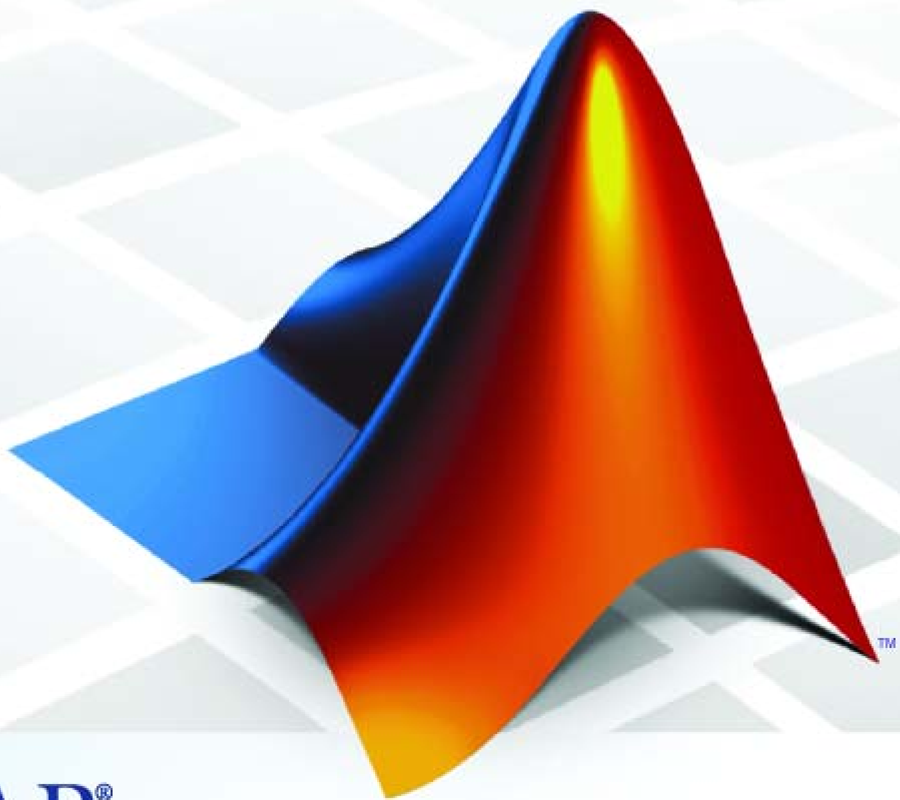
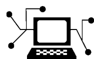
GARCH Toolbox™ 2
User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

GARCH Toolbox™ User's Guide

© COPYRIGHT 1999–2008 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

July 1999	First printing	New for Version 1.0 (Release 11)
November 2000	Online only	Revised for Version 1.0.1 (Release 12)
July 2002	Online only	Revised for Version 1.0.2 (Release 13)
November 2002	Second printing	Revised for Version 2.0 (Release 13+)
June 2004	Online only	Minor revision for Version 2.0.1 (Release 14)
August 2004	Third printing	Revised for Version 2.0.1
September 2005	Online only	Revised for Version 2.1 (Release 14SP3)
March 2006	Fourth printing	Revised for Version 2.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.3.1 (Release 2007a)
September 2007	Online only	Revised for Version 2.3.2 (Release 2007b)
March 2008	Online only	Revised for Version 2.4 (Release 2008a)

Contents

1 Getting Started

Product Overview **1-2**

What Is GARCH? **1-3**
About GARCH **1-3**
Modeling with GARCH **1-3**
Limitations of GARCH Modeling **1-4**

Expected Background **1-6**

Technical Conventions **1-7**
Array and Vector Size **1-7**
Vector Length **1-7**
Time-Series Arrays **1-7**
Conditional vs. Unconditional **1-8**
Precision **1-8**
Prices, Returns, and Compounding **1-8**
Stationary and Non-stationary Time Series **1-9**

Example Financial Time-Series Data Sets **1-12**
About the Examples in this Documentation **1-12**
DEM2GBP **1-12**
NASDAQ **1-13**
NYSE **1-13**
SDE_Data **1-14**

2 Introduction

About Financial Time Series Modeling **2-2**
Characteristics of Financial Time Series **2-2**
Forecasting and Correlation of Financial Time Series **2-5**

Serial Dependence in Innovations 2-5

Conditional Mean and Variance Models 2-7
About Conditional Mean and Variance Models 2-7
Conditional Mean Models 2-9
Conditional Variance Models 2-10

The Default Model 2-13

Primary Toolbox Functions 2-14

Example: Analysis and Estimation Using the Default Model 2-16
Pre-Estimation Analysis 2-16
Parameter Estimation 2-24
Post-Estimation Analysis 2-27

3 GARCH Specification Structures

Introduction 3-2

Associating Model Equation Variables with Corresponding Parameters in Specification Structures 3-4
About Specification Structure Parameter Names 3-4
Conditional Mean Model 3-4
Conditional Variance Models 3-5

Example: Interpreting Specification Structures 3-6

Working with Specification Structures 3-9
Creating Specification Structures 3-9
Modifying Specification Structures 3-11
Retrieving Specification Structure Values 3-12

4 Simulation of GARCH Models

Simulating Single and Multiple Paths **4-2**
Introduction **4-2**
Preparing the Example Data **4-2**
Simulating Single Paths **4-3**
Simulating Multiple Paths **4-5**

Working with Presample Data **4-7**
About Presample Data **4-7**
Automatically Generating Presample Data **4-7**
Running Simulations With User-Specified Presample Data **4-13**

5 Monte Carlo Simulation of Stochastic Differential Equations

Introduction **5-2**

Terminology **5-3**
Trials vs. Paths **5-3**
NTRIALS, NPERIODS, and NSTEPS **5-4**

Behavior and Syntax of SDE Objects **5-5**
Relationship Between SDE Models and Objects **5-5**
Displaying Objects **5-5**
Assigning and Referencing Object Parameters **5-6**
Constructing and Evaluating Models **5-6**

Parametric Specification **5-7**
General Parametric Specification **5-7**
General SDEs **5-7**
Drift and Diffusion Specifications **5-8**

Using SDE Objects to Create Models **5-11**
SDE Classes **5-11**

Creating Base SDE Objects 5-14
Creating Drift and Diffusion Objects 5-16
Creating Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) 5-19
Creating Stochastic Differential Equations from Linear Drift (SDELD) 5-20
Creating Brownian Motion (BM) Models 5-21
Creating Constant Elasticity of Variance (CEV) Models .. 5-22
Creating Geometric Brownian Motion (GBM) Models 5-23
Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) 5-24
Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models 5-25
Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models 5-26

Solving Problems with SDE Models **5-29**
Implementing Multidimensional Equity Market Models .. 5-29
Stochastic Interpolation and the Brownian Bridge 5-42
Inducing Dependence and Correlation 5-48
Incorporating Dynamic Behavior 5-51
End-of-Period Processes 5-57
User-Specified Random Number Generation: Stratified Sampling 5-63

Creating User-Specified Functions **5-69**
Evaluating Object Parameters, Noise, and End-of-Period Processing Functions 5-69
Random Number Generation Functions vs. End-of-Period Processing Functions 5-70

Managing Memory, Performance, and Solution Accuracy **5-72**
Managing Memory 5-72
Enhancing Performance 5-73
Optimizing Accuracy of Solutions 5-74

6 Estimation

Maximum Likelihood Estimation **6-2**

Initial Parameter Estimates **6-4**
User-Specified Initial Estimates **6-4**
Automatically Generated Initial Estimates **6-6**
Parameter Bounds **6-10**

Presample Observations **6-12**
Calculating Presample Data **6-12**
User-Specified Presample Observations **6-12**
Automatically Generated Presample Observations **6-13**

Termination Criteria and Optimization Results **6-15**
Optimization Parameters **6-15**
MaxIter and MaxFunEvals **6-15**
TolCon, TolFun, and TolX **6-16**
Convergence **6-17**
Optimization Results **6-17**
Constraint Violation Tolerance **6-18**

Examples: Specifying Your Own Presample Data to Estimate ARMA(R,M) Parameters **6-21**
Specifying Presample Data **6-21**
Presample Data and Transient Effects **6-24**
Alternative Technique for Estimating ARMA(R,M) Parameters **6-30**
Active Lower Bound Constraint **6-30**
Determining Convergence Status **6-34**

7 Forecasting the Conditional Mean and Standard Deviation of Return Series

Minimum Mean Square Error Forecasting **7-2**
About the Forecasting Engine **7-2**

Conditional Standard Deviations of Future Innovations .. **7-2**
Conditional Mean Forecasting of the Return Series **7-3**
MMSE Volatility Forecasting of Returns **7-3**
RMSE Associated with Conditional Mean Forecasts **7-4**

Generating Presample Observations **7-6**

Asymptotic Behavior for Long-Range Forecast Horizons **7-7**

Examples: Computing Forecasts **7-9**
Forecasting Using `garchpred` **7-9**
Volatility Forecasting over Multiple Periods **7-12**
Forecasting with Multiple Realizations **7-15**

8 Regression Components

Introduction **8-2**

Example: Incorporating a Regression Model into an Estimation **8-3**
Fitting a Model to a Simulated Return Series **8-3**
Fitting a Regression Model to the Same Return Series ... **8-5**

Simulation and Inference Using a Regression Component **8-8**

Forecasting Using a Regression Component **8-9**
Using Forecasted Explanatory Data **8-9**
Generating Forecasted Explanatory Data **8-10**

Ordinary Least Squares Regression **8-11**

Regression in a Monte Carlo Framework **8-13**

9 Univariate Unit Root Tests

Introduction 9-2
Critical Values 9-2
Serial Dependence 9-2

Dickey-Fuller Tests 9-4
Definitions of Operators 9-4
dfARTest 9-4
dfARDTest 9-4
dfTSTest 9-5

Phillips-Perron Tests 9-6
Definitions of Operators 9-6
ppARTest 9-6
ppARDTest 9-6
ppTSTest 9-7

How to Test for Unit Roots: Inputs and Outputs 9-8
About the Common Interface 9-8
Lags 9-8
Significance Level 9-9
TestType 9-9
Outputs 9-10

Interpretation of Results 9-11

Examples: Unit Root Tests 9-13
About These Examples 9-13
Testing GDP by OLS Regression with a Stationary Component 9-14
Testing T-Bill Rate by OLS Regression with a Drift Component 9-17

10 Model Selection and Analysis

Using The Autocorrelation and Partial Autocorrelation Functions **10-2**

Likelihood Ratio Tests **10-3**
Testing Support for a GARCH(2,1) Model **10-3**

Akaike and Bayesian Information Criteria **10-6**

Equality Constraints and Parameter Significance **10-9**
Specification Structure Fix Fields **10-9**
Comparing the GARCH (1, 1) Estimation Results with the GARCH (2,1) Model Fit to the NASDAQ Returns **10-11**

Equality Constraints and Initial Parameter Estimates **10-14**
About this Example **10-14**
Complete Model Specification **10-14**
Empty Fix Fields **10-15**
Limiting Use of Equality Constraints **10-16**

Examples: Simplicity and Parsimony **10-17**

11 Example Workflow: Estimation, Forecasting, and Simulation

Estimating the Model **11-3**

Forecasting **11-5**

Forecasting Using Monte Carlo Simulation **11-7**

Comparing Forecasts with Simulation Results **11-9**

12 Function Reference

Data Preprocessing **12-2**

GARCH Specification Structure **12-2**

GARCH Modeling **12-3**

General Utilities **12-3**

Graphics **12-4**

Statistics and Tests **12-4**

13 Functions — Alphabetical List

14 Method Reference

Monte Carlo Simulation of Stochastic Differential Equations (SDEs) **14-2**

Stochastic Differential Equation (SDE) Class Constructors **14-3**

15 Methods — Alphabetical List

A Bibliography

B Examples

Introduction B-2

Simulation B-2

Simulating Univariate Brownian Motion Models B-2

Monte Carlo Simulation of Stochastic Differential Equations B-2

Estimation B-3

Forecasting B-3

Regression B-3

Unit Root Tests B-4

Model Selection and Analysis B-4

Example Workflow: Estimation, Forecasting, and Monte Carlo Simulation B-4

Glossary

Index

1

Getting Started

Product Overview (p. 1-2)	GARCH Toolbox™ software usage and capabilities
What Is GARCH? (p. 1-3)	Defines GARCH, and characteristics of GARCH models that are commonly associated with financial time series
Expected Background (p. 1-6)	Describes the intended audience for this product.
Technical Conventions (p. 1-7)	Usage of common mathematical terms in this documentation
Example Financial Time-Series Data Sets (p. 1-12)	The financial time-series data sets, available in the MAT-file `garchdata.mat`, that you use in examples throughout this documentation

Product Overview

The GARCH Toolbox™ software, combined with MATLAB®, Optimization Toolbox™, and Statistics Toolbox™ software, provides an integrated computing environment for modeling the volatility of economic time series. The GARCH Toolbox software uses ARMAX conditional mean models combined with conditional variance models of GARCH, GJR, or EGARCH form to perform simulation, forecasting, and parameter estimation of time series in the presence of conditional heteroscedasticity. 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:

- Specify general ARMAX conditional mean models combined with conditional variance models of GARCH, GJR, or EGARCH form for univariate asset returns
- Estimate parameters of general ARMAX conditional mean models combined with conditional variance models of GARCH, GJR, or EGARCH form
- 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 autocorrelation, cross-correlation, and partial autocorrelation
- Convert price/return series to return/price series, and transform finite-order ARMA models to infinite-order AR and MA models
- Perform Monte Carlo simulation of univariate returns, innovations, and conditional volatilities
- Model dependent financial and economic variables, such as interest rates and equity prices, by performing Monte Carlo simulation of Stochastic Differential Equations (SDEs)

What Is GARCH?

In this section...

"About GARCH" on page 1-3

"Modeling with GARCH" on page 1-3

"Limitations of GARCH Modeling" on page 1-4

About GARCH

GARCH stands for *generalized autoregressive conditional heteroscedasticity*. You can think of heteroscedasticity as time-varying variance (*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 technique that you use to model the serial dependence of volatility.

In this documentation, whenever a time series is said to have *GARCH effects*, the series is *heteroscedastic*, meaning that its variances vary with time. If its variances remain constant with time, the series is *homoscedastic*.

Modeling with GARCH

GARCH modeling builds on advances in the understanding and modeling of volatility in the last decade. It takes into account excess kurtosis (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. Therefore, you can apply GARCH models to such diverse fields as:

- Risk management
- Portfolio management and asset allocation
- Option pricing
- Foreign exchange
- The term structure of interest rates

You can find highly significant GARCH effects in equity markets [7] for:

- Individual stocks
- Stock portfolios and indices
- Equity futures markets

These effects are important in areas such 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.
- Analyze time-varying risk premiums [7] as the uncertainty for rates over various horizons changes over time.
- Model foreign-exchange markets, which couple highly persistent periods of volatility and tranquility with significant fat-tail behavior [7].

Note Bollerslev [6] developed GARCH as a generalization of Engle's [14] original ARCH volatility modeling technique. He designed it to offer a more parsimonious model (using fewer parameters) that lessens the computational burden.

Limitations of GARCH Modeling

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

- GARCH models are only part of a solution. Although GARCH models usually apply 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 [18]. GARCH is explicitly designed to model time-varying conditional variances. However, GARCH models often fail to capture highly irregular phenomena. These include wild market fluctuations (for example, crashes and later rebounds) and other unanticipated events that can lead to significant structural change.

- GARCH models often fail to fully capture the fat tails observed in asset return series. Heteroscedasticity explains some, but not all, fat-tail behavior. To compensate for this limitation, fat-tailed distributions such as Student's t have been applied to GARCH modeling.

Expected Background

In general, this documentation assumes that you are familiar with the basic concepts of generalized autoregressive conditional heteroscedasticity (GARCH) modeling.

In designing GARCH Toolbox™ documentation, we assume your title is similar to one of these:

- Analyst, quantitative analyst
- Risk manager
- Portfolio manager
- Asset allocator
- 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
- Focus on quantitative approaches to financial problems

Technical Conventions

In this section...
"Array and Vector Size" on page 1-7
"Vector Length" on page 1-7
"Time-Series Arrays" on page 1-7
"Conditional vs. Unconditional" on page 1-8
"Precision" on page 1-8
"Prices, Returns, and Compounding" on page 1-8
"Stationary and Non-stationary Time Series" on page 1-9

Tip This section describes usage of common mathematical terms in this documentation. For definitions of GARCH-specific terms, see the "Glossary" on page Glossary-1.

Array and Vector Size

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. If two vectors are of the same size, they have the same length and the same orientation.

Vector Length

The *length* of a vector indicates only the number of elements in 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.

Time-Series Arrays

A *time series* is an ordered set of observations stored in a MATLAB® array. The rows of a time-series array correspond to time-tagged indices, or observations,

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. In this representation, a time-series array is column-oriented.

Note Some GARCH Toolbox™ functions can process univariate time-series arrays formatted as either row or column vectors. However, many functions now strictly enforce the column-oriented representation of a time series. To avoid ambiguity, format single realizations of univariate time series as column vectors. Representing a time series in column-oriented format avoids misinterpretation of the arguments. It also makes it easier for you to display data in the MATLAB Command Window.

Conditional vs. Unconditional

The term *conditional* implies explicit dependence on a past sequence of observations. The term *unconditional* applies more to long-term behavior of a time series, and assumes no explicit knowledge of the past.

Precision

The GARCH Toolbox software performs all its calculations in double precision. To set the numeric format for your display, click **File** > **Preferences** > **Command Window** > **Text display**. The default is **short**.

Prices, Returns, and Compounding

The GARCH Toolbox software assumes that time-series vectors and matrices are time-tagged series of observations. The toolbox lets you convert a given price series to a return series using either continuous compounding or simple periodic compounding.

