also

`garchfit`, `garchinfer`, `garchpred`, `garchsim`

References

Bollerslev, T. (1986), "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of Econometrics*, Vol. 31, pp. 307-327.

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Engle, Robert (1982), "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, Vol. 50, pp. 987-1007.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

garchma

Purpose

Convert finite-order ARMA models to infinite-order moving average (MA) models

Syntax

```
InfiniteMA = garchma(AR, MA, NumLags)
```

Arguments

`AR` — *R*-element vector of auto-regressive coefficients associated with the lagged observations of a univariate return series modeled as a finite order, stationary, invertible ARMA(R,M) model.

`MA` — *M*-element vector of moving-average coefficients associated with the lagged innovations of a finite-order, stationary, invertible univariate ARMA(R,M) model.

`NumLags` — (optional) Number of lagged MA coefficients that `garchma` includes in the approximation of the infinite-order MA representation. `NumLags` is an integer scalar and determines the length of the infinite-order MA output vector. If `NumLags = []` or is not specified, the default is `10`.

Description

`InfiniteMA = garchma(AR, MA, NumLags)` computes the coefficients of an infinite-order MA model, using the coefficients of the equivalent univariate, stationary, invertible finite-order ARMA(R,M) model as input. `garchma` truncates the infinite-order MA coefficients to accommodate the number of lagged MA coefficients you specify in `NumLags`.

This function is particularly useful for calculating the standard errors of minimum mean square error forecasts of univariate ARMA models.

`InfiniteMA` — Vector of coefficients of the infinite-order MA representation associated with the finite-order ARMA model specified by `AR` and `MA`. `InfiniteMA` is a vector of length `NumLags`. The *j*th element of `InfiniteMA` is the coefficient of the *j*th lag of the innovations noise sequence in an infinite-order MA representation. Note that Box, Jenkins, and Reinsel refer to the infinite-order MA coefficients as the "ψ weights."

In the following ARMA(R,M) model, $\{y_t\}$ is the return series of interest and $\{\varepsilon_t\}$ the innovations noise process.

$$y_t = \sum_{i=1}^{R} AR_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{M} MA_j \varepsilon_{t-j}$$

If you write this model equation as

$$y_t = AR_1 y_{t-1} + \ldots + AR_R y_{t-R} + \varepsilon_t + MA_1 \varepsilon_{t-1} + \ldots + MA_M \varepsilon_{t-M}$$

you can specify the `garchar` input coefficient vectors, `AR` and `MA`, exactly as you read them from the model. In general, the *j*th elements of `AR` and `MA` are the coefficients of the *j*th lag of the return series and innovations processes y_{t-j} and ε_{t-j}, respectively. `garchma` assumes that the current-time-index coefficients of y_t and ε_t are `1` and are not part of `AR` and `MA`.

In theory, you can use the ψ weights returned in `InfiniteMA` to approximate y_t as a pure MA process.

$$y_t = \varepsilon_t + \sum_{i=1}^{\infty} \psi_i \varepsilon_{t-i}$$

Consistently, the *j*th element of the truncated infinite-order moving-average output vector, ψ_j or `InfiniteMA(j)`, is the coefficient of the *j*th lag of the innovations process, ε_{t-j}, in this equation. See Box, Jenkins, and Reinsel [7], Section 5.2.2, pages 139-141.

Given the above discussion, the `AR` and `MA` vectors differ from the corresponding AR and MA polynomials formally presented in time series references such as Box, Jenkins, and Reinsel. The conversion from GARCH Toolbox vectors to the corresponding GARCH Toolbox polynomials is as follows:

- AR polynomial tested for stationarity = `[1 ; -AR]`
- MA polynomial tested for invertibility = `[1 ; MA]`

Example

Suppose you want a forecast horizon of 10 periods for the following ARMA(2,2) model.

$$y_t = 0.5y_{t-1} - 0.8y_{t-2} + \varepsilon_t - 0.6\varepsilon_{t-1} + 0.08\varepsilon_{t-2}$$

To obtain probability limits for these forecasts, use `garchma` to compute the first 9 (i.e., `10 - 1`) weights of the infinite order MA approximation.

From the model, `AR = [0.5 -0.8]` and `MA = [-0.6 0.08]`.

Since the current-time-index coefficients of y_t and ε_t are `1`, the example omits them from `AR` and `MA`. This saves time and effort when you specify parameters via the `garchset` and `garchget` user interfaces.

```
PSI = garchma([0.5 -0.8], [-0.6 0.08], 9);
PSI'

ans =

   -0.1000
   -0.7700
   -0.3050
    0.4635
    0.4758
   -0.1329
   -0.4471
   -0.1172
    0.2991
```

See Also

`garchar`, `garchpred`

Reference

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Purpose

Plot matched univariate innovations, volatility, and return series

Syntax

```
garchplot(Innovations, Sigma, Series)
```

Arguments

`Innovations`	Vector or matrix of innovations. As a vector, `Innovations` represents a single realization of a univariate time series in which the first element contains the oldest observation and the last element the most recent. As a matrix, each column of `Innovations` represents a single realization of a univariate time series in which the first row contains the oldest observation of each realization and the last row the most recent. If `Innovations = []`, then `Innovations` is not displayed.
`Sigma`	Vector or matrix of conditional standard deviations. In general, `Innovations` and `Sigma` are the same size, and form a matching pair of arrays. If `Sigma = []`, then `Sigma` is not displayed.
`Series`	Vector or matrix of asset returns. In general, `Series` is the same size as `Innovations` and `Sigma`, and is organized in exactly the same manner. If `Series = []` or is not specified, then `Series` is not displayed.

Description

`garchplot` lets you visually compare matched innovations, conditional standard deviations, and returns. It provides a convenient way to compare innovations series, simulated using `garchsim` or estimated using `garchfit`, with companion conditional standard deviations, and/or returns series. You can also use `garchplot` to plot forecasts, computed using `garchpred`, of conditional standard deviations and returns.

In general, `garchplot` produces a tiered plot of matched time series. garchplot does not display an empty or missing input array, i.e., `garchplot` allocates no space in the tiered figure window to the array. `garchplot` displays valid (nonempty) `Innovations`, `Sigma`, and `Series` arrays in the top, center, and bottom plots, respectively. Since `garchplot` assigns a title and label to each plot according to its position in the argument list, you can ensure correct plot annotation by using empty matrices (`[]`) as placeholders.

You can plot several realizations of each array simultaneously because `garchplot` color codes corresponding realizations of each input array. However, the plots may become cluttered if you try to display more than a few realizations of each input at one time.

Examples

Assume `Innovations`, `Sigma`, and `Series` are not empty.

```
garchplot(Innovations)                % Plot Innovations only.
garchplot(Innovations, [], Series)    % Plot Innovations and
                                      % Series only.
garchplot([], Sigma, Series)          % Plot Sigma and Series
                                      % only.
garchplot(Innovations, Sigma, Series) % Plot all three vectors.
garchplot(Innovations, Sigma, [])     % Plot Innovations and
                                      % Sigma only.
garchplot(Innovations, Sigma)         % Plot Innovations and
                                      % Sigma only.
```

See Also

`garchfit`, `garchpred`, `garchsim`

Purpose

Univariate GARCH process forecasting

Syntax

```
SigmaForecast = garchpred(Spec, Series, NumPeriods, X)
[SigmaForecast, MeanForecast] = garchpred(Spec, Series, NumPeriods,
  X, XF)
[SigmaForecast, MeanForecast, SigmaTotal, MeanRMSE] =
  garchpred(Spec, Series, NumPeriods)
```

Arguments

`Spec`	GARCH specification structure for the conditional mean and variance models. You can create `Spec` by calling the function `garchset` or the estimation function `garchfit`.
`Series`	Matrix of observations of the underlying univariate return series of interest for which `garchpred` generates forecasts. Each column of `Series` is an independent realization (i.e., path). The last row of `Series` holds the most recent observation of each realization. `garchpred` assumes that `Series` is a stationary stochastic process. It also assumes that the ARMA component of the conditional mean model (if any) is stationary and invertible.
`NumPeriods`	(optional) Positive, scalar integer representing the forecast horizon of interest. The value you specify should be compatible with the sampling frequency of `Series`. If `NumPeriods = []` or is not specified, the default is `1`.

`X` (optional) Time series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (e.g., the return series of an equity index), and represents the past history of the explanatory data. Each column of `X` is an individual time series used as an explanatory variable in the regression component of the conditional mean. In each column, the first row contains the oldest observation and the last row the most recent.

The number of valid (non-`NaN`) most recent observations in each column of `X` must equal or exceed the number of valid most recent observations in `Series`. If the number of valid observations in a column of `X` exceeds that of `Series`, `garchpred` uses only the most recent observations of `X`.

If `X = []` or is not specified, the conditional mean (`MeanForecast`) has no regression component.

`XF` (optional) Time series matrix of forecasted explanatory data. `XF` represents the evolution into the future of the same explanatory data found in `X`. Because of this, `XF` and `X` must have the same number of columns. In each column of `XF`, the first row contains the one-period-ahead forecast, the second row the two-period-ahead forecast, and so on.

The number of rows (forecasts) in each column (time series) of `XF` must equal or exceed the forecast horizon `NumPeriods`. When the number of forecasts in `XF` exceeds `NumPeriods`, `garchpred` uses only the first `NumPeriods` forecasts.

If `XF = []` or is not specified, the conditional mean (`MeanForecast`) has no regression component.

Description

`garchpred` forecasts the conditional mean and standard deviation of the univariate return series `NumPeriods` into the future, using specifications for the conditional mean and variance of an observed univariate return series as input. The conditional mean and variance can be of general ARMAX and GARCH form, respectively.

`SigmaForecast = garchpred(Spec, Series, NumPeriods, X)` forecasts only the standard deviation of the univariate return series, `Series`. The regression matrix `X` is optional. If you specify `XF`, `garchpred` ignores it.

`[SigmaForecast, MeanForecast] = garchpred(Spec, Series, NumPeriods, X, XF)` forecasts both the conditional mean and standard deviation of the univariate return series, `Series`. `X` and `XF` are optional. However, for `MeanForecast`, if you specify `X`, you must also specify `XF`. For `SigmaForecast`, `garchpred` ignores `XF`.

`[SigmaForecast, MeanForecast, SigmaTotal, MeanRMSE] = garchpred(Spec, Series, NumPeriods)` in addition to forecasting the conditional mean and standard deviation of the univariate return series, computes the volatility forecasts of asset returns over multiperiod holding intervals, and the standard errors of conditional mean forecasts. If you compute `SigmaTotal` or `MeanRMSE`, `SigmaForecast` and `MeanForecast` can have no regression component.If you compute `SigmaTotal` or `MeanRMSE`, `SigmaForecast` and `MeanForecast` can have no regression component.

`garchpred` requires a complete conditional mean specification to correctly infer the innovations process that drives the forecasts. Because of this, you would typically use the same regression matrix of observed returns (`X`), if any, that you used for simulation (using `garchsim`) and/or estimation (using `garchfit`). `XF`, however, is just the forecast of `X`, and you *only* need it to forecast the conditional mean (`MeanForecast`). If you want to forecast only the conditional variance (`SigmaForecast`), `XF` is unnecessary.

`SigmaForecast` Matrix of minimum mean square error (MSE) forecasts of the conditional standard deviations of `Series` on a per period basis. `SigmaForecast` has `NumPeriods` rows and the same number of columns as `Series`. The first row contains the one-period-ahead forecast for each realization of `Series`, the second row contains the two-period-ahead forecast, and so on. If a forecast horizon is > `1` (i.e., `NumPeriods > 1`), `garchpred` returns the per-period forecasts of all intermediate horizons, as well as the forecast at the specified horizon which is in the last row.

`MeanForecast` Matrix of minimum MSE forecasts of the conditional mean of `Series` on a per-period basis. `MeanForecast` is the same size as `SigmaForecast`. The first row contains the forecast in the first period for each realization of `Series`, the second row contains the forecast in the second period, and so on.

Both `X` and `XF` must be non-empty for `MeanForecast` to have a regression component. If `X` and `XF` are empty (`[]`) or not specified, `MeanForecast` is based on the ARMA model. If you specify `X` and `XF`, `MeanForecast` is based on the full ARMAX model.

`SigmaTotal` Matrix of minimum mean square error (MSE) volatility forecasts of `Series` over multiperiod holding intervals. `SigmaTotal` is the same size as `SigmaForecast`. The first row contains the standard deviation of returns expected for assets held for one period for each realization of `Series`, the second row contains the standard deviation of returns expected for assets held for two periods, and so on. The last row contains the volatility forecast of the cumulative return obtained if an asset was held for the entire `NumPeriods` forecast horizon.

`garchpred` computes the elements of `SigmaTotal` by taking the square root of

$$var_t\left[\sum_{i=1}^{s} y_{t+i}\right] = \sum_{i=1}^{s}\left[\left[1+\sum_{j=1}^{s-i}\psi_j\right]^2 \mathrm{E}_t(\sigma_{t+i}^2)\right]$$

where s is the forecast horizon of interest (`NumPeriods`), and ψ_j is the coefficient of the jth lag of the innovations process in an infinite-order MA representation of the conditional mean model (see the function `garchma`).