If you denote successive price observations made at times 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

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

Simple periodic compounding defines the transformation as

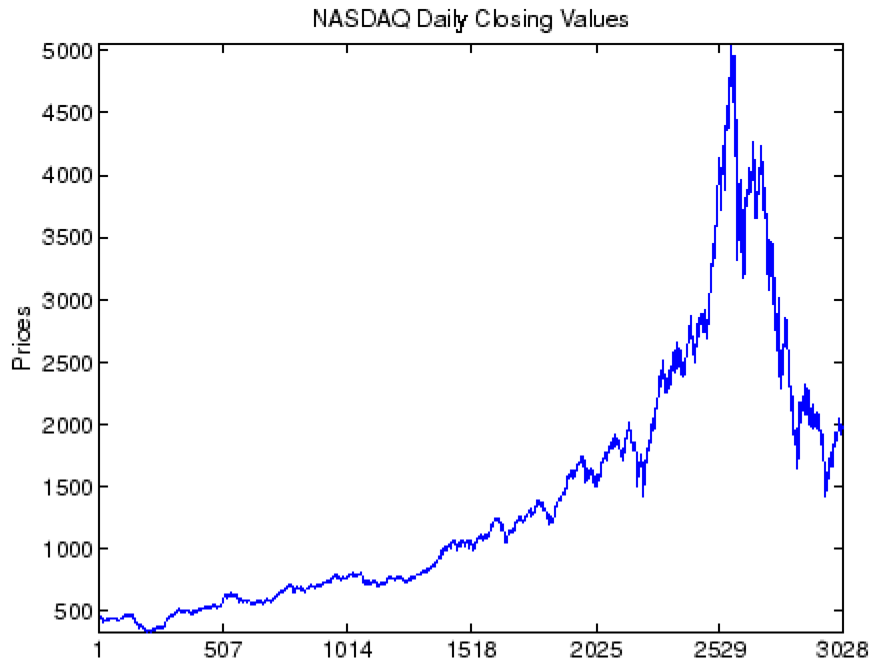
$$y_t = \frac{P_{t+1} - P_t}{P_t} = \frac{P_{t+1}}{P_t} - 1 \qquad \textbf{(1-2)}$$

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

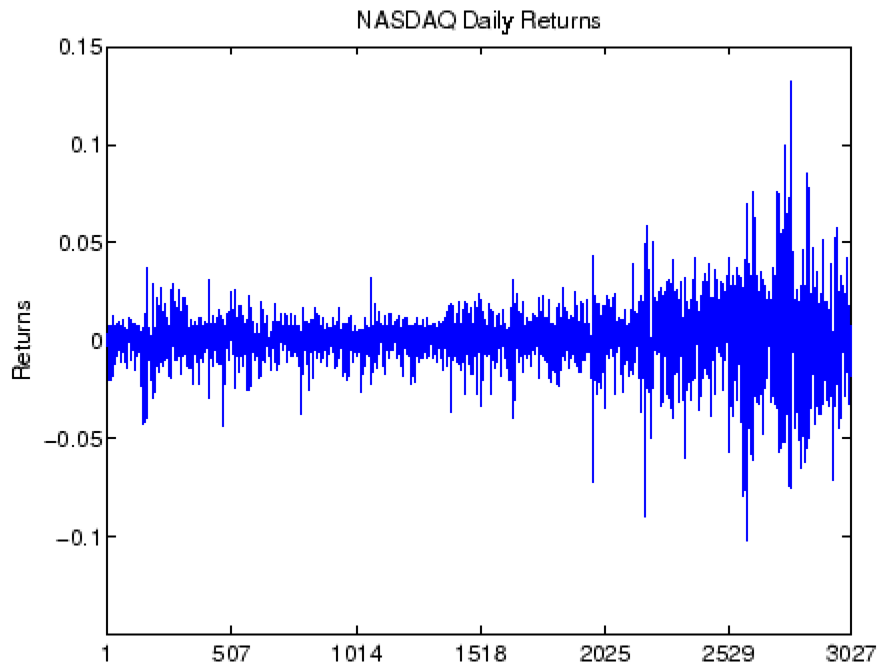
Stationary and Non-stationary Time Series

The GARCH Toolbox software assumes that return series are stationary processes. The price-to-return transformation generally guarantees a stable data set for GARCH modeling.

The following figure illustrates an equity price series. It shows daily closing values of the NASDAQ Composite Index, as described in "NASDAQ" on page 1-13. There appears to be no long-run average level about which the series evolves, indicating a non-stationary time series.



The following figure illustrates the continuously compounded returns associated with the same price series. In contrast, the returns appear to be stable over time, and the transformation from prices to returns has produced a stationary time series.


NASDAQ Daily Returns
0.15
0.1
0.05
0
–0.05
–0.1
Returns
1
507
1014
1518
2025
2529
3027

Example Financial Time-Series Data Sets

In this section...

"About the Examples in this Documentation" on page 1-12

"DEM2GBP" on page 1-12

"NASDAQ" on page 1-13

"NYSE" on page 1-13

"SDE_Data" on page 1-14

About the Examples in this Documentation

The results you obtain when you recreate examples in this documentation may differ slightly from those shown in the text because of differences in:

- Platforms (operating systems)
- Versions of the MATLAB® software
- Versions of the Optimization Toolbox™ software
- Versions of supporting math libraries

These differences in results propagate through later examples that use these results as input. These differences, however, do not affect the outcome of the examples.

DEM2GBP

The `DEM2GBP` series contains daily observations of the Deutschmark/British Pound foreign-exchange rate; that is, it is an FX price series. The sample period is from January 2, 1984, to December 31, 1991, for a total of 1975 daily observations of FX exchange rates.

This price series derives from the corresponding daily percentage nominal returns for the Deutschemark/British Pound exchange rate computed as

$$y_t = 100\ln(\frac{P_{t+1}}{P_t}) = 100[\ln(P_{t+1}) - \ln(P_t)]$$

where P_t is the bilateral Deutschmark/British Pound FX rate constructed from the corresponding U.S. dollar rates. The original nominal returns, expressed in percent, were originally published in Bollerslev and Ghysels [9].

You can also obtain the percentage returns data from the *Journal of Business and Economic Statistics* (JBES) FTP site:

- Go to `ftp://www.amstat.org/JBES_View/96-2-APR/bollerslev_ghysels`.
- Download the file `bollerslev.sec41.dat`.

The sample period discussed in the Bollerslev and Ghysels article is from January 3, 1984, to December 31, 1991, for a total of 1974 observations of daily percentage nominal returns. This data is from the Currency Web site, `http://www.oanda.com`. These returns, combined with an approximate closing exchange rate from January 2, 1984, allow an approximate reconstruction of the corresponding FX closing price series.

This particular FX price series appears in this documentation because it has been promoted as an informal benchmark for GARCH time-series software validation. See McCullough & Renfro [27], and Brooks, Burke, & Persand [11] for details. The estimation results published in these references are based on the original percentage returns. The GARCH Toolbox™ software presents the data as a price series merely to maintain consistency with other data sets included in this documentation.

NASDAQ

The `nasdaq` series contains daily closing values of the NASDAQ Composite Index. The sample period is from January 2, 1990, to December 31, 2001, for a total of 3028 daily equity index observations.

The NASDAQ Composite closing index values are from the Market Data section of the NASDAQ Web page, `http://www.nasdaq.com/`.

NYSE

The `NYSE` series contains daily closing values of the New York Stock Exchange Composite Index. The sample period is from January 2, 1990, to December

31, 2001, for a total of 3028 daily equity index observations of the NYSE Composite Index.

The NYSE Composite Index daily closing values are from the Market Information section of the NYSE Web page, `http://www.nyse.com/`.

SDE_Data

The `SDE_Data` series consists of a daily historical data set whose sample period is from February 7, 2001 to April 24, 2006, that includes the following:

- 3-month EURIBOR, quoted as an annual percentage rate and converted to daily effective yield.
- The closing index levels of representative large-cap equity indices of Canada (TSX Composite), France (CAC 40), Germany (DAX), Japan (Nikkei 225), UK (FTSE 100), and US (S&P 500).

Introduction

About Financial Time Series Modeling (p. 2-2)	General financial time-series modeling concepts
Conditional Mean and Variance Models (p. 2-7)	GARCH Toolbox™ models that describe conditional mean and variance
The Default Model (p. 2-13)	GARCH Toolbox default conditional mean and variance models
Primary Toolbox Functions (p. 2-14)	Core functions you use to perform estimation, simulation, and forecasting
Example: Analysis and Estimation Using the Default Model (p. 2-16)	Uses the default model to examine the Deutschmark/British Pound foreign-exchange rate series

About Financial Time Series Modeling

In this section...

"Characteristics of Financial Time Series" on page 2-2

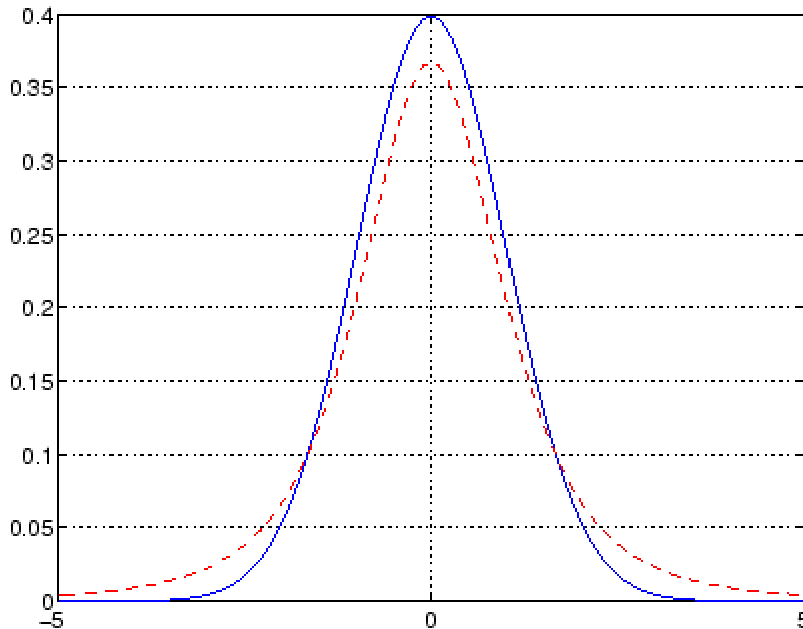
"Forecasting and Correlation of Financial Time Series" on page 2-5

"Serial Dependence in Innovations" on page 2-5

Characteristics of Financial Time Series

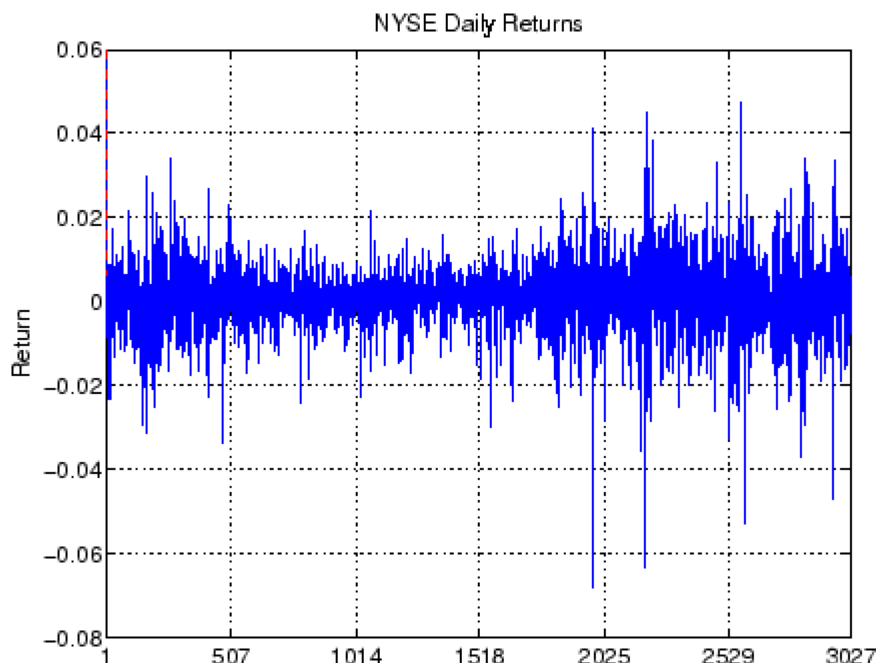
GARCH models are designed to capture characteristics that are commonly associated with financial time series, including fat tails, volatility clustering, and leverage effects.

Probability distributions for asset returns often exhibit fatter tails than the standard normal, or *Gaussian*, distribution. The fat tail phenomenon is called *excess kurtosis*. Time series that exhibit a fat tail distribution are often called *leptokurtic*. The red (dashed) line in the following figure illustrates excess kurtosis. The blue (solid) line shows a Gaussian distribution.



In addition, financial time series usually exhibit a characteristic called *volatility clustering*. In volatility clustering, large changes tend to follow large changes, and small changes tend to follow small changes. In either case, the changes from one period to the next are typically of unpredictable sign. 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 way, large shocks of either sign can persist and influence volatility forecasts for several periods.

Volatility clustering, or *persistence*, suggests a time-series model in which successive disturbances are uncorrelated, yet serially dependent. The following figure illustrates this characteristic. It shows the daily returns of the New York Stock Exchange Composite Index, as described in "NYSE" on page 1-13.



Volatility clustering (a type of heteroscedasticity) accounts for some but not all of the fat tail effect (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 happen to have fat tails. An example of such a distribution is Student's t.

Finally, certain classes of asymmetric GARCH models can also capture the *leverage effect*. This effect often results in observed asset returns being negatively correlated with changes in volatility. For certain asset classes, volatility tends to rise in response to lower than expected returns and to fall in response to higher than expected returns. These asset classes include equities, but exclude foreign exchange.

Such an effect suggests GARCH models that include an asymmetric response to positive and negative surprises.

Forecasting and Correlation of Financial Time Series

You can treat a financial time series as a sequence of random observations. This random sequence, or *stochastic process*, may exhibit a 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 the following components:

- A *deterministic component* (the forecast)
- A *random component* (the error, or uncertainty, associated with the forecast)

The following equation uses these components to represent a univariate model of an observed time series P_t:

$$y_t = f(t-1, X) + \varepsilon_t$$

In this equation,

- $f(t-1, X)$ represents the forecast, or deterministic component, of the current return as a function of information known at time $(t-1)$. This forecast includes the following:
  - Past innovations $\{\varepsilon_{t-1}, \varepsilon_{t-2}, \ldots\}$
  - Past observations $\{y_{t-1}, y_{t-2}, \ldots\}$
  - Any other relevant explanatory time-series data, X
- $\{\varepsilon_t\}$ is the random component. It represents the innovation in the mean of $\{y_t\}$. You can also interpret the random disturbance, or *shock*, $\{\varepsilon_t\}$, as the single-period-ahead forecast error.

Serial Dependence in Innovations

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

$$E\{\varepsilon_t\} = 0$$
$$E\{\varepsilon_t, \varepsilon_T\} = 0$$
$$t \neq T$$

Although successive innovations are uncorrelated, they are not independent. In fact, an explicit generating mechanism for a GARCH innovations process, $\{\varepsilon_t\}$, is

$$\varepsilon_t = \sigma_t z_t \quad \textbf{(2-1)}$$

where σ_t is the conditional standard deviation derived from one of the conditional variance equations shown in "Conditional Variance Models" on page 2-10.

z_t is a standardized, independent, identically distributed (i.i.d.) random draw from some specified probability distribution. The GARCH Toolbox™ software provides two distributions for modeling GARCH processes: Gaussian and Student's t.

Equation 2-1 illustrates that a GARCH innovations process $\{\varepsilon_t\}$ rescales an i.i.d process $\{z_t\}$ such that the conditional standard deviation incorporates the serial dependence of the conditional variance equation. Equivalently,Equation 2-1 also states that a standardized GARCH disturbance, ε_t / σ_t is itself an i.i.d. random variable $\{z\}$.

GARCH models are consistent with various forms of efficient market theory. These theories 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.

Conditional Mean and Variance Models

In this section...
"About Conditional Mean and Variance Models" on page 2-7
"Conditional Mean Models" on page 2-9
"Conditional Variance Models" on page 2-10

About Conditional Mean and Variance Models

GARCH literature often lacks consensus regarding the exact definition of any particular class of GARCH model. Software vendors, researchers, and references often disagree about the exact functional form and/or parameter constraints of almost all GARCH models. The following information may help reconcile some of these discrepancies.

- Although the functional form of a GARCH(P,Q) model, as described in Equation 2-4, is standard, alternative positivity constraints exist. However, these alternatives involve additional nonlinear inequalities that are difficult to impose in practice. They also do not affect the GARCH(1,1) model, which is by far the most common model. In contrast, the standard linear positivity constraints imposed by the GARCH Toolbox™ software are commonly used, and are straightforward to implement.
- Many references and software vendors refer to the GJR(P,Q) model, as described in Equation 2-5, as a TGARCH, or Threshold GARCH, model. However, others make a clear distinction between GJR(P,Q) and TGARCH(P,Q) models: a GJR(P,Q) model is a recursive equation for the conditional variance, and a TGARCH(P,Q) model is the identical recursive equation for the conditional standard deviation (see, for example, Hamilton [22] page 669, Bollerslev, et. al. [8] page 2970). Furthermore, additional discrepancies exist regarding whether to allow both negative and positive innovations to affect the conditional variance process. The GJR(P,Q) model included in the GARCH Toolbox software is relatively standard.
- The GARCH Toolbox software parameterizes GARCH(P,Q) and GJR(P,Q) models, as described in Equation 2-4 and Equation 2-5, including constraints, in a way that allows you to interpret a GJR(P,Q) model as an extension of a GARCH(P,Q) model. You can also interpret a GARCH(P,Q) model as a restricted GJR(P,Q) model with zero leverage terms. This latter

interpretation is useful for estimation and hypothesis testing via likelihood ratios.