In the special case of the default model for the conditional mean, $y_t = C + \varepsilon_t$, this reduces to

$$var_t\left[\sum_{i=1}^{s} y_{t+i}\right] = \sum_{i=1}^{s} \mathrm{E}_t(\sigma_{t+i}^2)$$

The `SigmaTotal` forecasts are correct for continuously compounded returns, and approximate for periodically compounded returns. `SigmaTotal` is the same size as `SigmaForecast` if the conditional mean is modeled as a stationary invertible ARMA process.

If you specify `X` and/or `XF`, `SigmaTotal = []`.

`MeanRMSE` Matrix of root mean square errors (RMSE) associated with `MeanForecast`. That is, `MeanRMSE` is the conditional standard deviation of the forecast errors (i.e., the standard error of the forecast) of the corresponding `MeanForecast` matrix. `MeanRMSE` is the same size as `MeanForecast` and `garchpred` organizes it in exactly the same manner, provided the conditional mean is modeled as a stationary/invertible ARMA process.

If you specify `X` and/or `XF`, `MeanRMSE = []`.

Note `garchpred` calls the function `garchinfer` to access the past history of innovations and conditional standard deviations inferred from `Series`. If you need the innovations and conditional standard deviations, call `garchinfer` directly.

See Also

`garchfit`, `garchinfer`, `garchma`, `garchset`, `garchsim`

References

Baillie, R.T., T. Bollerslev (1992), “Prediction in Dynamic Models with Time-Dependent Conditional Variances,” *Journal of Econometrics*, Vol. 52, pp. 91-113.

Bollerslev, T. (1986), “Generalized Autoregressive Conditional Heteroskedasticity,” *Journal of Econometrics*, Vol. 31, pp. 307-327.

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Engle, Robert (1982), "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation,” *Econometrica*, Vol. 50, pp. 987-1007.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

Purpose

Create or modify GARCH specification structure

Syntax

```
garchset
Spec = garchset
Spec = garchset('Parameter1', Value1, 'Parameter2', Value2, ...)
Spec = garchset(OldSpec, 'Parameter1', Value1, ...)
```

Arguments

`Parameter1, Parameter2, ...`	String representing the name of a valid parameter field of the output specification structure `Spec`. Table 2-1, GARCH Specification Parameters below lists the valid parameters. The GARCH Toolbox ignores case for parameter names.
`Value1, Value2, ...`	Value assigned to the corresponding `Parameter`.
`OldSpec`	(optional) Existing GARCH specification structure. Fields of the structure were previously generated by calling `garchset` or `garchfit`.

Description

`garchset` provides the main user interface for specifying a GARCH model, and is the preferred method for creating and modifying GARCH specification structures. Use `garchget` to retrieve the values of specification structure parameters.

`garchset` (with no input arguments and no output arguments) displays all parameter names and the default values where appropriate.

`Spec = garchset` creates a GARCH specification structure `Spec` with all fields set to their default settings. This default GARCH specification structure models an observed univariate return series as a constant, C, plus GARCH(1,1) conditionally Gaussian innovations. The C + GARCH(1,1) model is the default model of the GARCH Toolbox. You can use this `Spec` as input to `garchfit`, but it is invalid as input to `garchpred` or `garchsim`.

`Spec = garchset('Parameter1', Value1, 'Parameter2', Value2, ...)` creates a GARCH specification structure `Spec` using the parameter/value pairs specified in the input argument list. The `Parameter` part of the pair must be a valid GARCH specification structure field. `garchset` assigns the `Value` part of

the pair to its paired Parameter field. If you specify coefficient vectors (AR, MA, GARCH, ARCH) but not their corresponding model orders (R, M, P, Q), garchset infers the values of the model orders from the lengths of the coefficient vectors. In all other cases, garchset sets all parameters you do not specify to their respective defaults. A parameter name needs to include only sufficient leading characters to uniquely identify the parameter.

Spec = garchset(OldSpec, 'Parameter1', Value1, ...) modifies an existing GARCH specification structure, OldSpec, by changing the named parameters to the specified values.

Spec — GARCH specification structure. This structure contains the orders and coefficients (if specified) of the conditional mean and variance specifications of a GARCH model. It also contains the parameters associated with the function fmincon in the MATLAB Optimization Toolbox.

A GARCH specification structure includes the parameters shown in Table 2-1, GARCH Specification Parameters.

Table 2-1: GARCH Specification Parameters

Parameter	Description	Possible Values
Comment	User-defined summary comment	String. The default lists expressions for the mean and variance models derived from the current values of R, M, P, and Q. For example, 'Mean: ARMAX(0,0,?); Variance: GARCH(1,1)'. If you explicitly specify a comment, the toolbox does not overwrite it.
R	Auto-regressive component of the conditional mean model order of an ARMA(R,M) model	Nonnegative integer scalar. Default = 0.
M	Moving average component of the conditional mean model order of an ARMA(R,M) model	Nonnegative integer scalar. Default = 0.

Table 2-1: GARCH Specification Parameters (Continued)

Parameter	Description	Possible Values
`P`	GARCH component of the conditional variance model order of an GARCH(P,Q) model	Nonnegative integer scalar. `P` must be `0` if `Q` is `0`. Default = `0`.
`Q`	ARCH component of the conditional variance model order of an GARCH(P,Q) model	Nonnegative integer scalar. Default = `0`.
`Distribution`	Conditional distribution of innovations	String. The only valid value is `'Gaussian'`.
`C`	Conditional mean constant	Scalar coefficient. Default = `[]`.
`AR`	Conditional mean auto-regressive coefficients	Vector of `R` coefficients of lagged returns. Default = `[]`.
`MA`	Conditional mean moving average coefficients	Vector of `M` coefficients of lagged innovations. Default = `[]`.
`Regress`	Conditional mean regression coefficients	Vector of coefficients. Default = `[]`.
`K`	Conditional variance constant	Positive scalar coefficient. Default = `[]`.
`GARCH`	Conditional variance coefficients for lagged variances	Vector of `P` nonnegative coefficients. Default = `[]`.
`ARCH`	Conditional variance coefficients for lagged squared residuals	Vector of `Q` nonnegative coefficients. Default = `[]`.
`FixC`	Equality constraint indicator for `C` coefficient of the conditional mean	Boolean scalar. Default = `0`.
`FixAR`	Equality constraint indicator for `AR` coefficients of the conditional mean	Boolean vector. Default = `[0, 0, ..., 0]`.
`FixMA`	Equality constraint indicator for `MA` coefficients of the conditional mean	Boolean vector. Default = `[0, 0, ..., 0]`.

Table 2-1: GARCH Specification Parameters (Continued)

Parameter	Description	Possible Values
`FixRegress`	Equality constraint indicator for the `REGRESS` coefficients of the conditional mean	Boolean vector. Default = `[0, 0, ..., 0]`.
`FixK`	Equality constraint indicator for the `K` coefficient of the conditional variance	Boolean scalar. Default = `0`.
`FixGARCH`	Equality constraint indicator for the `GARCH` coefficients of the conditional variance	Boolean vector. Default = `[0, 0, ..., 0]`.
`FixARCH`	Equality constraint indicator for the `ARCH` coefficients of the conditional variance	Boolean vector. Default = `[0, 0, ..., 0]`.
`Display`	Display flag for iterative optimization information	String. Valid values are **`on`** (default) and **`off`**.
`MaxFunEvals`	Maximum number of log-likelihood objective function evaluations allowed in the estimation process	Positive integer. Default = `(100 *` number of parameters in the model). For a Gaussian distribution, this is `100 * (2 + R + M + Nx + P + Q)` where `Nx` is the number of explanatory variables in the regression component of the conditional mean.
`MaxIter`	Maximum number of iterations allowed in the estimation process	Positive integer. Default = `400`.
`TolCon`	Termination tolerance on constraint violation	Positive scalar. Default = `1e-006`.
`TolFun`	Termination tolerance on the objective function value	Positive scalar. Default = `1e-006`.
`TolX`	Termination tolerance on parameter estimates	Positive scalar. Default = `1e-006`.

Example

This example creates a GARCH(1,1) model and prints the specification structure. The nested `Optimization` structure, shown in the printed specification structure, contains the `Display`, `MaxFunEvals`, `MaxIter`, `TolCon`, `TolFun`, and `TolX` parameters. Use `garchget` to retrieve the values of these parameters.

```
spec = garchset('P',1,'Q',1)   % Create a GARCH(P=1,Q=1) model.

spec =

         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(1,1)'
               R: 0
               M: 0
               P: 1
               Q: 1
    Distribution: 'Gaussian'
               C: []
              AR: []
              MA: []
         Regress: []
               K: []
           GARCH: []
            ARCH: []
            FixC: []
           FixAR: []
           FixMA: []
      FixRegress: []
            FixK: []
        FixGARCH: []
         FixARCH: []
    Optimization: [1x1 struct]

spec = garchset(spec,'Q',2)    % Change it to a GARCH(P=1,Q=2)
                               % model.
spec =

         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(1,2)'
               R: 0
               M: 0
               P: 1
```

```
                Q: 2
     Distribution: 'Gaussian'
                C: []
               AR: []
               MA: []
          Regress: []
                K: []
            GARCH: []
             ARCH: []
             FixC: []
            FixAR: []
            FixMA: []
      FixRegress: []
             FixK: []
        FixGARCH: []
         FixARCH: []
     Optimization: [1x1 struct]
```

See Also

garchfit, garchget, garchpred, garchsim
optimset (in the online MATLAB Function Reference)

Purpose

Univariate GARCH process simulation

Syntax

```
[Innovations, Sigma, Series] = garchsim(Spec, NumSamples, NumPaths,
   Seed, X)
```

Arguments

`Spec`	GARCH specification structure for the conditional mean and variance models. You create `Spec` by calling the function `garchset` or the estimation function `garchfit`. The conditional mean can be of general ARMAX form and conditional variance of general GARCH form.
`NumSamples`	(optional) Positive integer indicating the number of samples `garchsim` generates for each path of the `Innovations`, `Sigma`, and `Series` outputs. If `NumSamples = []` or is not specified, the default is `100`.
`NumPaths`	(optional) Positive integer indicating the number of sample paths (realizations) `garchsim` generates for the `Innovations`, `Sigma`, and `Series` outputs. If `NumPaths = []` or is not specified, the default is `1`, i.e. `Innovations`, `Sigma` and `Series` are column vectors.

`Seed` (optional) Scalar random number generator seed. If `Seed = []` or is not specified, the default is `0` (the MATLAB initial state).

`X` (optional) Time series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (e.g., the return series of an equity index), and represents the past history of the explanatory data. Each column of `X` is an individual time series used as an explanatory variable in the regression component of the conditional mean. In each column, the first row contains the oldest observation and the last row the most recent.

If `X = []` or is not specified, the conditional mean has no regression component. If specified, then at least the most recent `NumSamples` observations of each return series must be valid (i.e., non-`NaN`). When the number of valid observations in each series exceeds `NumSamples`, `garchsim` uses only the most recent `NumSamples` observations of `X`.

Description

`[Innovations, Sigma, Series] = garchsim(Spec, NumSamples, NumPaths, Seed, X)` simulates sample paths for return series, innovations, and conditional standard deviation processes, using specifications for the conditional mean and variance of a univariate time series as input. `garchsim` samples each of `NumPaths` sample paths at `NumSamples` observations.

`Innovations` `NumSamples` by `NumPaths` matrix of innovations, representing a mean zero, discrete-time stochastic process. The `Innovations` time series follows the conditional variance (GARCH) specification defined in `Spec`. Rows are sequential times samples, columns are independent realizations.

`Sigma`	`NumSamples` by `NumPaths` matrix of conditional standard deviations of the corresponding `Innovations` matrix. `Innovations` and `Sigma` are the same size. Rows are sequential times samples. Columns are independent realizations.
`Series`	`NumSamples` by `NumPaths` matrix of the return series of interest. `Series` is the dependent stochastic process and follows the conditional mean specification of general ARMAX form defined in `Spec`. Rows are sequential times samples. Columns are independent realizations.

See Also

`garchfit`, `garchget`, `garchpred`, `garchset`

References

Bollerslev, T. (1986), "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of Econometrics*, Vol. 31, pp. 307-327.

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Engle, Robert (1982), "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, Vol. 50, pp. 987-1007.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

lagmatrix

Purpose

Create a lagged time series matrix

Syntax

```
XLAG = lagmatrix(X, Lags)
```

Arguments

`X` Time series of explanatory data. `X` can be a vector or a matrix. As a vector (row or column), `X` represents a univariate time series whose first element contains the oldest observation and whose last element contains the most recent observation. As a matrix, `X` represents a multivariate time series whose rows correspond to time indices in which the first row contains the oldest observations and the last row contains the most recent observations. `lagmatrix` assumes that observations across any given row occur at the same time. Each column is an individual time series.