- For GARCH(P,Q) and GJR(P,Q) models, the lag lengths P and Q, and the magnitudes of the coefficients G_i and A_j, determine the extent to which disturbances persist. These values then determine the minimum amount of presample data needed to initiate the simulation and estimation processes. The G_i terms capture persistence in EGARCH models.
- Although the functional form of an EGARCH(P,Q) model (Equation 2-6) is relatively standard, it is not the same as Nelson's original (see Nelson [28]). Many forms of EGARCH(P,Q) models exist. Another form is

$$\log \sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \log \sigma_{t-1}^2 + \sum_{j=1}^{Q} A_j \left[\frac{|\varepsilon_{t-j}| + L_j \varepsilon_{t-j}}{\sigma_{t-j}}\right]$$

  This EGARCH(P,Q) model form appears to offer an advantage. It does not explicitly make assumptions about the conditional probability distribution. That is, it does not assume that the distribution of $z_t = (\varepsilon_t/\sigma_t)$ is Gaussian or Student's t. However, this is not entirely true. Though the EGARCH(P,Q) model does not explicitly assume a distribution in this equation, such an assumption is required for forecasting and Monte Carlo simulation in the absence of user-specified presample data. In fact, you can easily rearrange this equation to highlight the probability distribution.

  The GARCH Toolbox software implements the form of the EGARCH(P,Q) model described by Equation 2-6 because this model closely resembles Nelson's original form.

- Although EGARCH(P,Q) models require no parameter constraints to ensure positive conditional variances, stationarity constraints are necessary. The GARCH Toolbox software treats EGARCH(P,Q) models as ARMA(P,Q) models for the logarithm of the conditional variance. Therefore, this toolbox imposes nonlinear constraints on the G_i coefficients to ensure that the eigenvalues of the characteristic polynomial are all inside the unit circle. (See, for example, page 2969 of Bollerslev, Engle, and Nelson [8], and page 12 of Bollerslev, Chou, and Kroner [7].)
- Consider the EGARCH(P,Q) and GJR(P,Q) models, as described in Equation 2-6 and Equation 2-5. These asymmetric models capture the leverage effect, or negative correlation, between asset returns and volatility. Both models include leverage terms that explicitly take into account the sign

and magnitude of the innovation noise term. Although both models are designed to capture the leverage effect, the way in which they do so differs.

For EGARCH(P,Q) models, the leverage coefficients L_i apply to the actual innovations ε_{t-1}. For GJR(P,Q) models, the leverage coefficients enter the model through a Boolean indicator, or dummy, variable. Therefore, if the leverage effect does indeed hold, the leverage coefficientsL_i should be negative for EGARCH(P,Q) models and positive for GJR(P,Q) models. This is in contrast to GARCH(P,Q) models, which ignore the sign of the innovation.

- Although GARCH(P,Q) and GJR(P,Q) models include terms related to the model innovations, $\varepsilon_t = \sigma_t z_t$, EGARCH(P,Q) models include terms related to the standardized innovations, $z_t = (\varepsilon_t/\sigma_t)$, such that z_t acts as the forcing variable for both the conditional variance and the error. In this respect, EGARCH(P,Q) models are fundamentally unique.
- Generally, there are no asymmetries in foreign-exchange rates. Therefore, asymmetric EGARCH(P,Q) and GJR(P,Q) conditional variance models are often inappropriate for modeling such return series.

Conditional Mean Models

This general ARMAX(R,M,Nx) model for the conditional mean

$$y_t = C + \sum_{i=1}^{R} \phi_i y_{t-1} + \varepsilon_t \sum_{j=1}^{M} \theta_j \varepsilon_{t-j} + \sum_{k=1}^{Nx} \beta_k x(t,k) \quad \textbf{(2-2)}$$

applies to all variance models with autoregressive coefficients $\{\Phi_i\}$, moving average coefficients $\{\Phi_j\}$, innovations $\{\varepsilon_t\}$, and returns $\{y_t\}$.

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

The eigenvalues $\{\lambda_i\}$ associated with the characteristic AR polynomial

$$\lambda^R - \phi_1 \lambda^{R-1} - \phi_2 \lambda^{R-2} - \ldots - \phi_R$$

must lie inside the unit circle to ensure stationarity. Similarly, the eigenvalues associated with the characteristic MA polynomial

$$\lambda^M + \phi_1 \lambda^{M-1} + \phi_2 \lambda^{M-2} + \ldots + \phi_M$$

must lie inside the unit circle to ensure invertibility.

Conditional Variance Models

The conditional variance of the innovations, σ_t^2, is by definition

$$Var_{t-1}(y_t) = E_{t-1}(\varepsilon_t^2) = \sigma_t^2 \quad \textbf{(2-3)}$$

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* applies more to long-term behavior of a time series, and assumes no explicit knowledge of the past.

The various GARCH models characterize the conditional distribution of ε_t by imposing alternative parameterizations to capture serial dependence on the conditional variance of the innovations. “About Conditional Mean and Variance Models” on page 2-7 further defines the conditional variance models.

GARCH(P,Q) Conditional Variance

The general GARCH(P,Q) model for the conditional variance of innovations is

$$\sigma_t^2 = \kappa + \sum_{i-1}^{P} G_i \sigma_{t-i}^2 + \sum_{j-1}^{Q} A_j \varepsilon_{t-j}^2 \quad \textbf{(2-4)}$$

with constraints

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

$\kappa > 0$

$G_i \geq 0 \; i = 1,2, \ldots, P$

$A_j \geq 0 \; j = 1,2, \ldots, Q$

The basic GARCH(P,Q) model is a symmetric variance process, in that it ignores the sign of the disturbance.

GJR(P,Q) Conditional Variance

The general GJR(P,Q) model for the conditional variance of the innovations with leverage terms is

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} A_j \varepsilon_{t-j}^2 + \sum_{j=1}^{Q} L_j S_{t-j} \varepsilon_{t-j}^2 \qquad \textbf{(2-5)}$$

where

$S_{t-j} = 1$ if $\varepsilon_{t-j} < 0$

$S_{t-j} = 0$ otherwise,

and

$$\sum_{i-1}^{P} G_i + \sum_{j-1}^{Q} A_j + \frac{1}{2} \sum_{j-1}^{Q} L_j < 1$$

$\kappa > 0$

$G_i \geq 0 \; i = 1,2, \ldots, P$

$A_j \geq 0 \; j = 1,2, \ldots, Q$

$A_j + L_j \geq 0 \; j = 1,2, \ldots, Q$

EGARCH(P,Q) Conditional Variance

The general EGARCH(P,Q) model for the conditional variance of the innovations, with leverage terms and an explicit probability distribution assumption, is

$$\log \sigma_t^2 = \kappa + \sum_{i=1}^{P} G_i \log \sigma_{t-1}^2 + \sum_{j=1}^{Q} A_j \left[\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} - E\left\{ \frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} \right\} \right] + \sum_{j=1}^{Q} L_j \left(\frac{\varepsilon_{t-j}}{\sigma_{t-j}} \right) \quad \textbf{(2-6)}$$

where

$$E\{|z_{t-j}|\} = E\left(\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right) = \sqrt{\frac{2}{\pi}}$$

for the Gaussian distribution, and

$$E\{|z_{t-j}|\} = E\left(\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right) = \sqrt{\frac{\nu - 2}{\pi}} \left(\frac{\Gamma\left(\frac{\nu - 1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \right)$$

for the Student's t distribution, with degrees of freedom $\nu > 2$.

The GARCH Toolbox software treats EGARCH(P,Q) models as ARMA(P,Q) models for $\log \sigma_t^2$. Thus, it includes the stationarity constraint for EGARCH(P,Q) models by ensuring that the eigenvalues of the characteristic polynomial

$$\lambda^P - G_1 \lambda^{P-1} - G_2 \lambda^{P-2} - \ldots - G_p$$

are inside the unit circle.

EGARCH models are fundamentally different from GARCH and GJR models in that the standardized innovation, z_t, serves as the forcing variable for both the conditional variance and the error. GARCH and GJR models allow for volatility clustering (persistence) via a combination of the G_i and A_j terms. The G_i terms capture persistence in EGARCH models.

The Default Model

The GARCH Toolbox™ default model is a simple constant mean model with GARCH(1,1) Gaussian innovations, based on Equation 2-2 and Equation 2-4.

$$y_t = C + \varepsilon_t \tag{2-7}$$

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + A_1\varepsilon_{t-1}^2 \tag{2-8}$$

Consider the conditional mean model, Equation 2-7. 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.

Consider the conditional variance model, Equation 2-8. 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 Equation 1-1 and Equation 1-2, 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 Equation 2-7 and Equation 2-8 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 [10], the fewer parameters to estimate, the less that can go wrong. Elaborate models often fail to offer real benefits when forecasting (see Hamilton [22], 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 [7], pages 10 and 22).

Primary Toolbox Functions

GARCH Toolbox™ software usage focuses on the following functions, which perform different tasks on GARCH models:

- `garchfit`, which you use for model estimation.
- `garchpred`, which you use for forecasting.
- `garchsim`, which you use for Monte Carlo simulation.
- `garchinfer`, which infers innovations and conditional standard deviations using inverse filtering. This function is related to `garchfit`, since both functions call the appropriate objective function.

These functions use a *GARCH specification structure* to share information about the specified model. The specification structure contains the model orders for the chosen conditional mean and variance models, and the parameters for those models.

Note All these functions accept a specification structure as input, but only `garchfit` can update the structure and provide it as an output. For more information, see Chapter 3, "GARCH Specification Structures" .

An analysis process using real-world data may involve calling these processing functions:

`garchfit` Estimates the model parameters. This function accepts a specification structure as an input. If you provide only the model orders for the chosen conditional mean and variance model, `garchfit` populates it with the coefficients resulting from the estimation process. If you also provide valid coefficients, `garchfit` uses them as initial estimates that the estimation process later refines. If you provide no specification structure, `garchfit` assumes the default model, as described in "The Default Model" on page 2-13.

In all cases, `garchfit` returns an updated specification structure, which encapsulates parameter estimates. This

	output structure is of the same form as the input structure. You can use it as an input for further modeling.
`garchpred`	Forecasts returns and conditional standard deviations. It accepts as input the specification structure provided by the `garchfit` estimation engine. You can also use `garchpred` to forecast volatility of asset returns over multiperiod holding intervals, or to forecast the standard errors of conditional mean forecasts.
`garchsim`	Simulates one or more sample paths for the return series, innovations, and conditional standard deviation processes, for the specified conditional mean and variance model. You can use these sample paths to perform Monte Carlo simulation of a given process.

For more details about these functions, see Chapter 12, “Function Reference”.

Example: Analysis and Estimation Using the Default Model

In this section...

"Pre-Estimation Analysis" on page 2-16

"Parameter Estimation" on page 2-24

"Post-Estimation Analysis" on page 2-27

Pre-Estimation Analysis

About This Example

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 before convergence. Or, it may converge to an unexpected, suboptimal solution.

You can avoid many of these difficulties by selecting the simplest model that adequately describes your data, and then performing a pre-fit analysis. The following pre-estimation analysis example shows how to:

- Plot the return series and examine the ACF and PACF.
- Perform preliminary tests, including Engle's ARCH test and the *Q*-test.

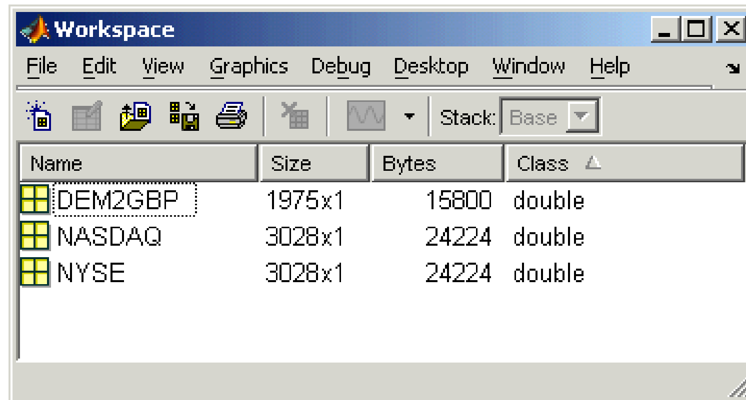
More specifically, the example does the following:

- Loads the data in the form of a price series.
- Converts the price series to a return series.
- Checks the return series for correlation.
- Checks for correlation in the squared returns.
- Quantifies the correlation.

Loading the Price Series Data

1 Load the MATLAB® binary file `garchdata.mat`, and view its contents in the Workspace Browser:

```
load garchdata
```



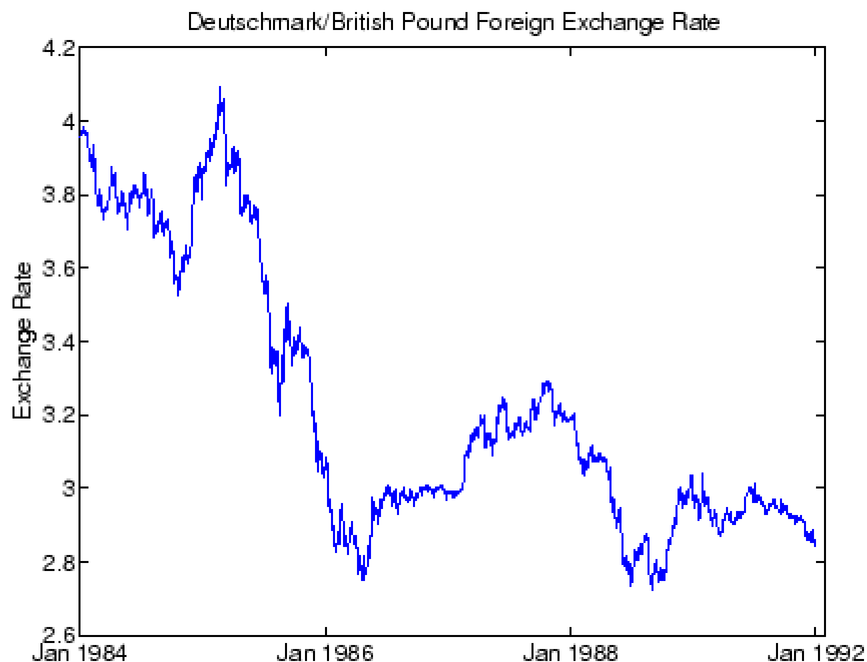
The data consists of three single-column vectors of different lengths, `DEM2GBP`, `NASDAQ`, and `NYSE`. Each vector is a separate price series for the named group.

2 Use the `whos` command to see all the variables in the current workspace:

```
whos

  Name          Size        Bytes    Class

  DEM2GBP     1975x1        15800    double array
  NASDAQ      3028x1        24224    double array
  NYSE        3028x1        24224    double array

Grand total is 8031 elements using 64248 bytes
```

3 `DEM2GBP` contains daily price observations of the Deutschemark/British Pound foreign-exchange rate. Use the MATLAB `plot` function to examine this data. Then, use the `set` function to set the position of and relabel the x-axis ticks of the current figure:

```
% plot([0:1974],DEM2GBP)
% set(gca,'XTick',[1 659 1318 1975])
% set(gca,'XTickLabel',{'Jan 1984' 'Jan 1986' 'Jan 1988' ...
%   'Jan 1992'})
%ylabel('Exchange Rate')
%title('Deutschmark/British Pound Foreign-Exchange Rate')
```



Converting the Prices to a Return Series

Because GARCH modeling assumes a return series, you need to convert the prices to returns.

1 Run the utility function `price2ret`:

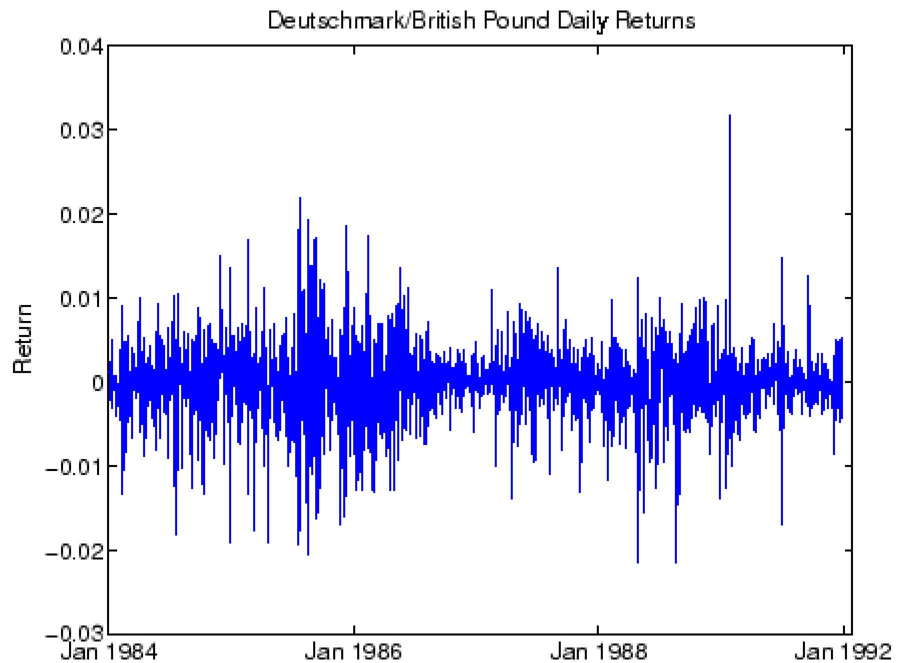
```
dem2gbp = price2ret(DEM2GBP);
```

Examine the result. The workspace information shows both the 1975-point price series and the 1974-point return series derived from it.

2 Now, use the `plot` function to see the return series:

```
plot(dem2gbp)
set(gca,'XTick',[1 659 1318 1975])
set(gca,'XTickLabel',{'Jan 1984' 'Jan 1986' 'Jan 1988' ...
    'Jan 1992'})
ylabel('Return')
title('Deutschmark/British Pound Daily Returns')
```

The raw return series shows volatility clustering.