`Lags` Vector of integer lags. `lagmatrix` applies the first lag to every series in `X`, then applies the second lag to every series in `X`, and so forth. To include a time series as is, include a `0` lag. Positive lags correspond to delays, and shift a series back in time. Negative lags correspond to leads, and shift a series forward in time.

Description

`XLAG = lagmatrix(X, Lags)` creates a lagged (i.e., shifted) version of a time series matrix. The `lagmatrix` function is useful for creating a regression matrix of explanatory variables for fitting the conditional mean of a return series.

`XLAG` Lagged transform of the time series `X`. To create `XLAG`, `lagmatrix` shifts each time series in `X` by the first lag, then shifts each time series in `X` by the second lag, and so forth. Since `XLAG` represents an explanatory regression matrix, each column is an individual time series. `XLAG` has the same number of rows as there are observations in `X`, but its column dimension is equal to the product of the number of columns in `X` and the length of `Lags`. `lagmatrix` uses a `NaN` (Not-a-Number) to indicate an undefined observation.

Example

The following example creates a bivariate time series matrix `X` with five observations each, then creates a lagged matrix `XLAG` composed of `X` and the first two lags of `X`. The result, `XLAG`, is a 5-by-6 matrix.

```
X = [1 -1; 2 -2 ;3 -3 ;4 -4 ;5 -5]  % Create a simple bivariate
                                     % series.
X =

     1    -1
     2    -2
     3    -3
     4    -4
     5    -5
XLAG = lagmatrix(X , [0 1 2])       % Create the lagged matrix.

XLAG =

     1    -1   NaN   NaN   NaN   NaN
     2    -2     1    -1   NaN   NaN
     3    -3     2    -2     1    -1
     4    -4     3    -3     2    -2
     5    -5     4    -4     3    -3
```

See Also

`filter`, `isnan`, and `nan` (in the online MATLAB Function Reference)

lbqtest

Purpose

Ljung-Box Q-statistic lack-of-fit hypothesis test

Syntax

```
[H, pValue, Qstat, CriticalValue] = lbqtest(Series, Lags, Alpha,
   DoF)
```

Arguments

`Series` Vector of observations of a univariate time series for which `lbqtest` computes the sample Q-statistic. The last row of `Series` contains the most recent observation of the stochastic sequence. Typically, `Series` is either the sample residuals derived from fitting a model to an observed time series, or the standardized residuals obtained by dividing the sample residuals by the conditional standard deviations.

`Lags` (optional) Vector of positive integers indicating the lags of the sample autocorrelation function included in the Q-statistic. If specified, each lag must be less than the length of `Series`. If `Lags = []` or is not specified, the default is `Lags = min([20, length(Series)-1])`.

`Alpha` (optional) Significance level(s). `Alpha` can be a scalar applied to all lags, or a vector the same length as `Lags`. If `Alpha = []` or is not specified, the default is `0.05`. For all elements, α, of `Alpha`, $0 < \alpha < 1$.

`DoF` (optional) Degree(s) of freedom. `DoF` can be a scalar applied to all lags, or a vector the same length as `Lags`. If specified, all elements of `DoF` must be positive integers less than the corresponding element of `Lags`. If `DoF = []` or is not specified, the elements of `Lags` serve as the default degrees of freedom for the Chi-Square distribution.

Description

`[H, pValue, Qstat, CriticalValue] = lbqtest(Series, Lags, Alpha, DoF)` performs the Ljung-Box lack-of-fit hypothesis test for model misspecification, which is based on the Q-statistic

$$Q = N(N+2)\sum_{k=1}^{L}\frac{r_k^2}{(N-k)}$$

where N = sample size, L = number of autocorrelation lags included in the statistic, and r_k^2 is the squared sample autocorrelation at lag k. Once you fit a univariate model to an observed time series, you can use the Q-statistic as a lack-of-fit test for a departure from randomness. Under the null hypothesis that the model fit is adequate, the test statistic is asymptotically Chi-Square distributed.

`H`	Boolean decision vector. 0 indicates acceptance of the null hypothesis that the model fit is adequate (no serial correlation at the corresponding element of `Lags`). 1 indicates rejection of the null hypothesis. `H` is the same size as `Lags`.
`pValue`	Vector of P-values (significance levels) at which `lbqtest` rejects the null hypothesis of no serial correlation at each lag in `Lags`.
`Qstat`	Vector of Q-statistics for each lag in `Lags`.
`CriticalValue`	Vector of critical values of the Chi-Square distribution for comparison with the corresponding element of `Qstat`.

Example

Create a vector of 100 Gaussian random numbers, then compute the Q-statistic for autocorrelation lags 20 and 25 at the 10 percent significance level.

```
randn('state',100)               % Start from a known state.
Series         = randn(100,1);   % 100 Gaussian deviates ~ N(0,1)
[H,P,Qstat,CV] = lbqtest(Series, [20 25]', 0.10);
[H,P,Qstat,CV]

ans =

         0    0.9615   10.3416   28.4120
         0    0.9857   12.1015   34.3816
```

See Also

`archtest`, `autocorr`

Reference

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Gourieroux, C., *ARCH Models and Financial Applications*, Springer-Verlag, 1997.

Purpose

Likelihood ratio hypothesis test

Syntax

```
[H, pValue, Ratio, CriticalValue] = lratiotest(BaseLLF, NullLLF,
   DoF, Alpha)
```

Arguments

`BaseLLF`	Scalar value of the optimized log-likelihood objective function of the baseline, unrestricted estimate. `lratiotest` assumes `BaseLLF` is the output of the estimation function `garchfit`, or the inference function `garchinfer`.
`NullLLF`	Vector of optimized log-likelihood objective function values of the restricted estimates. `lratiotest` assumes you obtained the `NullLLF` values using `garchfit` or `garchinfer`.
`DoF`	Degrees of freedom (i.e, the number of parameter restrictions) associated with each value in `NullLLF`. `DoF` can be a scalar applied to all values in `NullLLF`, or a vector the same length as `NullLLF`. All elements of `DoF` must be positive integers.
`Alpha`	(optional) Significance levels of the hypothesis test. `Alpha` can be a scalar applied to all values in `NullLLF`, or a vector the same length as `NullLLF`. If `Alpha = []` or is not specified, the default is `0.05`. For all elements, α, of `Alpha`, $0 < \alpha < 1$.

Description

`[H, pValue, Ratio, CriticalValue] = lratiotest(BaseLLF, NullLLF, DoF, Alpha)` performs the likelihood ratio hypothesis test. `lratiotest` uses as input the optimized log-likelihood objective function (LLF) value associated with an unrestricted maximum likelihood parameter estimate, and the LLF values associated with restricted parameter estimates.

The unrestricted LLF is the baseline case used to fit conditional mean and variance specifications to an observed univariate return series. The restricted models determine the null hypotheses of each test, and the number of restrictions they impose determines the degrees of freedom of the resulting Chi-Square distribution.

`BaseLLF` is usually the LLF of a larger estimated model and serves as the alternative hypothesis. Elements of `NullLLF` are then the LLFs associated with smaller, restricted specifications. `BaseLLF` should exceed the values in `NullLLF`, and the asymptotic distribution of the test statistic is Chi-Square distributed with degrees of freedom equal to the number of restrictions.

`H`	Vector of Boolean decisions the same size as `NullLLF`. A `0` indicates acceptance of the restricted model under the null hypothesis. `1` indicates rejection of the restricted, null hypothesis model relative to the unrestricted alternative associated with `BaseLLF`.
`pValue`	Vector of P-values (significance levels) at which `lratiotest` rejects the null hypothesis of each restricted model. `pValue` is the same size as `NullLLF`.
`Ratio`	Vector of likelihood ratio test statistics the same size as `NullLLF`. The test statistic is $$Ratio = 2(BaseLLF - NullLLF)$$
`CriticalValue`	Vector of critical values of the Chi-Square distribution. `CriticalValue` is the same size as `NullLLF`.

See Also

`garchfit`, `garchinfer`

Reference

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

Purpose

Plot or return computed sample partial auto-correlation function

Syntax

```
[PartialACF, Lags, Bounds] = parcorr(Series, nLags, R, nSTDs)
```

Arguments

`Series` Vector of observations of a univariate time series for which `parcorr` returns or plots the sample partial auto-correlation function (partial ACF). The last element of `Series` contains the most recent observation of the stochastic sequence.

`nLags` (optional) Positive, scalar integer indicating the number of lags of the partial ACF to compute. If `nLags = []` or is not specified, `parcorr` computes the partial ACF sequence at lags 0, 1, 2, ..., *T*, where *T* = `min([20, length(Series)-1])`.

`R` (optional) Nonnegative integer scalar indicating the number of lags beyond which `parcorr` assumes the theoretical partial ACF is zero. Assuming that `Series` is an AR(R) process, the estimated partial ACF coefficients at lags `> R` are approximately zero-mean, independently distributed Gaussian variates. In this case, the standard error of the estimated partial ACF coefficients of a fitted `Series` with *N* observations is approximately $1/\sqrt{N}$ for lags `> R`. If `R = []` or is not specified, the default is `0`. The value of `R` must be `< nLags`.

`nSTDs` (optional) Positive scalar indicating the number of standard deviations of the sample partial ACF estimation error to display, assuming that Series is an AR(R) process. If the `R`th regression coefficient (i.e., the last ordinary least squares (OLS) regression coefficient of `Series` regressed on a constant and `R` of its lags) includes *N* observations, specifying `nSTDs` results in confidence bounds at (`nSTDs`$/\sqrt{N}$). If `nSTDs = []` or is not specified, the default is `2` (i.e., approximate 95 percent confidence interval).

Description

`parcorr(Series, nLags, R, nSTDs)` computes and plots the sample partial auto-correlation function (partial ACF) of a univariate, stochastic time series. `parcorr` computes the partial ACF by fitting successive autoregressive models of orders 1, 2, ... by ordinary least squares, retaining the last coefficient of each

regression. To plot the partial ACF sequence without the confidence bounds, set `nSTDs = 0`.

`[PartialACF, Lags, Bounds] = parcorr(Series, nLags, R, nSTDs)` computes and returns the partial ACF sequence.

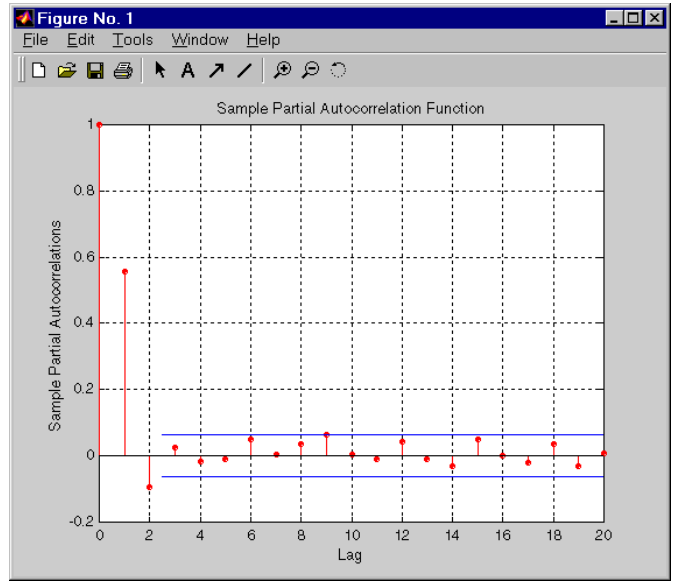
`PartialACF`	Sample partial ACF of `Series`. `PartialACF` is a vector of length `nLags` + 1 corresponding to lags 0, 1, 2, ..., `nLags`. The first element of `PartialACF` is unity, i.e., `PartialACF(1) = 1` = OLS regression coefficient of `Series` regressed upon itself. `parcorr` includes this element as a reference.
`Lags`	Vector of lags corresponding to `PartialACF(0, 1, 2, . . ., nLags)`.
`Bounds`	Two-element vector indicating the approximate upper and lower confidence bounds, assuming that `Series` is an AR(R) process. Note that `Bounds` is approximate for lags > `R` only.

Example

Create a stationary AR(2) process from a sequence of 1000 Gaussian deviates, and then visually assess whether the partial ACF is zero for lags > 2.

```
randn('state',0)               % Start from a known state.
x = randn(1000,1);             % 1000 Gaussian deviates ~ N(0,1).
y = filter(1,[1 -0.6 0.08],x); % Create a stationary AR(2)
                               % process.
[PartialACF, Lags, Bounds] = parcorr(y , [] , 2); % Compute the
                               % partial ACF with 95 percent
                               % confidence.
[Lags, PartialACF]

ans =

         0    1.0000
    1.0000    0.5570
    2.0000   -0.0931
    3.0000    0.0249
    4.0000   -0.0180
    5.0000   -0.0099
    6.0000    0.0483
    7.0000    0.0058
    8.0000    0.0354
```

```
    9.0000    0.0623
   10.0000    0.0052
   11.0000   -0.0109
   12.0000    0.0421
   13.0000   -0.0086
   14.0000   -0.0324
   15.0000    0.0482
   16.0000    0.0008
   17.0000   -0.0192
   18.0000    0.0348
   19.0000   -0.0320
   20.0000    0.0062

Bounds

Bounds =

    0.0633
   -0.0633

parcorr(y , [] , 2)            % Use the same example, but plot
                               % the partial ACF sequence with
                               % confidence bounds.
```