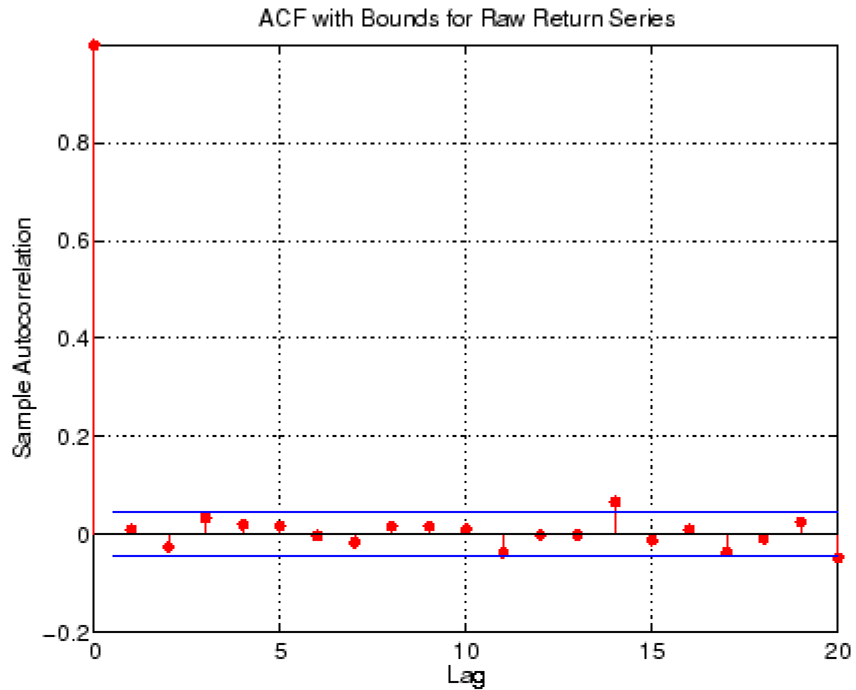


Checking for Correlation in the Return Series

Call the functions `autocorr` and `parcorr` to examine the sample autocorrelation (ACF) and partial-autocorrelation (PACF) functions, respectively.

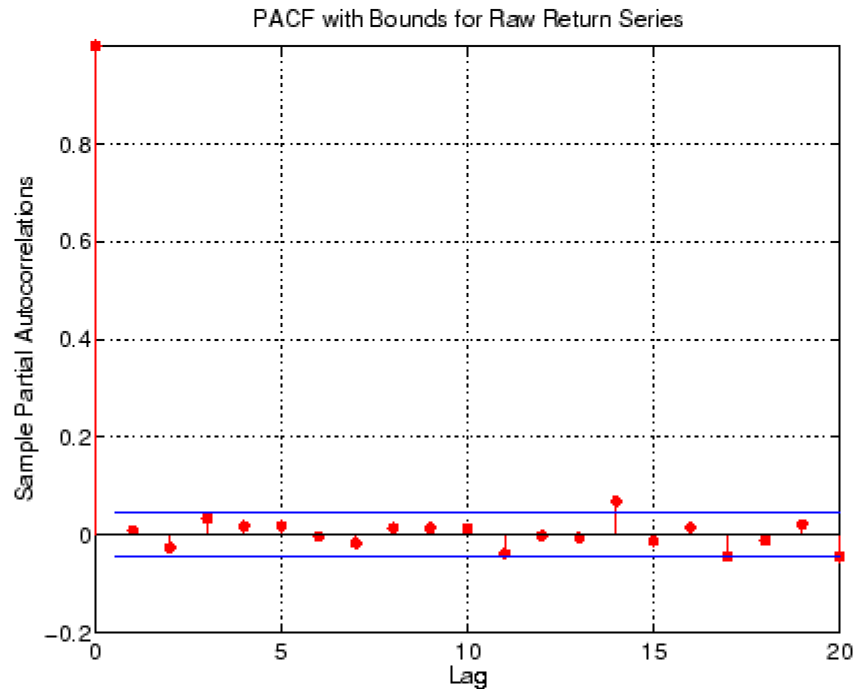
1 Assuming that all autocorrelations are zero beyond lag zero, use the `autocorr` function to compute and display the sample ACF of the returns and the upper and lower standard deviation confidence bounds:

```
autocorr(dem2gbp)
title('ACF with Bounds for Raw Return Series')
```



2 Use the `parcorr` function to display the sample PACF with upper and lower confidence bounds:

```
parcorr(dem2gbp)
title('PACF with Bounds for Raw Return Series')
```

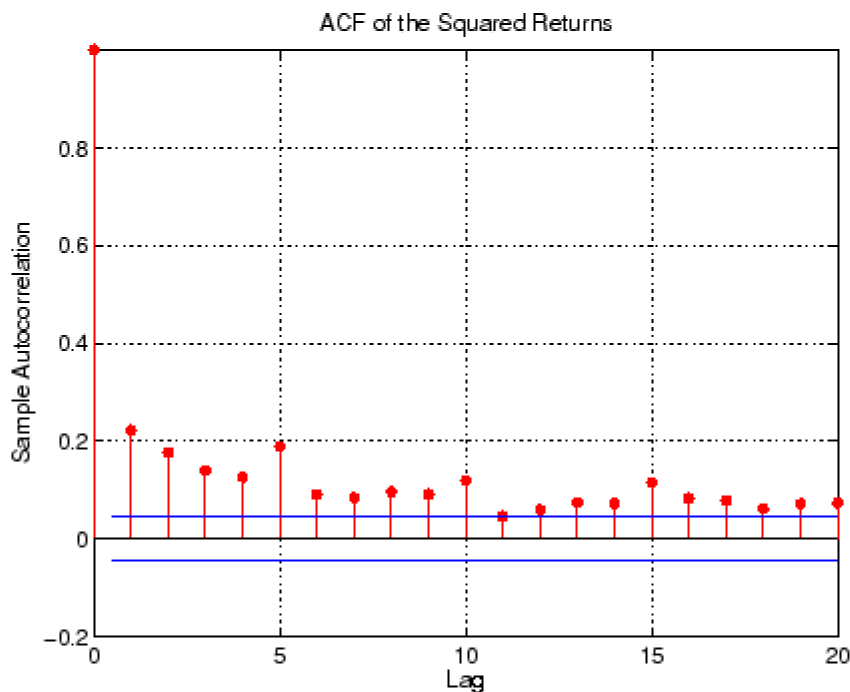


View the sample ACF and PACF with care (see Box, Jenkins, Reinsel [10], pages 34 and 186). The individual ACF values can have large variances and can also be autocorrelated. However, as preliminary identification tools, the ACF and PACF provide some indication of the broad correlation characteristics of the returns. From these figures for the ACF and PACF, there is little indication that you need to use any correlation structure in the conditional mean. Also, note the similarity between the graphs.

Checking for Correlation 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(dem2gbp.^2)
title('ACF of the Squared Returns')
```



This figure 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 2-13. The ACF shown in this figure appears to die out slowly, indicating the possibility of a variance process close to being nonstationary.

Quantifying the Correlation

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 `lbqtest` function 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 (residuals). In this case, however, you can also use it as part of the pre-fit analysis. This is because the default model assumes that returns are 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 [10], 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 (that is, no ARCH effects exist), this test statistic is also asymptotically Chi-Square distributed (see Engle [14], 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 (that is, do not reject the null hypothesis). `H = 1` means that significant correlation exists (that is, 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`).

1. Use `lbqtest` to verify (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(dem2gbp-mean(dem2gbp),[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]

ans =
         0    0.7278    6.9747   18.3070
         0    0.2109   19.0628   24.9958
         0    0.1131   27.8445   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((dem2gbp-mean(dem2gbp)).^2,[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]

ans =
    1.0000         0  392.9790   18.3070
    1.0000         0  452.8923   24.9958
    1.0000         0  507.5858   31.4104
```

2 Perform Engle's ARCH test using the function `archtest`:

```
[H,pValue,Stat,CriticalValue] = ...
      archtest(dem2gbp-mean(dem2gbp),[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]

ans =
    1.0000         0  192.3783   18.3070
    1.0000         0  201.4652   24.9958
    1.0000         0  203.3018   31.4104
```

This test also shows significant evidence in support of GARCH effects (heteroscedasticity). 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

This section continues the "Pre-Estimation Analysis" on page 2-16 example. It estimates model parameters, then examines the estimated GARCH model.

1 The presence of heteroscedasticity, 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 Default Model" on page 2-13. This only requires that you specify the return series of interest as an argument to the `garchfit` function:

```
[coeff,errors,LLF,innovations,sigmas,summary] = ...
   garchfit(dem2gbp);

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

Number of variables: 4

Functions
 Objective:                            garchllfn
 Gradient:                             finite-differencing
 Hessian:                              finite-differencing (or Quasi-Newton)
```

```
 Nonlinear constraints:                armanlc
 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:          4

Algorithm selected
   medium-scale

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

                                 max                    Directional   First-order
 Iter F-count        f(x)   constraint   Step-size   derivative   optimality
    1      28    -7916.01    -2.01e-006  7.63e-006           857    1.42e+005
    2      36    -7959.65   -1.508e-006       0.25           389     9.8e+007
    3      45    -7963.98   -3.113e-006      0.125           131    5.29e+006
    4      52    -7965.59   -1.586e-006        0.5          55.9    4.45e+007
    5      65     -7966.9   -1.574e-006    0.00781           101    1.46e+007
    6      74    -7969.46   -2.201e-006      0.125          14.9    2.77e+007
    7      83    -7973.56   -2.663e-006      0.125          36.6    1.45e+007
    8      90    -7982.09   -1.332e-006        0.5         -6.39    5.59e+006
    9     103    -7982.13   -1.399e-006    0.00781          6.49    1.32e+006
   10     111    -7982.53   -1.049e-006       0.25          12.5    1.87e+007
   11     120    -7982.56   -1.186e-006      0.125          3.72     3.8e+006
   12     128    -7983.69    -1.11e-006       0.25         0.184    4.91e+006
   13     134    -7983.91   -7.813e-007          1         0.732    1.22e+006
   14     140    -7983.98   -9.265e-007          1         0.186    1.17e+006
   15     146       -7984   -8.723e-007          1        0.0427    9.52e+005
   16     154       -7984   -8.775e-007       0.25        0.0152    6.33e+005
   17     160       -7984    -8.75e-007          1       0.00197    6.98e+005
   18     166       -7984   -8.763e-007          1      0.000931    7.38e+005
   19     173       -7984   -8.759e-007        0.5      0.000469    7.37e+005
   20     179       -7984   -8.761e-007          1       0.00012    7.22e+005
```

```
21    199      -7984  -8.761e-007  -6.1e-005       0.0167    7.37e+005
22    213      -7984  -8.761e-007    0.00391      0.00582    7.26e+005
 Optimization terminated successfully:
Search direction less than 2*options.TolX and
 maximum constraint violation is less than options.TolCon
No Active Constraints
```

The default value of the `Display` parameter in the specification structure is `'on'`. As a result, `garchfit` prints diagnostic optimization and summary information to the command window in the following example. (For information about the `Display` parameter, see the Optimization Toolbox™ `fmincon` function.)

2 Once you complete the estimation, display the parameter estimates and their standard errors using the `garchdisp` function:

```
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
  Number of Parameters Estimated: 4

                                 Standard          T
  Parameter        Value           Error       Statistic
 -----------   -----------   ------------   -----------
          C    -6.1919e-005   8.4331e-005     -0.7342
          K    1.0761e-006    1.323e-007       8.1341
   GARCH(1)    0.80598        0.016561        48.6685
    ARCH(1)    0.15313        0.013974        10.9586
```

If you substitute these estimates in the definition of the default model, Equation 2-7 and Equation 2-8, the estimation process implies that the constant conditional mean/GARCH(1,1) conditional variance model that best fits the observed data is

$$y_t = -6.1919e^{-005} + \varepsilon_t$$
$$\sigma_t^2 = 1.0761e^{-006} + 0.80598\sigma_{t-1}^2 + 0.15313\varepsilon_{t-1}^2$$

where G_1 = *GARCH(1)* = `0.80598` and A_1 = *ARCH(1)* = `0.15313`. In addition, C = C = `-6.1919e-005` and κ = K = `1.0761e-006`.

Post-Estimation Analysis

The post-estimation analysis example continues the "Pre-Estimation Analysis" on page 2-16 and "Parameter Estimation" on page 2-24 examples.

This example does the following:

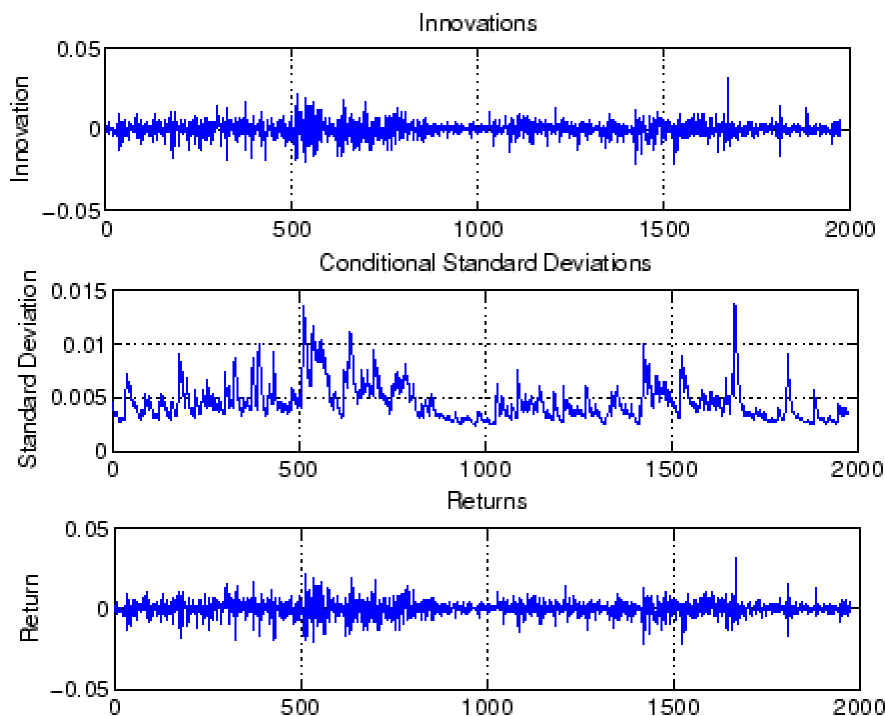
- Compares the residuals, conditional standard deviations, and returns.
- Uses plots and quantitative techniques to compare correlation of the standardized innovations.
- Quantifies and compares correlation of the standardized innovations.

Comparing 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 (`sigmas`).

Use the `garchplot` function to inspect the relationship between the innovations (residuals) derived from the fitted model, the corresponding conditional standard deviations, and the observed returns.

```
garchplot(innovations,sigmas,dem2gbp)
```



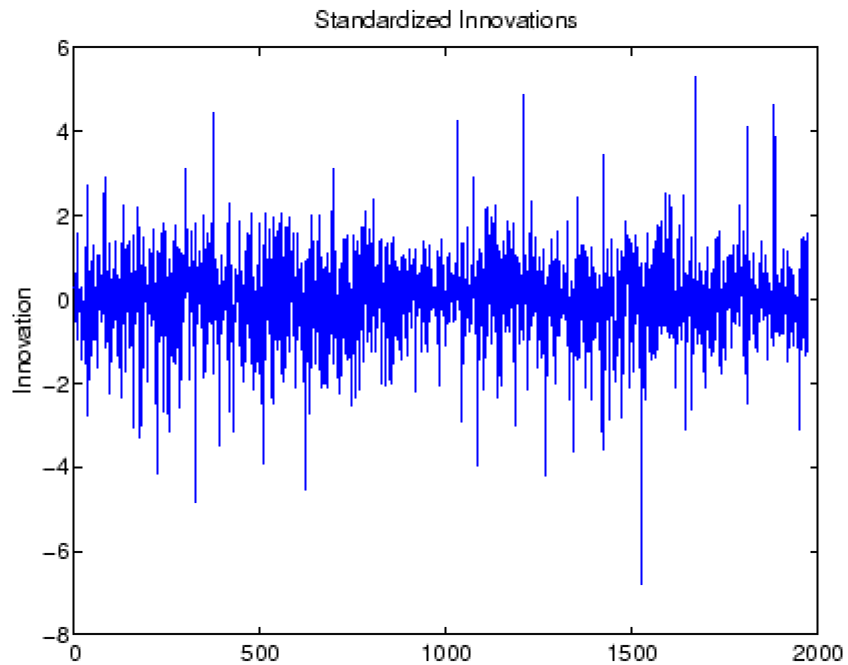
Both the innovations (shown in the top plot) and the returns (shown in the bottom plot) exhibit volatility clustering. Also, the sum, $G_1 + A_1 = 0.80598 + 0.15313 = 0.95911$, is close to the integrated, nonstationary boundary given by the constraints associated with Equation 2-4.

Comparing Correlation of the Standardized Innovations

The figure in "Comparing the Residuals, Conditional Standard Deviations, and Returns" on page 2-27 shows that the fitted innovations exhibit volatility clustering.

1 Plot the standardized innovations (the innovations divided by their conditional standard deviation):

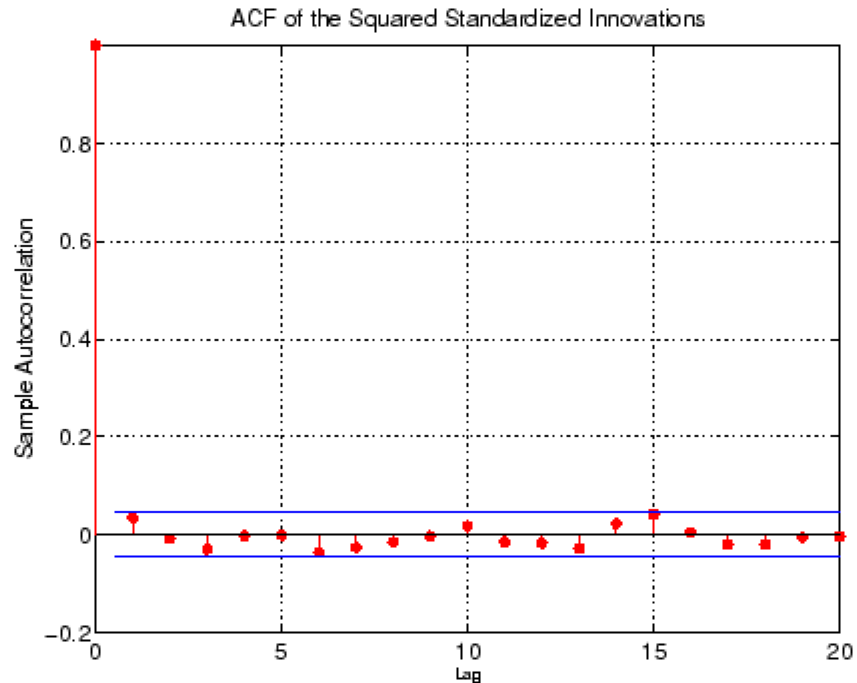
```
plot(innovations./sigmas)
ylabel('Innovation')
title('Standardized Innovations')
```



The standardized innovations appear generally stable with little clustering.

2 Plot the ACF of the squared standardized innovations:

```
autocorr((innovations./sigmas).^2)
title('ACF of the Squared Standardized Innovations')
```



The standardized innovations also show no correlation. Now compare the ACF of the squared standardized innovations in this figure to the ACF of the squared returns before fitting the default model. (See "Pre-Estimation Analysis" on page 2-16.) The comparison shows that the default model sufficiently explains the heteroscedasticity in the raw returns.

Quantifying and Comparing Correlation of the Standardized Innovations

Compare the results of the Q-test and the ARCH test with the results of these same tests in "Pre-Estimation Analysis" on page 2-16:

```
[H, pValue,Stat,CriticalValue] = ...
    lbqtest((innovations./sigmas).^2,[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]

ans =
         0    0.5262     9.0626   18.3070
         0    0.3769    16.0777   24.9958
```

```
        0    0.6198   17.5072   31.4104

[H, pValue, Stat, CriticalValue] = ...
    archtest(innovations./sigmas,[10 15 20]',0.05);
[H  pValue  Stat  CriticalValue]

ans =
        0    0.5625    8.6823   18.3070
        0    0.4408   15.1478   24.9958
        0    0.6943   16.3557   31.4104
```

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

GARCH Specification Structures

Introduction (p. 3-2) — How the primary analysis and modeling functions operate on the GARCH specification structure

Associating Model Equation Variables with Corresponding Parameters in Specification Structures (p. 3-4) — Associates variables used in the model equations ("Conditional Mean and Variance Models" on page 2-7) with their corresponding parameters in the specification structure

Example: Interpreting Specification Structures (p. 3-6) — Examples of how to interpret the contents of specification structures

Working with Specification Structures (p. 3-9) — Creating, modifying, and retrieving values from specification structures

Introduction

The GARCH Toolbox™ software maintains the parameters that define a model and control the estimation process in a *specification structure*.

The `garchfit` function creates the specification structure for the default model (see “The Default Model” on page 2-13), and stores the model orders and estimated parameters in it. For more complex models, use the `garchset` function to explicitly specify, in a specification structure:

- The conditional variance model
- The mean and variance model orders
- (Optionally) The initial coefficient estimates

The primary analysis and modeling functions, `garchfit`, `garchpred`, and `garchsim`, all operate on the specification structure. The following table describes how each function uses the specification structure.

For more information about specification structure parameters, see the `garchset` function reference page.

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, GJR, or EGARCH form.	• **Input.** Optionally accepts a GARCH specification structure as input. If the structure contains the model orders (`R`, `M`, `P`, `Q`) but no coefficient vectors (`C`, `AR`, `MA`, `Regress`, `K`, `ARCH`, `GARCH`, `Leverage`), `garchfit` uses maximum likelihood to estimate the coefficients for the specified mean and variance models. If the structure contains coefficient vectors, `garchfit` uses them as initial estimates for further refinement. If you do not provide a specification structure,

Function	Description	Use of GARCH Specification Structure
		garchfit 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 forecasts the conditional mean and standard deviation. • **Output.** The garchpred function does not modify or return the specification structure.
garchsim	Uses Monte Carlo methods to simulate 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 simulates sample paths. • **Output.** garchsim does not modify or return the specification structure.

Associating Model Equation Variables with Corresponding Parameters in Specification Structures

In this section...
"About Specification Structure Parameter Names" on page 3-4
"Conditional Mean Model" on page 3-4
"Conditional Variance Models" on page 3-5

About Specification Structure Parameter Names

The names of specification structure parameters that define the ARMAX and GARCH models usually reflect the variable names of their corresponding components in the conditional mean and variance model equations described in "Conditional Mean and Variance Models" on page 2-7.

Conditional Mean Model

In the conditional mean model:

- `R` and `M` represent the order of the ARMA(R,M) conditional mean model.
- `C` represents the constant *C*.
- `AR` represents the `R`-element autoregressive coefficient vector Φ_i.
- `MA` represents the `M`-element moving average coefficient vector Θ_j.
- `Regress` represents the regression coefficients β_k.

Unlike the other components of the conditional mean equation, the GARCH specification structure does not include *X*. *X* is an optional matrix of returns that some GARCH Toolbox™ functions use as explanatory variables in the regression component of the conditional mean. For example,*y* could contain return series of a market index collected over the same period as the return series *X*. Toolbox functions that require a regression matrix provide a separate argument you can use to specify it.

Conditional Variance Models

In conditional variance models:

- `P` and `Q` represent the order of the GARCH(P,Q), GJR(P,Q), or EGARCH(P,Q) conditional variance model.
- `K` represents the constant κ.
- `GARCH` represents the `P`-element coefficient vector G_i.
- `ARCH` represents the `Q`-element coefficient vector A_j.
- `Leverage` represents the `Q`-element leverage coefficient vector, L_j, for asymmetric EGARCH(P,Q) and GJR(P,Q) models.

Example: Interpreting Specification Structures

1 Display the fields of the `coeff` specification structure, for the estimated default model from “Example: Analysis and Estimation Using the Default Model” on page 2-16:

```
coeff
coeff =
          Comment: 'Mean: ARMAX(0,0,0); Variance: GARCH(1,1)'
     Distribution: 'Gaussian'
                C: -6.1919e-005
    VarianceModel: 'GARCH'
                P: 1
                Q: 1
                K: 1.0761e-006
            GARCH: 0.8060
             ARCH: 0.1531
```

The terms to the left of the colon (`:`) denote parameter names.

When you display a specification structure, only the fields that are applicable to the specified model appear. For example, `R = M = 0` in this model, so these fields do not appear.

By default, `garchset` and `garchfit` automatically generate the `Comment` field. This field summarizes the ARMAX and GARCH models used for the conditional mean and variance equations. You can use `garchset` to set the value of the `Comment` field, but the value you give it replaces the summary statement.

2 Display the MA(1)/GJR(1,1) estimated model from “Specifying Presample Data” on page 6-21:

```
coeff =
          Comment: 'Mean: ARMAX(0,1,0); Variance: GJR(1,1)'
     Distribution: 'Gaussian'
                M: 1
                C: 5.6403e-004
               MA: 0.2501
    VarianceModel: 'GJR'
                P: 1
```

```
               Q: 1
               K: 1.1907e-005
           GARCH: 0.6945
            ARCH: 0.0249
        Leverage: 0.2454
         Display: 'off'
```

`length(MA) = M`, `length(GARCH) = P`, and `length(ARCH) = Q`.

3 Consider what you would see if you had created the specification structure for the same MA(1)/GJR(1,1) example, but had not yet estimated the model coefficients. The specification structure would appear as follows:

```
spec = garchset('VarianceModel','GJR','M',1,'P',1,'Q',1,...
       'Display','off')
spec = 
          Comment: 'Mean: ARMAX(0,1,?); Variance: GJR(1,1)'
     Distribution: 'Gaussian'
                M: 1
                C: []
               MA: []
    VarianceModel: 'GJR'
                P: 1
                Q: 1
                K: []
            GARCH: []
             ARCH: []
         Leverage: []
          Display: 'off'
```

The empty matrix symbols, `[]`, indicate that the specified model requires these fields, but that you have not yet assigned them values. For the specification to be *complete*, you must assign valid values to these fields.

You can use `garchset` to assign values, for example, as initial parameter estimates, to these fields. You can also pass such a specification structure to `garchfit`, which uses the parameters it estimates to complete the model specification. You cannot pass such a structure to `garchsim`, `garchinfer`, or `garchpred`. These functions require complete specifications.

For descriptions of all the specification structure fields, see the `garchset` function reference page.

Working with Specification Structures

In this section...

"Creating Specification Structures" on page 3-9

"Modifying Specification Structures" on page 3-11

"Retrieving Specification Structure Values" on page 3-12

Creating Specification Structures

In general, you must use the `garchset` function to create a specification structure that contains at least the chosen variance model and the mean and variance model orders. The only exception is the default model, for which `garchfit` can create a specification structure. The model parameters you provide must specify a valid model.

When you create a specification structure, you can specify both the conditional mean and variance models. Alternatively, you can specify either the conditional mean or the conditional variance model. If you do not specify both models, `garchset` assigns default parameters to the one that you did not specify.

For the conditional mean, the default is a constant ARMA(0,0,?) model. For the conditional variance, the default is a constant GARCH(0,0) model. The question mark (?) indicates that `garchset` cannot interpret whether you intend to include a regression component (see Chapter 8, "Regression Components").

The following examples create specification structures and display the results. You need only enter the leading characters that uniquely identify the parameter. As illustrated here, `garchset` parameter names are case insensitive.

For the Default Model

The following is a sampling of statements that all create specification structures for the default model.

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

```
spec = garchset('p',1,'Q',1);
spec = garchset;
```

The output of each of these commands is the same. The `Comment` field summarizes the model. Because `R = M = 0`, the fields `R`, `M`, `AR`, and `MA` do not appear.

```
spec =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(1,1)'
    Distribution: 'Gaussian'
               C: []
   VarianceModel: 'GARCH'
               P: 1
               Q: 1
               K: []
           GARCH: []
            ARCH: []
```

For ARMA(0,0)/GJR(1,1)

`garchset` accepts the constant default for the mean model.

```
spec = garchset('VarianceModel','GJR','P',1,'Q',1)

spec =
         Comment: 'Mean: ARMAX(0,0,?); Variance: GJR(1,1)'
    Distribution: 'Gaussian'
               C: []
   VarianceModel: 'GJR'
               P: 1
               Q: 1
               K: []
           GARCH: []
            ARCH: []
        Leverage: []
```

For AR(2)/GARCH(1,2) with Initial Parameter Estimates

garchset infers the model orders from the lengths of the coefficient vectors, assuming a GARCH(P,Q) conditional variance process as the default:

```
spec = garchset('C',0,'AR',[0.5 -0.8],'K',0.0002,...
                'GARCH',0.8,'ARCH',[0.1 0.05])
spec =

          Comment: 'Mean: ARMAX(2,0,?); Variance: GARCH(1,2)'
     Distribution: 'Gaussian'
                R: 2
                C: 0
               AR: [0.5000 -0.8000]
    VarianceModel: 'GARCH'
                P: 1
                Q: 2
                K: 2.0000e-004
            GARCH: 0.8000
             ARCH: [0.1000 0.0500]
```

Modifying Specification Structures

The following example creates an initial structure, and then updates the existing structure with additional parameter/value pairs. At each step, the result must be a valid specification structure:

```
spec = garchset('VarianceModel','EGARCH','M',1,'P',1,'Q',1);
spec = garchset(spec,'R',1,'Distribution','T')

spec =
          Comment: 'Mean: ARMAX(1,1,?); Variance: EGARCH(1,1)'
     Distribution: 'T'
              DoF: []
                R: 1
                M: 1
                C: []
               AR: []
               MA: []
    VarianceModel: 'EGARCH'
                P: 1
                Q: 1
```

```
              K: []
          GARCH: []
           ARCH: []
       Leverage: []
```

Retrieving Specification Structure Values

This example does the following:

1. Creates a specification structure, `spec`, by providing the model coefficients.

2. Uses the `garchset` function to infer the model orders from the lengths of specified model coefficients, assuming the GARCH(P,Q) default variance model.

3. Uses `garchget` to retrieve the variance model and the model orders for the conditional mean.

   You need only type the leading characters that uniquely identify the parameter.

```
spec = garchset('C',0,'AR',[0.5 -0.8],'K',0.0002,...
                'GARCH',0.8,'ARCH',[0.1 0.05])
spec =

          Comment: 'Mean: ARMAX(2,0,?); Variance: GARCH(1,2)'
     Distribution: 'Gaussian'
                R: 2
                C: 0
               AR: [0.5000 -0.8000]
    VarianceModel: 'GARCH'
                P: 1
                Q: 2
                K: 2.0000e-004
            GARCH: 0.8000
             ARCH: [0.1000 0.0500]
R = garchget(spec,'R')
R =
      2

M = garchget(spec,'m')
```

```
M =
      0

var = garchget(spec,'VarianceModel')
var =
      GARCH
```

Simulation of GARCH Models

Simulating Single and Multiple Paths (p. 4-2)	How to simulate single and multiple paths for return series, innovations, and conditional standard deviation processes
Working with Presample Data (p. 4-7)	How to use automatically generated and user-supplied presample data

Simulating Single and Multiple Paths

In this section...

"Introduction" on page 4-2
"Preparing the Example Data" on page 4-2
"Simulating Single Paths" on page 4-3
"Simulating Multiple Paths" on page 4-5

Introduction

Given models for the conditional mean and variance, as described in "Conditional Mean and Variance Models" on page 2-7, the `garchsim` function can simulate one or more sample paths for return series, innovations, and conditional standard deviation processes.

The section "Example: Analysis and Estimation Using the Default Model" on page 2-16 uses the default GARCH(1,1) model to model the Deutschmark/British pound foreign-exchange series. These examples use the resulting model

$$y_t = -6.1919e^{-005} + \varepsilon_t$$

$$\sigma_t^2 = 1.0761e^{-006} + 0.80598\sigma_{t-1}^2 + 0.15313\varepsilon_{t-1}^2$$

to simulate sample paths for return series, innovations, and conditional standard deviation processes.

Preparing the Example Data

Restore your workspace as needed. Due to space constraints, this example shows only part of the output of the estimation:

```
load garchdata
dem2gbp = price2ret(DEM2GBP);
[coeff,errors,LLF,innovations,sigmas] = garchfit(dem2gbp);
coeff
```

```
coeff =
          Comment: 'Mean: ARMAX(0,0,0); Variance: GARCH(1,1)'
     Distribution: 'Gaussian'
                C: -6.1919e-005
    VarianceModel: 'GARCH'
                P: 1
                Q: 1
                K: 1.0761e-006
            GARCH: 0.8060
             ARCH: 0.1531
```

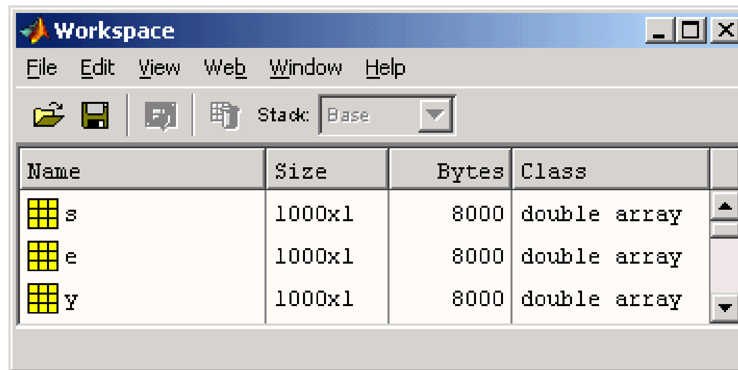
Simulating Single Paths

1 Generate a single path of 1000 observations starting from the initial MATLAB® random number generator state. Assuming 250 trading days per year, this represents roughly fours years' worth of daily data:

```
randn('state',0);
rand('twister',0);
[e,s,y] = garchsim(coeff,1000);
```

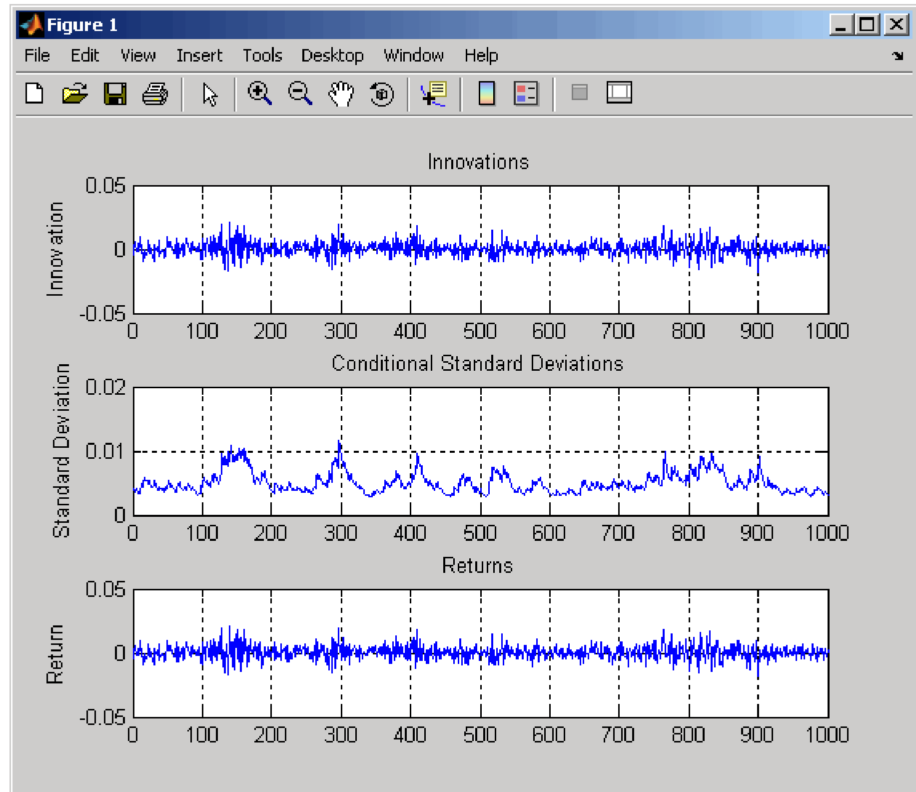
Tip For information about how to generate `coeff` for use in this example, see "Introduction" on page 4-2.