See Also

`autocorr`, `crosscorr`
`filter` (in the online MATLAB Function Reference)

References

Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

Purpose

Convert a price series to a return series

Syntax

```
[RetSeries, RetIntervals] = price2ret(TickSeries, TickTimes,
   Method)
```

Arguments

`TickSeries` Time series of price data. `TickSeries` can be a vector (row or column) or a matrix:

- As a vector, `TickSeries` represents a univariate price series. The length of the vector is the number of observations (`NUMOBS`). The first element contains the oldest observation, and the last element the most recent.
- As a matrix, `TickSeries` represents a `NUMOBS`-by-number of assets (`NUMASSETS`) matrix of asset prices. Rows correspond to time indices. The first row contains the oldest observations and the last row the most recent. `price2ret` assumes the observations across a given row occur at the same time for all columns, and each column is a price series of an individual asset.

`TickTimes` (optional) A `NUMOBS` element vector of monotonically increasing observation times. Times are numeric and taken either as serial date numbers (day units), or as decimal numbers in arbitrary units (e.g., yearly). If `TickTimes = []` or is not specified, then `price2ret` assumes sequential observation times from 1, 2, ..., `NUMOBS`.

`Method` (optional) Character string indicating the compounding method to compute asset returns. If `Method = 'Continuous'`, `= []`, or is not specified, then `price2ret` computes continuously compounded returns. If `Method = 'Periodic'`, then `price2ret` assumes simple periodic returns. `Method` is case insensitive.

Description

`[RetSeries, RetIntervals] = price2ret(TickSeries, TickTimes, Method)` computes asset returns for `NUMOBS` price observations of `NUMASSETS` assets.

`RetSeries` Array of asset returns:

- When `TickSeries` is a `NUMOBS` element row (column) vector, `RetSeries` is a `NUMOBS-1` row (column) vector.
- When `TickSeries` is a `NUMOBS`-by-`NUMASSETS` matrix, `RetSeries` is a (`NUMOBS-1`)-by-`NUMASSETS` matrix. `price2ret` quotes the *i*th return of an asset for the period `TickTimes(i)` to `TickTimes(i+1)` and normalizes it by the time interval between successive price observations.

Assuming that

$$RetIntervals(i) = TickTimes(i+1) - TickTimes(i)$$

then if `Method = 'Continuous'`, `= []`, or is not specified, `price2ret` computes the continuously-compounded *i*th return of an asset as

$$RetSeries(i) = \frac{log\left[\frac{TickSeries(i+1)}{TickSeries(i)}\right]}{RetIntervals(i)}$$

If `Method` = '`Periodic`', then `price2ret` computes the *i*th simple return as

$$RetSeries(i) = \frac{\left[\frac{TickSeries(i+1)}{TickSeries(i)}\right] - 1}{RetIntervals(i)}$$

`RetIntervals` `NUMOBS-1` element vector of interval times between observations. If `TickTimes = []` or is not specified, `price2ret` assumes that all intervals are `1`.

Example

Create a stock price process continuously compounded at 10 percent, then convert the price series to a 10 percent return series.

```
S = 100*exp(0.10 * [0:19]');   % Create the stock price series
R = price2ret(S);              % Convert the price series to a
                               % 10 percent returns series
```

```
[S [R ; NaN]]                % Pad the return series so vectors
                              % are of same length. price2ret
                              % computes the i-th return from
                              % the i-th and i+1-th prices.

ans =

  100.0000    0.1000
  110.5171    0.1000
  122.1403    0.1000
  134.9859    0.1000
  149.1825    0.1000
  164.8721    0.1000
  182.2119    0.1000
  201.3753    0.1000
  222.5541    0.1000
  245.9603    0.1000
  271.8282    0.1000
  300.4166    0.1000
  332.0117    0.1000
  366.9297    0.1000
  405.5200    0.1000
  448.1689    0.1000
  495.3032    0.1000
  547.3947    0.1000
  604.9647    0.1000
  668.5894       NaN
```

See Also

ret2price

ret2price

Purpose

Convert a return series to a price series

Syntax

```
[TickSeries, TickTimes] = ret2price(RetSeries, StartPrice,
   RetIntervals, StartTime, Method)
```

Arguments

`RetSeries` Time series array of returns. `RetSeries` can be a vector (row or column) or a matrix:

- As a vector, `RetSeries` represents a univariate series of returns of a single asset. The length of the vector is the number of observations (`NUMOBS`). The first element contains the oldest observation, and the last element the most recent.
- As a matrix, `RetSeries` represents a `NUMOBS`-by-number of assets (`NUMASSETS`) matrix of asset returns. Rows correspond to time indices. The first row contains the oldest observations and the last row the most recent. `ret2price` assumes the observations across a given row occur at the same time for all columns, and each column is a return series of an individual asset.

`StartPrice` (optional) A `NUMASSETS` element vector of initial prices for each asset, or a single scalar initial price applied to all assets. If `StartPrice = []` or is not specified, all asset prices start at `1`.

`RetIntervals` (optional) A `NUMOBS` element vector of time intervals between return observations, or a single scalar interval applied to all observations. If `RetIntervals = []` or is not specified, `ret2price` assumes all intervals have length `1`.

`StartTime` (optional) Scalar starting time for the first observation, applied to the price series of all assets. The default is `0`.

`Method` (optional) Character string indicating the compounding method used to compute asset returns. If `Method = 'Continuous', = []`, or is not specified, then `ret2price` computes continuously compounded returns. If `Method = 'Periodic'` then `ret2price` computes simple periodic returns. `Method` is case insensitive.

Description

`[TickSeries, TickTimes] = ret2price(RetSeries, StartPrice, RetIntervals, StartTime, Method)` generates a price series for each of `NUMASSETS` assets, given the asset starting prices and `NUMOBS` return observations for each asset.

`TickSeries` Array of asset prices:

- When `RetSeries` is a `NUMOBS` element row (column) vector, `TickSeries` is a `NUMOBS+1` row (column) vector. The first element contains the starting price of the asset, and the last element the most recent price.
- When `RetSeries` is a `NUMOBS`-by-`NUMASSETS` matrix, then `RetSeries` is a (`NUMOBS+1`)-by-`NUMASSETS` matrix. The first row contains the starting price of the assets, and the last row contains the most recent prices.

`TickTimes` A `NUMOBS`+1 element vector of price observation times. The initial time is zero unless specified in `StartTime`.