The result is a single realization of 1000 observations each for the innovations $\{\varepsilon_t\}$, conditional standard deviations $\{\sigma_t\}$, and returns $\{y_t\}$ processes. The output variables `e`, `s`, and `y` represent these processes.



2 Plot the `garchsim` output data.

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



Note If you do not specify the number of observations, the default is 100. For example, the command

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

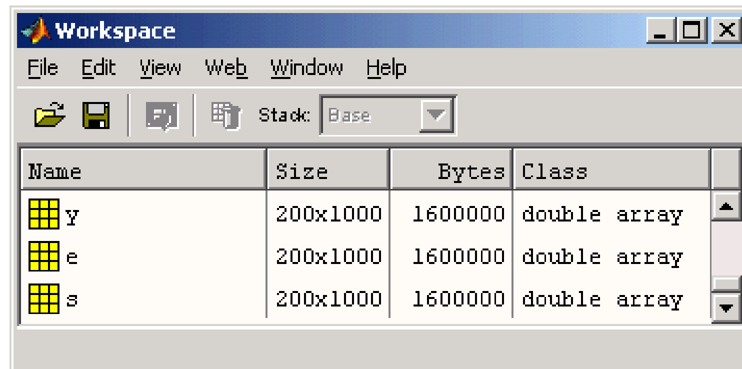
produces a single path of 100 observations.

Simulating Multiple Paths

In some cases, you may need multiple realizations. Use the same model as in “Simulating Single Paths” on page 4-3 to simulate 1000 paths of 200 observations each:

```
randn('state',0);
rand('twister',0);
[e,s,y] = garchsim(coeff,200,1000);
```

The $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$ processes are 200-by-1000 element matrices. These arrays that require large amounts of memory. Because of the way the GARCH Toolbox™ software manages transients, simulating this data requires more memory than the 4800000 bytes indicated in the Workspace Browser.



For more information about transients, see “Automatically Generating Presample Data” on page 4-7.

Working with Presample Data

In this section...

"About Presample Data" on page 4-7

"Automatically Generating Presample Data" on page 4-7

"Running Simulations With User-Specified Presample Data" on page 4-13

About Presample Data

Because the mean equation and the variance equations can be recursive in nature, they require initial, or presample, data to initiate the simulation. This section explains the use of automatically generated and user-supplied presample data. It also discusses response tolerance and the minimization of transient effects for automatically generated presample data.

Automatically Generating Presample Data

When you allow `garchsim` to automatically generate required initial data:

- `garchsim` performs *independent path simulation*. All simulated realizations are unique in that they evolve independently and share no common presample conditioning data.
- `garchsim` generates the presample data in a way that minimizes transient effects in the output processes.

Automatically Minimizing Transient Effects

`garchsim` generates output processes in a (approximately) steady state by attempting to eliminate transients in the data it simulates. It first estimates the number of observations needed for the transients to decay to some arbitrarily small value, subject to a 10000-observation maximum. It then generates a number of observations equal to the sum of this estimated value and the number of observations you request. `garchsim` then ignores the earlier estimated number of initial observations needed for the transients to decay sufficiently, and returns only the requested number of later observations.

To do this, `garchsim` interprets a GARCH(P,Q) or GJR(P,Q) conditional variance process as an ARMA(max(P,Q),P) model for the squared innovations.

It also interprets an EGARCH(P,Q) process as an ARMA(P,Q) model for the log of the conditional variance. (See, for example, Bollerslev [6], p. 310.) It then interprets the ARMA model as the correlated output of a linear filter and estimates its impulse response. It does so by finding the magnitude of the largest eigenvalue of its autoregressive polynomial. Based on this eigenvalue, `garchsim` estimates the number of observations (subject to a maximum of 10000) needed for the magnitude of the impulse response (which begins at 1) to decay below the default response tolerance 0.01 (1 percent). If the conditional mean has an ARMA(R,M) component, then `garchsim` also estimates the number of observations needed for the impulse response to decay below the response tolerance. This number is also subject to a maximum of 10000.

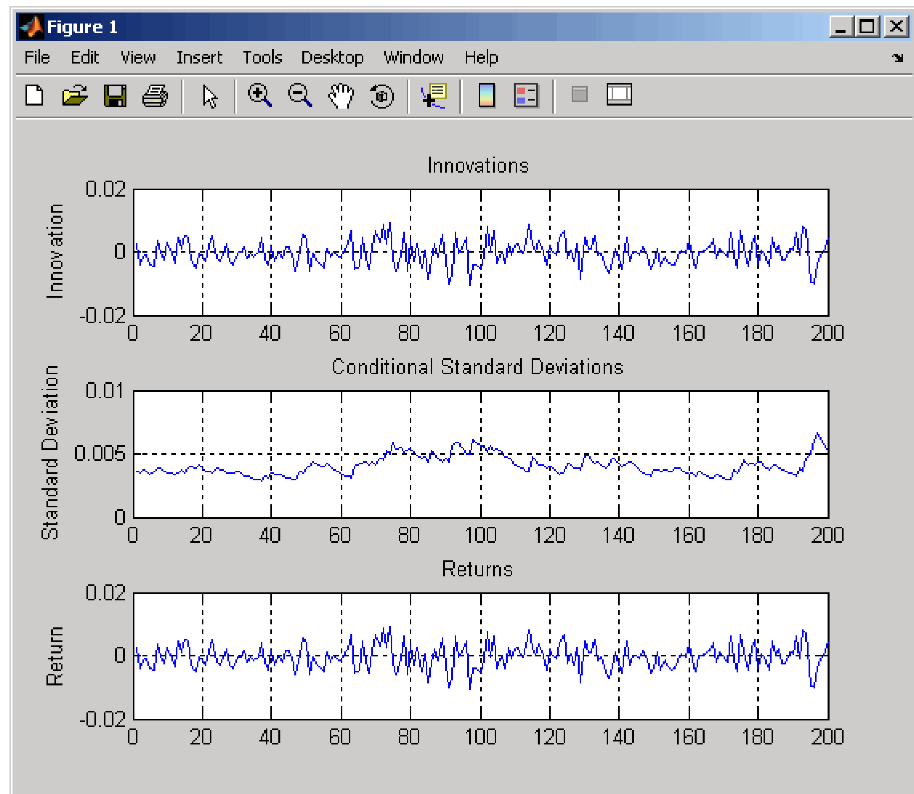
The effect of transients in the simulation process parallels that in the estimation, or inference, process. "Presample Data and Transient Effects" on page 6-24 provides an example of transient effects in the estimation process.

Specifying a Scalar Response Tolerance

This example compares simulated observations generated using the default response tolerance, `0.01`, and a larger tolerance, `0.05`, using the model from "Simulating Single and Multiple Paths" on page 4-2.

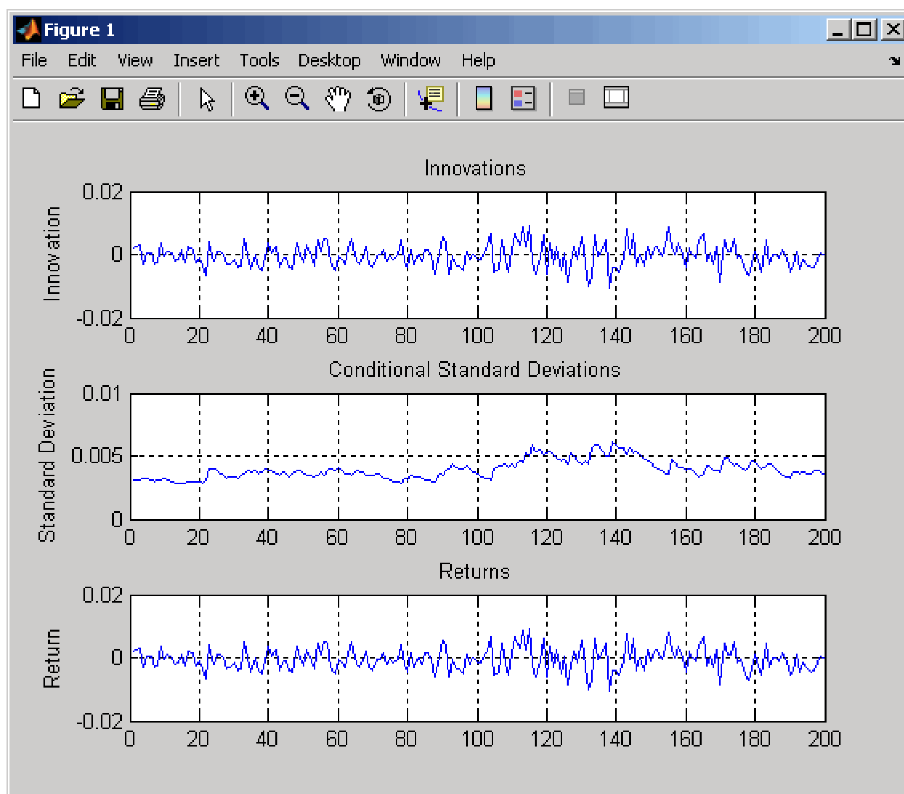
1 Simulate a single path of 200 observations, using the default tolerance `0.01`, and set the scalar integer random generator state to its initial state `0`:

```
randn('state',0);
rand('twister',0);
[e1,s1,y1] = garchsim(coeff,200,1);
garchplot(e1,s1,y1)
```



2 Now repeat the simulation, specifying the scalar `Tolerance` argument as `0.05`:

```
randn('state',0);
rand('twister',0);
[e5,s5,y5] = garchsim(coeff,200,1,[],[],0.05);
garchplot(e5,s5,y5)
```



The observations generated using the `0.05` response tolerance are the same as those generated for the default `0.01` tolerance, but shifted to the right. This is because fewer observations are required for the magnitude of the impulse response to decay below the larger `0.05` tolerance.

If `Tolerance` is smaller than `0.01`, `garchsim` might have to generate more observations. This could cause it to reach the 10000 observation transient decay period maximum, or run out of memory.

Storage Considerations

Depending on the values of the parameters in the simulated conditional mean and variance models, you may need long presample periods for the transients to die out. Although the simulation outputs relatively small matrices, the initial computation of these transients can consume large amounts of memory,

leading to performance degradation. Because of this, `garchsim` imposes a maximum of 10000 observations to the transient decay period of each realization. The example in "Simulating Multiple Paths" on page 4-5, which simulates three 200-by-1000 element arrays, requires intermediate storage for many more than 200 observations.

Other Ways to Minimize Transient Effects

If you suspect that transients persist in the simulated data `garchsim` returns, use one of the following methods to minimize their effect:

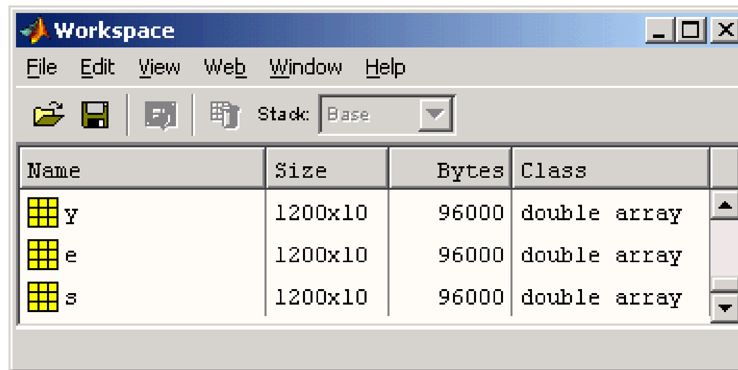
- "Oversampling" on page 4-11
- "Recycling Outputs" on page 4-13

Oversampling. Generate samples that are larger than you need, and delete observations from the beginning of each output series. For example, suppose you simulate 10 independent paths of 1000 observations each for $\{\varepsilon_t\}$, $\{\sigma_t\}$, and $\{y_t\}$, starting from a known scalar random number state (12345).

1. Generate 1200 observations:

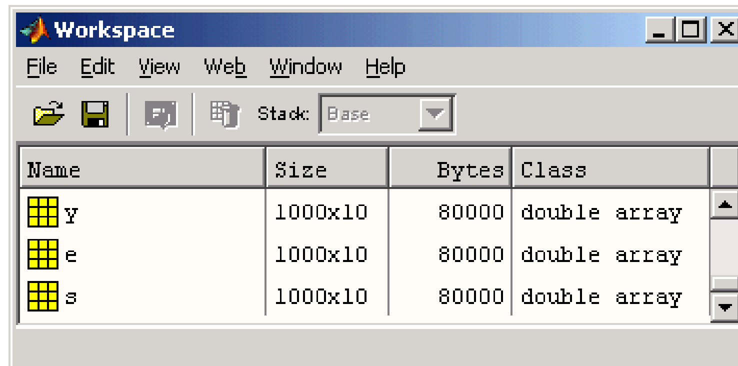
```
randn('state',12345);
rand('twister',12345);
[e,s,y] =  garchsim(coeff,1200,10);
```

   `garchsim` generates sufficient presample data so that it can ignore initial samples that might be affected by transients. It then returns only the requested 1200 later observations.



2 Further minimize the effect of transients by retaining only the last 1000 observations of interest:

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



Note This example also illustrates how to specify a scalar random number generator state. This use corresponds to the `rand` and `randn` syntaxes, `rand('state',j)` and `randn('state',j)`.

Recycling Outputs. Simulate the desired number of observations without explicitly providing presample data; that is, use `garchsim` to automatically generate the presample data. Then run the simulation again, using the simulated observations as the presample data. Repeat this process until you have sufficiently eliminated transient effects. For information about supplying presample data, see "Running Simulations With User-Specified Presample Data" on page 4-13.

Running Simulations With User-Specified Presample Data

To explicitly specify all required presample data, use the following time-series input arrays:

- `PreInnovations`, which is associated with the `Innovations` `garchsim` output
- `PreSigmas`, which is associated with the `Sigmas` `garchsim` output
- `PreSeries`, which is associated with the `Series` `garchsim` output

When specified, `garchsim` uses these presample arrays to initiate the filtering process and form the conditioning set upon which the simulated realizations are based.

The `PreInnovations`, `PreSigmas`, and `PreSeries` arrays and their associated outputs are column-oriented. Each column of each array is associated with a distinct *realization*, or *sample path*. The first row of each array stores the oldest data, and the last row stores the most recent data.

You can specify these input arguments as matrices (with multiple columns), or as single-column vectors. The following table summarizes the minimum number of rows required to successfully initiate the simulation process.

Garchsim Input Argument	Minimum Number of Rows GARCH(P,Q), GJR(P,Q)	EGARCH(P,Q)
`PreInnovations`	`max(M,Q)`	`max(M,Q)`

Garchsim Input Argument	Minimum Number of Rows GARCH(P,Q), GJR(P,Q)	EGARCH(P,Q)
`PreSigmas`	`P`	`max(P,Q)`
`PreSeries`	`R`	`R`

If you specify these input arguments as matrices, `garchsim` uses each column to initiate simulation of the corresponding column of the `Innovations`, `Sigmas`, and `Series` outputs. Each of the presample inputs must have `NUMPATHS` columns.

If you specify these input arguments as column vectors, and `NUMPATHS` is greater than `1`, `garchsim` performs *dependent path simulation*. In this case, `garchsim` applies the same vector to each column of the corresponding `Innovations`, `Sigmas`, and `Series` outputs. All simulated sample paths share a common conditioning set. Although all realizations evolve independently, they share common presample conditioning data. Dependent path simulation enables the simulated sample paths to evolve from a common starting point, and allows Monte Carlo simulation of forecasts and forecast error distributions. See Chapter 11, “Example Workflow: Estimation, Forecasting, and Simulation”.

If you specify at least one, but fewer than three, sets of presample data, `garchsim` does not attempt to derive presample observations for those you omit. When specifying your own presample data, include all required data for the given conditional mean and variance models. See the example “Specifying Presample Data” on page 6-21.

Note You can also use the `garchsim` input argument `State` to specify your own standardized noise process.

5

Monte Carlo Simulation of Stochastic Differential Equations

Introduction (p. 5-2)	High-level overview of tasks you can perform with stochastic differential equations
Terminology (p. 5-3)	Explanations of terms used in the following sections
Behavior and Syntax of SDE Objects (p. 5-5)	Common characteristics and behavior of SDE objects and utilities
Parametric Specification (p. 5-7)	Specifying parameters that support relationships commonly found in SDE simulation
Using SDE Objects to Create Models (p. 5-11)	How to instantiate objects from the SDE class hierarchy to represent models
Solving Problems with SDE Models (p. 5-29)	Examples of creating SDE objects to model and solve problems
Creating User-Specified Functions (p. 5-69)	Creating your own functions
Managing Memory, Performance, and Solution Accuracy (p. 5-72)	Information to optimize memory, performance, and accuracy of solutions

Introduction

The GARCH Toolbox™ software enables you to model dependent financial and economic variables, such as interest rates and equity prices, by performing Monte Carlo simulation of stochastic differential equations (SDEs). The flexible architecture of the SDE engine provides efficient simulation methods that allow you to create new simulation and derivative pricing methods.

Tasks you can perform using this functionality include:

- Running vectorized methods to simulate static univariate models.
- Simulating static, separable Geometric Brownian Motion and Brownian Motion multivariate models.
- Performing path-dependent analysis using end-of-period processing and state vector adjustments.
- Sampling SDEs at intermediate times without reporting those times, improving accuracy and reducing memory consumption.
- Approximating analytic solutions for separable geometric Brownian motion and Hull-White/Vasicek models.
- Reducing variance using antithetic sampling.

The SDE architecture also supports the following features:

- Euler approximation default simulation method
- Stochastic interpolation and Brownian bridge simulation methods
- Support for any combination of static and dynamic model parameters
- Support for state and Brownian vectors of arbitrary dimensionality
- Optional user-specified random number generation and dependence/correlation structure
- The ability to avoid storing state and noise time series to improve performance and memory efficiency

Terminology

In this section...
"Trials vs. Paths" on page 5-3
"NTRIALS, NPERIODS, and NSTEPS" on page 5-4

Trials vs. Paths

Monte Carlo simulation literature often uses different terminology for the evolution of the simulated variables of interest, such as *trials*, *paths*, *realizations*, or *replications*. The following sections use the terms *trial* and *path* interchangeably.