Example

Create a stock price process continuously compounded at 10 percent. Compute 10 percent returns for reference, then convert the resulting return series to the original price series and compare results.

```
S = 100*exp(0.10 * [0:19]');   % Create the stock price series
R = price2ret(S);              % Convert the price series to a
                               % 10 percent returns series
P = ret2price(R, 100);         % Convert to the original price
                               % series
```

```
[S P]                          % Compare the original and
                               % computed price series

ans =

  100.0000  100.0000
  110.5171  110.5171
  122.1403  122.1403
  134.9859  134.9859
  149.1825  149.1825
  164.8721  164.8721
  182.2119  182.2119
  201.3753  201.3753
  222.5541  222.5541
  245.9603  245.9603
  271.8282  271.8282
  300.4166  300.4166
  332.0117  332.0117
  366.9297  366.9297
  405.5200  405.5200
  448.1689  448.1689
  495.3032  495.3032
  547.3947  547.3947
  604.9647  604.9647
  668.5894  668.5894
```

See Also

price2ret

A

Glossary

Akaike information criteria (AIC) - A model order selection criteria based on parsimony. More complicated models are penalized for the inclusion of additional parameters. See also Bayesian information criteria (BIC).

AR - Auto-Regressive. AR models include past observations of the dependent variable in the forecast of future observations.

ARCH - Auto-Regressive Conditional Heteroskedasticity. A time series technique in which past observations of the variance are used to forecast future variances. See also GARCH

ARMA - Auto-Regressive Moving Average. A time series model that includes both AR and MA components. See also AR and MA.

auto-correlation function (ACF) - Correlation sequence of a random time series with itself. See also cross-correlation function (XCF).

auto-regressive - See AR.

Bayesian information criteria (BIC) - A model order selection criteria based on parsimony. More complicated models are penalized for the inclusion of additional parameters. Since BIC imposes a greater penalty for additional parameters than AIC, BIC always provides a model with a number of parameters no greater than that chosen by AIC. See also Akaike information criteria (AIC).

conditional - Time series technique with explicit dependence on the past sequence of observations.

conditional mean - Time series model for forecasting the expected value of the return series itself.

conditional variance - Time series model for forecasting the expected value of the variance of the return series.

cross-correlation function (XCF) - Correlation sequence between two random time series. See also auto-correlation function (ACF).

equality constraint - A constraint, imposed during parameter estimation, by which a parameter is held fixed at a user-specified value.

excess kurtosis - A characteristic, relative to a standard normal probability distribution, whereby an area under the probability density function is reallocated from the center of the distribution to the tails (fat

tails). Samples obtained from distributions with excess kurtosis have a higher probability of containing outliers than samples drawn from a normal (Gaussian) density. Time series that exhibit a fat tail distribution are often referred to as leptokurtic.

explanatory variables - Time series used to explain the behavior of another observed series of interest. Explanatory variables are typically incorporated into a regression framework.

fat tails - See excess kurtosis.

relative to a standard normal probability distribution

GARCH - Generalized Auto-Regressive Conditional Heteroskedasticity. A time series technique in which past observations of the variance and variance forecast are used to forecast future variances. See also ARCH.

heteroskedasticity - Time-varying, or time-dependent, variance.

homoskedasticity - Time-independent variance. The GARCH Toolbox also refers to homoskedasticity as constant conditional variance.

i.i.d. - Independent, identically distributed.

innovations - A sequence of unanticipated shocks, or disturbances. The GARCH Toolbox uses innovations and residuals interchangeably.

leptokurtic - See excess kurtosis.

MA - Moving average. MA models include past observations of the innovations noise process in the forecast of future observations of the dependent variable of interest.

MMSE - Minimum mean square error. An technique designed to minimize the variance of the estimation or forecast error. See also RMSE.

moving average - See MA

objective function - The function to be numerically optimized. In the GARCH Toolbox, the objective function is the log-likelihood function of a random process.

partial auto-correlation function (PACF) - Correlation sequence estimated by fitting successive order auto-regressive models to a random

time series by least squares. The PACF is useful for identifying the order of an auto-regressive model.

path - A random trial of a time series process.

P-value - The lowest level of significance at which a test statistic is significant.

realization - See path.

residuals - See innovations.

RMSE - Root mean square error. The square root of the mean square error. See also MMSE.

standardized innovations - The innovations divided by the corresponding conditional standard deviation.

stationarity constraint - Constraint imposed during estimation such that the sum of the GARCH model conditional variance parameters is less than unity.

time series - Discrete-time sequence of observations of a random process. The type of time series of interest in the GARCH Toolbox is typically a series of returns, or relative changes of some underlying price series.

transient - A response, or behavior, of a time series that is heavily dependent on the initial conditions chosen to begin a recursive calculation. The transient response is typically undesirable, and initially masks the true steady-state behavior of the process of interest.

unconditional - Time series technique in which explicit dependence on the past sequence of observations is ignored. Equivalently, the time stamp associated with any observation is ignored.

volatility - The risk, or uncertainty, measure associated with a financial time series. The GARCH Toolbox associates volatility with standard deviation.

B

Bibliography

[1] Baillie, R.T., T, Bollerslev, "Prediction in Dynamic Models with Time-Dependent Conditional Variances," *Journal of Econometrics*, Vol. 52, pp. 91-113, 1992.

[2] Bera, A.K., H.L. Higgins, "A Survey of ARCH Models: Properties, Estimation and Testing," *Journal of Economic Surveys*, Vol. 7 no. 4, 1993.

[3] Bollerslev, T., "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return," *Review of Economics and Statistics*, Vol. 69, pp. 542-547, 1987.

[4] Bollerslev, T., "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of Econometrics*, Vol. 31, pp. 307-327, 1986.

[5] Bollerslev, T., R.Y. Chou, K.F. Kroner, "ARCH Modeling in Finance: A Review of the Theory and Empirical Evidence," *Journal of Econometrics*, Vol. 52, pp. 5-59, 1992.

[6] Bollerslev, T., R.F. Engle, D.B. Nelson, *Handbook of Econometrics: Volume IV* (Chapter 49, ARCH Models), pp. 2959-3038, Elsevier Science B.V., 1994.

[7] Box, G.E.P., G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, third edition, Prentice Hall, 1994.

[8] Engle, Robert, "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, Vol. 50, pp. 987-1007, 1982.

[9] Gourieroux, C., *ARCH Models and Financial Applications*, Springer-Verlag, 1997.

[10] Hamilton, J.D., *Time Series Analysis*, Princeton University Press, 1994.

Index

F

G

H

S

T

U

V

X