However, there are situations where you should distinguish between these terms. Specifically, the term *trial* often implies the result of an independent random experiment (for example, the evolution of the price of a single stock or portfolio of stocks). Such an experiment computes the average or expected value of a variable of interest (for example, the price of a derivative security) and its associated confidence interval.

By contrast, the term *path* implies the result of a random experiment that is different or unique from other results, but that may or may not be independent.

The distinction between these terms is usually unimportant. It may, however, be useful when applied to *variance reduction* techniques that attempt to increase the efficiency of Monte Carlo simulation by inducing dependence across sample paths. A classic example involves pair-wise dependence induced by *antithetic sampling*, and applies to more sophisticated variance reduction techniques, such as *stratified sampling*.

For more information, about antithetic sampling, see "Antithetic Sampling" on page 15-83. For more information about stratified sampling, see "User-Specified Random Number Generation: Stratified Sampling" on page 5-63.

NTRIALS, NPERIODS, and NSTEPS

SDE methods in the GARCH Toolbox™ software use the parameters `NTRIALS`, `NPERIODS`, and `NSTEPS` as follows:

- The input argument `NTRIALS` specifies the number of simulated trials or sample paths to generate. This argument always determines the size of the third dimension (the number of pages) of the output 3-dimensional time-series array `Paths`. Indeed, in a traditional Monte Carlo simulation of one or more variables, each sample path is independent and represents an independent trial.
- The parameters `NPERIODS` and `NSTEPS` represent the number of simulation periods and time steps, respectively. Both periods and time steps are related to time increments that determine the exact sequence of observed sample times. The distinction between these terms applies only to issues of accuracy and memory management. For more information, see "Optimizing Accuracy of Solutions" on page 5-74 and "Managing Memory" on page 5-72.

Behavior and Syntax of SDE Objects

In this section...

"Relationship Between SDE Models and Objects" on page 5-5
"Displaying Objects" on page 5-5
"Assigning and Referencing Object Parameters" on page 5-6
"Constructing and Evaluating Models" on page 5-6

Relationship Between SDE Models and Objects

Most models and utilities available with Monte Carlo Simulation of SDEs are represented as MATLAB® objects. Therefore, this documentation often uses the terms *model* and *object* interchangeably.

However, although all models are represented as objects, not all objects represent models. In particular, drift and diffusion objects are used in model specification, but neither of these types of objects in and of themselves makes up a complete model. In most cases, you do not need to create drift and diffusion objects directly, so you do not need to differentiate between objects and models. It is important, however, to understand the distinction between these terms.

In many of the following examples, most model parameters are evaluated or invoked like any MATLAB function. Although it is helpful to examine and access model parameters as you would data structures, think of these parameters as *functions* that perform *actions*.

For more information about MATLAB objects, see "Using Objects to Write Data to a File" in the MATLAB documentation.

Displaying Objects

- Objects display like traditional MATLAB data structures.
- Displayed object parameters appear as nouns that begin with capital letters. In contrast, parameters such as `simulate` and `interpolate` appear as verbs that begin with lowercase letters, which indicate tasks to perform.

Assigning and Referencing Object Parameters

- Objects support referencing similar to data structures. For example, statements like the following are generally valid:

```
A = obj.A
```

- Objects support complete parameter assignment similar to data structures. For example, statements like the following are generally valid:

```
obj.A = 3
```

- Objects do not support partial parameter assignment as data structures do. Therefore, statements like the following are generally invalid:

```
obj.A(i,j) = 0.3
```

Constructing and Evaluating Models

- You can construct objects of any model class only if enough information is available to determine unambiguously the dimensionality of the model. Because various class constructors offer unique input interfaces, some models require additional information to resolve model dimensionality.
- You need only enter required input parameters in placeholder format, where a given input argument is associated with a specific position in an argument list. You can enter optional inputs in any order as parameter name-value pairs, where the name of a given parameter appears in single quotation marks and precedes its corresponding value.
- Association of dynamic (time-variable) behavior with function evaluation, where *time* and *state* (t,X_t) are passed to a common, published interface, is pervasive throughout the SDE class system. You can use this function evaluation approach to model or construct powerful analytics. For a simple example, see "Example: Creating Univariate GBM Models" on page 5-24.

Parametric Specification

In this section...

"General Parametric Specification" on page 5-7

"General SDEs" on page 5-7

"Drift and Diffusion Specifications" on page 5-8

General Parametric Specification

The SDE engine allows the simulation of generalized multivariate stochastic processes, and provides a flexible and powerful simulation architecture. The framework also provides you with utilities and model classes that offer a variety of parametric specifications and interfaces. The architecture is fully multidimensional in both the state vector and the Brownian motion, and offers both linear and mean-reverting drift-rate specifications.

You can specify most parameters as MATLAB® arrays or as functions accessible by a common interface, that support general dynamic/nonlinear relationships common in SDE simulation. Specifically, you can simulate correlated paths of any number of state variables driven by a vector-valued Brownian motion of arbitrary dimensionality. This simulation approximates the underlying multivariate continuous-time process using a vector-valued stochastic difference equation.

General SDEs

Consider the following general stochastic differential equation:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \tag{5-1}$$

where:

- X is an *NVARS*-by-1 state vector of process variables (for example, short rates or equity prices) to simulate.
- W is an *NBROWNS*-by-1 Brownian motion vector.
- F is an *NVARS*-by-1 vector-valued drift-rate function.

- G is an *NVARS*-by-*NBROWNS* matrix-valued diffusion-rate function.

The drift and diffusion rates, F and G, respectively, are general functions of a real-valued scalar sample time t and state vector X_t. Also, static (non-time-variable) coefficients are simply a special case of the more general dynamic (time-variable) situation, just as a function can be a trivial constant; for example, $f(t,X_t) = 4$. The SDE in Equation 5-1 is useful in implementing derived classes that impose additional structure on the drift and diffusion-rate functions.

Drift and Diffusion Specifications

For example, an SDE with a linear drift rate has the form:

$$F(t, X_t) = A(t) + B(t)X_t \quad \text{(5-2)}$$

where A is an *NVARS*-by-1 vector-valued function and B is an *NVARS*-by-*NVARS* matrix-valued function.

As an alternative, consider a drift-rate specification expressed in mean-reverting form:

$$F(t, X_t) = S(t)[L(t) - X_t] \quad \text{(5-3)}$$

where S is an *NVARS*-by-*NVARS* matrix-valued function of mean reversion speeds (that is, rates of mean reversion), and L is an *NVARS*-by-1 vector-valued function of mean reversion levels (that is, long run average level).

Similarly, consider the following diffusion-rate specification:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t) \quad \text{(5-4)}$$

where D is an *NVARS*-by-*NVARS* diagonal matrix-valued function. Each diagonal element of D is the corresponding element of the state vector raised to the corresponding element of an exponent *Alpha*, which is also an *NVARS*-by-1 vector-valued function. V is an *NVARS*-by-*NBROWNS* matrix-valued function of instantaneous volatility rates. Each row of V corresponds to a particular state variable, and each column corresponds to

a particular Brownian source of uncertainty. V associates the exposure of state variables with sources of risk.

The parametric specifications for the drift and diffusion-rate functions associate parametric restrictions with familiar models derived from the general SDE class, and provide coverage for many popular models.

As discussed in the following sections, the class system and hierarchy of the SDE engine use industry-standard technology to provide simplified interfaces for many models by placing user-transparent restrictions on drift and diffusion specifications. This design allows you to mix and match existing models, and customize drift or diffusion-rate functions.

For example, the following popular models are simply special cases of the general SDE model.

Popular Models

Model Name	Specification
Brownian Motion (BM)	$dX_t = A(t)dt + V(t)dW_t$
Geometric Brownian Motion (GBM)	$dX_t = B(t)X_t dt + V(t)X_t dW_t$
Constant Elasticity of Variance (CEV)	$dX_t = B(t)X_t dt + V(t)X_t^{\alpha(t)} dW_t$

Popular Models (Continued)

Model Name	Specification
Cox-Ingersoll-Ross (CIR)	$dX_t = S(t)(L(t) - X_t)dt + V(t)X_t^{\frac{1}{2}}dW_t$
Hull-White/Vasicek (HWV)	$dX_t = S(t)(L(t) - X_t)dt + V(t)dW_t$

Using SDE Objects to Create Models

In this section...

"SDE Classes" on page 5-11

"Creating Base SDE Objects" on page 5-14

"Creating Drift and Diffusion Objects" on page 5-16

"Creating Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO)" on page 5-19

"Creating Stochastic Differential Equations from Linear Drift (SDELD)" on page 5-20

"Creating Brownian Motion (BM) Models" on page 5-21

"Creating Constant Elasticity of Variance (CEV) Models" on page 5-22

"Creating Geometric Brownian Motion (GBM) Models" on page 5-23

"Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD)" on page 5-24

"Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models" on page 5-25

"Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 5-26

SDE Classes

The SDE Class Hierarchy

The GARCH Toolbox™ SDE class structure represents a generalization and specialization hierarchy. The top-level class provides the most general model interface and offers the default Monte Carlo simulation and interpolation methods. In turn, derived classes offer restricted interfaces that simplify model creation and manipulation while providing detail regarding model structure.

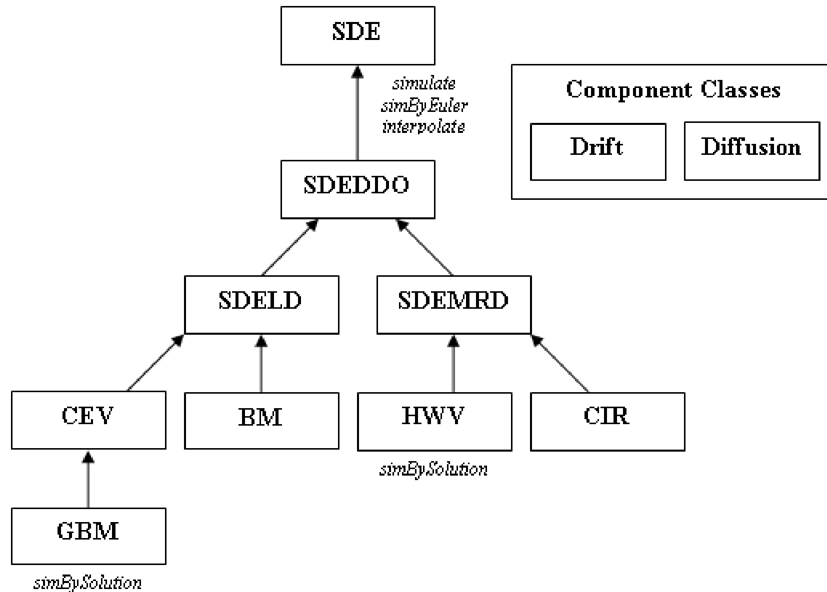
The following table lists the SDE classes. The introductory examples in the accompanying sections show how to use these classes to create objects associated with univariate models. Although the GARCH Toolbox SDE engine

supports multivariate models, univariate models facilitate object creation and display, and allow you to easily associate inputs with object parameters.

SDE Classes

Class Name	For More Information, See ...
`SDE`	"Creating Base SDE Objects" on page 5-14
`Drift`, `Diffusion`	"Creating Drift and Diffusion Objects" on page 5-16
`SDEDDO`	"Creating Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO)" on page 5-19
`SDELD`	"Creating Stochastic Differential Equations from Linear Drift (SDELD)" on page 5-20
`CEV`	"Creating Constant Elasticity of Variance (CEV) Models" on page 5-22
`BM`	"Creating Brownian Motion (BM) Models" on page 5-21
`SDEMRD`	"Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD)" on page 5-24
`GBM`	"Creating Geometric Brownian Motion (GBM) Models" on page 5-23
`HWV`	"Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 5-26
`CIR`	"Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models" on page 5-25

The following figure illustrates the inheritance relationships among SDE classes.



SDE Methods

The `SDE` class provides default simulation and interpolation methods for all derived classes:

- `simulate`: High-level wrapper around the user-specified simulation method stored in the `Simulation` method
- `simByEuler`: Default Euler approximation simulation method
- `interpolate`: Stochastic interpolation method (that is, Brownian bridge)

The `HWV` and `GBM` classes feature an additional method, `simBySolution`, that simulates approximate solutions of diagonal-drift processes.

For more information, see Method Reference.

SDE Class Constructors

You use *class constructors* to create SDE objects. The following sections include examples of how to do this.

For more information, see "Stochastic Differential Equation (SDE) Class Constructors" on page 14-3.

Creating Base SDE Objects

About Base SDE Models

The base `SDE` class:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

represents the most general model.

Tip The `SDE` class is not an abstract class. You can instantiate `SDE` objects directly to extend the set of core models.

Constructing an `SDE` object requires two inputs:

- A drift-rate function `F`. `F` returns an `NVARS`-by-1 drift-rate vector when run with the following inputs:
  - A real-valued scalar observation time t.
  - An `NVARS`-by-1 state vector X_t.
- A diffusion-rate function `G`. `G` returns an `NVARS`-by-`NBROWNS` diffusion-rate matrix when run with the inputs t and X_t.

Evaluating object parameters by passing (t, X_t) to a common, published interface allows most parameters to be referenced by a common input argument list that reinforces common method programming. You can use this simple function evaluation approach to model or construct powerful analytics, as in the following example.

Example: Creating Base SDE Models

Construct an `SDE` object `obj` to represent a univariate geometric Brownian Motion model of the form:

$$dX_t = 0.1X_t dt + 0.3X_t dW_t \quad \textbf{(5-5)}$$

1 Create drift and diffusion functions that are accessible by the common (t,X_t) interface:

```
F = @(t,X) 0.1 * X;
G = @(t,X) 0.3 * X;
```

2 Pass the functions to the `SDE` constructor to create an object `obj` of class `SDE`:

```
obj = sde(F, G)    % dX = F(t,X)dt + G(t,X)dW
obj =
   Class SDE: Stochastic Differential Equation
   -------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

`obj` displays like a MATLAB® structure, with the following information:

- The object's class
- A brief description of the object
- A summary of the dimensionality of the model

The object's displayed parameters are as follows:

- `StartTime`: The initial observation time (real-valued scalar)
- `StartState`: The initial state vector (`NVARS`-by-1 column vector)
- `Correlation`: The correlation structure between Brownian process
- `Drift`: The drift-rate function $F(t,X_t)$
- `Diffusion`: The diffusion-rate function $G(t,X_t)$
- `Simulation`: The simulation method or function.

Of these displayed parameters, only `Drift` and `Diffusion` are required inputs.

The only exception to the (t, X_t) evaluation interface is `Correlation`. Specifically, when you enter `Correlation` as a function, the SDE engine assumes that it is a deterministic function of time, *C(t)*. This restriction on `Correlation` as a deterministic function of time allows Cholesky factors to be computed and stored before the formal simulation. This inconsistency dramatically improves run-time performance for dynamic correlation structures. If `Correlation` is stochastic, you can also include it within the simulation architecture as part of a more general random number generation function.

Specifying Object Parameters and Simulation Inputs

The class-naming conventions introduced here, discussed in more detail in subsequent sections, are meant to be meaningful. When you specify object parameters or simulation inputs as functions, the object assumes no knowledge of implementation details. A given function is required only to evaluate properly when you pass time and state to it.

Creating Drift and Diffusion Objects

About Drift and Diffusion Objects

Because base-level SDE objects accept drift and diffusion objects in lieu of functions accessible by (t, X_t), you can create `SDE` objects with combinations of customized drift or diffusion functions and objects. The `drift` and `diffusion` rate classes encapsulate the details of input parameters to optimize run-time efficiency for any given combination of classes.

Although `drift` and `diffusion` objects differ in the details of their representation, they are identical in their basic implementation and interface. They look, feel like, and are evaluated as functions:

- The `drift` class allows you to create drift-rate objects of the form:

  $$F(t, X_t) = A(t) + B(t)X_t$$

  where:

- A is an NVARS-by-1 vector-valued function accessible using the (t, X_t) interface.
- B is an NVARS-by-NVARS matrix-valued function accessible using the (t, X_t) interface.

- Similarly, the diffusion class allows you to create diffusion-rate objects:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- D is an NVARS-by-NVARS diagonal matrix-valued function.
- Each diagonal element of D is the corresponding element of the state vector raised to the corresponding element of an exponent Alpha, which is an NVARS-by-1 vector-valued function.
- V is an NVARS-by-NBROWNS matrix-valued volatility rate function Sigma.
- Alpha and Sigma are also accessible using the (t, X_t) interface.

Note You can express drift and diffusion classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components A and B as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: Creating Drift and Diffusion Rate Objects as Model Inputs

In this example, you create drift and diffusion rate objects to create the same model as in "Example: Creating Base SDE Models" on page 5-14.

Create a drift-rate function F and a diffusion-rate function G:

```
F = drift(0, 0.1)      % Drift rate function F(t,X)
G = diffusion(1, 0.3)  % Diffusion rate function G(t,X)
F =
   Class DRIFT: Drift Rate Specification
   -------------------------------------
      Rate: drift rate function F(t,X(t))
```

```
          A: 0
          B: 0.1
G =
   Class DIFFUSION: Diffusion Rate Specification
   ---------------------------------------------
       Rate: diffusion rate function G(t,X(t))
      Alpha: 1
      Sigma: 0.3
```

Each object displays like a MATLAB structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because `drift` and `diffusion` classes create model components rather than models. Neither `F` nor `G` contains enough information to characterize the dimensionality of a problem.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

Again, `Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Creating Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO)

About SDEDDO Models

The SDEDDO class derives from the base SDE class. To use this class, you must pass drift and diffusion-rate objects to the SDEDDO constructor.

Example: Creating SDEDDO Models

1 Create drift and diffusion rate objects:

```
F = drift(0, 0.1);      % Drift rate function F(t,X)
G = diffusion(1, 0.3);  % Diffusion rate function G(t,X)
```

2 Pass these objects to the SDEDDO constructor:

```
obj = sdeddo(F, G)      % dX = F(t,X)dt + G(t,X)dW
obj =
   Class SDEDDO: SDE from Drift and Diffusion Objects
   --------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   --------------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 0
              B: 0.1
          Alpha: 1
          Sigma: 0.3
```

In this example, the object displays the additional parameters associated with input drift and diffusion objects.

Creating Stochastic Differential Equations from Linear Drift (SDELD)

About SDELD Models

The SDELD class derives from the SDEDDO class. These objects allow you to simulate correlated paths of NVARS state variables expressed in linear drift-rate form:

$$dX_t = (A(t) + B(t)X_t)dt + D(t, X_t^{\alpha(t)})V(t)dW_t \quad \textbf{(5-6)}$$

SDELD objects provide a parametric alternative to the mean-reverting drift form, as discussed in "Example: Creating SDEMRD Models" on page 5-25. They also provide an alternative interface to the SDEDDO parent class, because you can create an object without first having to create its drift and diffusion-rate components.

Example: Creating SDELD Models

Create the same model as in "Example: Creating Base SDE Models" on page 5-14:

```
obj = sdeld(0, 0.1, 1, 0.3) % (A, B, Alpha, Sigma)
obj =
   Class SDELD: SDE with Linear Drift
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 0
              B: 0.1
          Alpha: 1
          Sigma: 0.3
```

Creating Brownian Motion (BM) Models

About BM Models

The Brownian Motion (BM) model derives directly from the linear drift (SDELD) class:

$$dX_t = \mu(t)dt + V(t)dW_t \tag{5-7}$$

Example: Creating BM Models

Create a univariate Brownian motion (BM) object to represent the model:

$$dX_t = 0.3dW_t$$

```
obj = bm(0, 0.3) % (A = Mu, Sigma)
obj =
   Class BM: Brownian Motion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 0
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
             Mu: 0
          Sigma: 0.3
```

BM objects display the parameter A as the more familiar Mu.

The BM class also provides an overloaded Euler simulation method that improves run-time performance in certain common situations. Use the specialized method only if all of the following conditions are met:

- The expected drift, or trend, rate Mu is a column vector.
- The volatility rate, Sigma, is a matrix.
- No end-of-period adjustments and/or processes are made.

- If specified, the random noise process Z is a 3-dimensional array.
- If Z is unspecified, the assumed Gaussian correlation structure is a double matrix.

Creating Constant Elasticity of Variance (CEV) Models

About CEV Models

The Constant Elasticity of Variance (CEV) model also derives directly from the linear drift (SDELD) class:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t \quad (5\text{-}8)$$

The CEV class constrains A to an NVARS-by-1 vector of zeros. D is an unrestricted diagonal matrix whose elements are the corresponding element of the state vector X, raised to an exponent *alpha*.

Example: Creating Univariate CEV Models

Create a univariate CEV object to represent the model:

$$dX_t = 0.25X_t + 0.3X_t^{\frac{1}{2}}dW_t$$

```
obj = cev(0.25, 0.5, 0.3) % (B = Return, Alpha, Sigma)
obj =
   Class CEV: Constant Elasticity of Variance
   ------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.25
          Alpha: 0.5
          Sigma: 0.3
```

`CEV` and `GBM` objects display the parameter `B` as the more familiar `Return`.

Creating Geometric Brownian Motion (GBM) Models

About GBM Models

The Geometric Brownian Motion (`GBM`) model derives directly from the `CEV` model:

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t \quad \textbf{(5-9)}$$

Compared to `CEV`, `GBM` constrains all elements of the *alpha* exponent vector to one such that D is now a diagonal matrix with the state vector X along the main diagonal.

The `GBM` class also provides two simulation methods that can be used by separable models:

- An overloaded Euler simulation method that improves run-time performance in certain common situations. You can use this specialized method only if all of the following conditions are true:
  - The expected rate of return (`Return`) is a diagonal matrix.
  - The volatility rate (`Sigma`) is a matrix.
  - No end-of-period adjustments/processes are made.
  - If specified, the random noise process `Z` is a 3-dimensional array.
  - If `Z` is unspecified, the assumed Gaussian correlation structure is a double matrix.

- An approximate analytic solution (`simBySolution`) obtained by applying an Euler approach to the transformed (using Ito's formula) logarithmic process. In general, this is *not* the exact solution to this `GBM` model, as the probability distributions of the simulated and true state vectors are identical *only* for piece-wise constant parameters. If the model parameters are piece-wise constant over each observation period, the state vector

X_t is log-normally distributed and the simulated process is exact for the observation times at which X_t is sampled.

Example: Creating Univariate GBM Models

Create a univariate GBM object to represent the model:

$$dX_t = 0.25X_t dt + 0.3X_t dW_t$$

```
obj = gbm(0.25, 0.3)  % (B = Return, Sigma)
obj =
   Class GBM: Generalized Geometric Brownian Motion
   ------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.25
          Sigma: 0.3
```

Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD)

About SDEMRD Models

The SDEMRD class derives directly from the SDEDDO class. It provides an interface in which the drift-rate function is expressed in mean-reverting drift form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\alpha(t)})V(t)dW_t \tag{5-10}$$

SDEMRD objects provide a parametric alternative to the linear drift form by reparameterizing the general linear drift such that:

$$A(t) = S(t)L(t), B(t) = -S(t)$$

Example: Creating SDEMRD Models

Create an `SDEMRD` object `obj` with a square root exponent to represent the model:

$$dX_t = 0.2(0.1 - X_t)dt + 0.05X_t^{\frac{1}{2}}dW_t$$

```
obj = sdemrd(0.2, 0.1, 0.5, 0.05)  % (Speed, Level, Alpha, Sigma)
obj =
   Class SDEMRD: SDE with Mean-Reverting Drift
   -------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Alpha: 0.5
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

`SDEMRD` objects display the familiar `Speed` and `Level` parameters instead of `A` and `B`.

Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models

About CIR Models

The Cox-Ingersoll-Ross (`CIR`) short rate class derives directly from SDE with mean-reverting drift (`SDEMRD`):

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\frac{1}{2}})V(t)dW_t \qquad \textbf{(5-11)}$$

where D is a diagonal matrix whose elements are the square root of the corresponding element of the state vector.

Example: Creating CIR Models

Create a `CIR` object to represent the same model as in "Example: Creating SDEMRD Models" on page 5-25:

```
obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
obj =
   Class CIR: Cox-Ingersoll-Ross
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

Although the last two objects are of different classes, they represent the same mathematical model. They differ in that you create the `CIR` object by specifying only three input arguments. This distinction is reinforced by the fact that the `Alpha` parameter does not display – it is defined to be `1/2`.

Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models

About HWV Models

The Hull-White/Vasicek (`HWV`) short rate class derives directly from SDE with mean-reverting drift (`SDEMRD`):

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t \qquad \textbf{(5-12)}$$

Example: Creating HWV Models

Using the same parameters as in the previous example, create an `HWV` object to represent the model:

$$dX_t = 0.2(0.1 - X_t)dt + 0.05dW_t$$

```
obj = hwv(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
obj =
   Class HWV: Hull-White/Vasicek
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

`CIR` and `HWV` constructors share the same interface and display methods. The only distinction is that `CIR` and `HWV` models constrain `Alpha` exponents to `1/2` and `0`, respectively. Furthermore, the `HWV` class also provides an additional method that simulates approximate analytic solutions (`simBySolution`) of separable models. This method simulates the state vector X_t using an approximation of the closed-form solution of diagonal drift `HWV` models. Each element of the state vector X_t is expressed as the sum of `NBROWNS` correlated Gaussian random draws added to a deterministic time-variable drift.

When evaluating expressions, all model parameters are assumed piece-wise constant over each simulation period. In general, this is *not* the exact solution to this `HWV` model, because the probability distributions of the simulated and true state vectors are identical *only* for piece-wise constant parameters. If $S(t,X_t)$, $L(t,X_t)$, and $V(t,X_t)$ are piece-wise constant over each observation

period, the state vector X_t is normally distributed, and the simulated process is exact for the observation times at which X_t is sampled.

Hull-White vs. Vasicek Models

Many references differentiate between Vasicek models and Hull-White models. Where such distinctions are made, Vasicek parameters are constrained to be constants, while Hull-White parameters vary deterministically with time. Think of Vasicek models in this context as constant-coefficient Hull-White models and equivalently, Hull-White models as time-varying Vasicek models. However, from an architectural perspective, the distinction between static and dynamic parameters is trivial. Since both models share the same general parametric specification as previously described, a single `HWV` class encompasses the models.

Solving Problems with SDE Models

In this section...

"Implementing Multidimensional Equity Market Models" on page 5-29

"Stochastic Interpolation and the Brownian Bridge" on page 5-42

"Inducing Dependence and Correlation" on page 5-48

"Incorporating Dynamic Behavior" on page 5-51

"End-of-Period Processes" on page 5-57

"User-Specified Random Number Generation: Stratified Sampling" on page 5-63

Implementing Multidimensional Equity Market Models

This example compares alternative implementations of a separable multivariate geometric Brownian motion process that is often referred to as a *multidimensional market model*. It simulates sample paths of an equity index portfolio using `SDE`, `SDEDDO`, `SDELD`, `CEV`, and `GBM` objects.

The market model to simulate is:

$$dX_t = \mu X_t dt + D(X_t)\sigma dW_t \tag{5-13}$$

where:

- μ is a diagonal matrix of index returns.
- D is a diagonal matrix with X_t along the diagonal.
- σ is a diagonal matrix of index standard deviation.

Implementation 1: Using SDE Objects

Create an `SDE` object to represent the equity market model.

1 Load the `SDE_Data` data set:

```
load SDE_Data
```

```
SDE_Data =
        Dates: [1359x1 double]
       Canada: [1359x1 double]
       France: [1359x1 double]
      Germany: [1359x1 double]
        Japan: [1359x1 double]
           UK: [1359x1 double]
           US: [1359x1 double]
    Euribor3M: [1359x1 double]

prices  =  [SDE_Data.Canada SDE_Data.France SDE_Data.Germany ...
     SDE_Data.Japan SDE_Data.UK SDE_Data.US];
```

2 Convert daily prices to returns:

```
returns =  price2ret(prices);
```

3 Compute data statistics to input to simulation methods:

```
nVariables  = size(returns, 2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
covariance  = cov(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));
```

4 Create simple anonymous drift and diffusion functions accessible by (t, X_t):

```
F = @(t,X) diag(expReturn) * X;
G = @(t,X) diag(X) * diag(sigma);
```

5 Use these functions to create an SDE object to represent the market model in Equation 5-13:

```
SDE = sde(F, G, 'Correlation', correlation, 'StartState', X)
SDE =
   Class SDE: Stochastic Differential Equation
   -------------------------------------------
     Dimensions: State = 6, Brownian = 6
```

```
   ----------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

The SDE constructor requires additional information to determine the dimensionality of the model, because the functions passed to the SDE constructor are known only by their (t, X_t) interface. In other words, the SDE constructor requires only two inputs: a drift-rate function and a diffusion-rate function, both accessible by passing the sample time and the corresponding state vector (t, X_t).

In this case, this information is insufficient to determine unambiguously the dimensionality of the state vector and Brownian motion. You resolve the dimensionality by specifying an initial state vector, StartState. Note that the SDE engine has assigned the default simulation method, simByEuler, to the Simulation parameter.

Implementation 2: Using SDEDDO Objects

Create an SDEDDO object to represent the market model in Equation 5-13:

1 Create drift and diffusion objects:

```
F = drift(zeros(nVariables,1), diag(expReturn))
F =
   Class DRIFT: Drift Rate Specification
   -------------------------------------
      Rate: drift rate function F(t,X(t))
         A: 6x1 double array
         B: 6x6 diagonal double array

G = diffusion(ones(nVariables,1), diag(sigma))
G =
   Class DIFFUSION: Diffusion Rate Specification
   ---------------------------------------------
       Rate: diffusion rate function G(t,X(t))
```

```
         Alpha: 6x1 double array
         Sigma: 6x6 diagonal double array
```

2 Pass the `drift` and `diffusion` objects to the `SDEDDO` constructor:

```
SDEDDO = sdeddo(F, G, 'Correlation', correlation, ...
      'StartState', 100)
SDEDDO =
   Class SDEDDO: SDE from Drift and Diffusion Objects
   --------------------------------------------------
     Dimensions: State = 6, Brownian = 6
   --------------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 6x1 double array
              B: 6x6 diagonal double array
          Alpha: 6x1 double array
          Sigma: 6x6 diagonal double array
```

The `SDEDDO` constructor requires two input objects that provide more information than the two functions from step 4 of "Implementation 1: Using SDE Objects" on page 5-29. Thus, the dimensionality is more easily resolved. In fact, the initial price of each index is as a scalar (`StartState = 100`). This is in contrast to the `SDE` constructor, which required an explicit state vector to uniquely determine the dimensionality of the problem.

Once again, the class of each object is clearly identified, and parameters display like fields of a structure. In particular, the `Rate` parameter of drift and diffusion objects is identified as a callable function of time and state, $F(t,X_t)$ and $G(t,X_t)$, respectively. The additional parameters, `A`, `B`, `Alpha`, and `Sigma`, are arrays of appropriate dimension, indicating static (non-time-varying) parameters. In other words, $A(t,X_t)$, $B(t,X_t)$, $Alpha(t,X_t)$, and $Sigma(t,X_t)$ are constant functions of time and state.

Implementation 3: Using SDELD, CEV, and GBM Objects

Create SDELD, CEV, and GBM objects to represent the market model in Equation 5-13.

1 Create an SDELD object:

```
SDELD = sdeld(zeros(nVariables,1), diag(expReturn), ...
   ones(nVariables,1), diag(sigma),'Correlation', ...
   correlation, 'StartState', X)
SDELD =
   Class SDELD: SDE with Linear Drift
   ----------------------------------------
     Dimensions: State = 6, Brownian = 6
   ----------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 6x1 double array
              B: 6x6 diagonal double array
          Alpha: 6x1 double array
          Sigma: 6x6 diagonal double array
```

2 Create a CEV object:

```
CEV = cev(diag(expReturn), ones(nVariables,1), ...
    diag(sigma), 'Correlation', correlation, ...
    'StartState', X)
CEV =
   Class CEV: Constant Elasticity of Variance
   ------------------------------------------
     Dimensions: State = 6, Brownian = 6
   ------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
```

```
     Simulation: simulation method/function simByEuler
         Return: 6x6 diagonal double array
          Alpha: 6x1 double array
          Sigma: 6x6 diagonal double array
```

3 Create a `GBM` object:

```
GBM = gbm(diag(expReturn), diag(sigma), 'Correlation', ...
    correlation, 'StartState', X)
GBM =
   Class GBM: Generalized Geometric Brownian Motion
   ------------------------------------------------
     Dimensions: State = 6, Brownian = 6
   ------------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 6x6 diagonal double array
          Sigma: 6x6 diagonal double array
```

Note the succession of interface restrictions:

- `SDELD` objects require you to specify `A`, `B`, `Alpha`, and `Sigma`.
- `CEV` objects require you to specify `Return`, `Alpha`, and `Sigma`.
- `GBM` objects require you to specify only `Return` and `Sigma`.

However, all three objects represent the same multidimensional market model.

Also, `CEV` and `GBM` objects display the underlying parameter `B` derived from the `SDELD` class as `Return`. This is an intuitive name commonly associated with equity models.

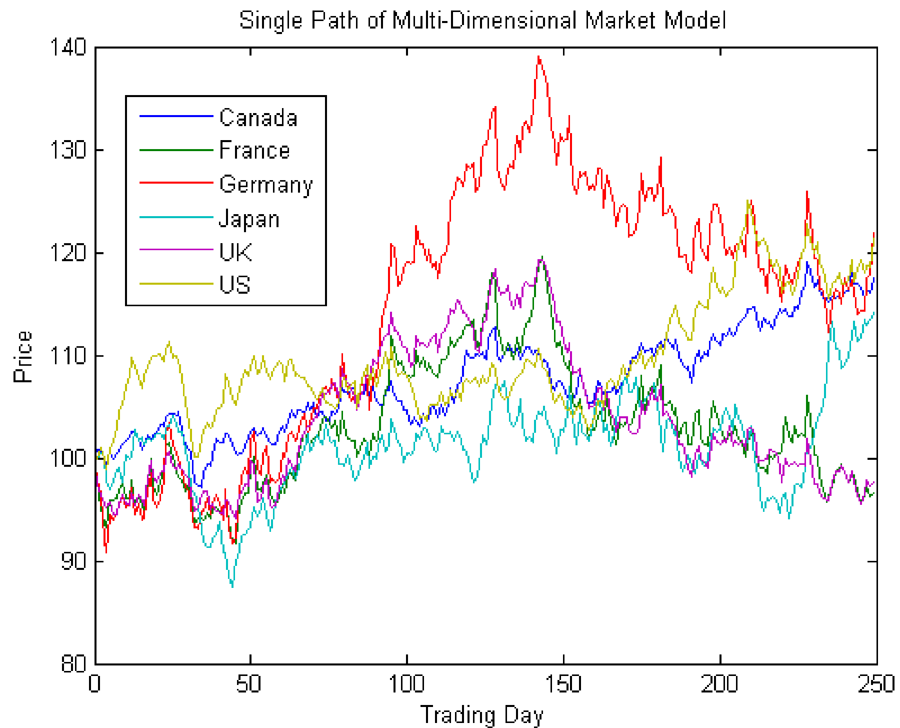
Implementation 4: Using the Default Simulate Method

Simulate a single path of correlated equity index prices over one calendar year (defined as approximately 250 trading days) using the default `simulate` method:

```
nPeriods = 249;      % # of simulated observations
dt       =   1;      % time increment = 1 day
randn('state', 100)
[S,T] = SDE.simulate(nPeriods, 'DeltaTime', dt);
```

The output array `S` is a 250-by-6 = `(NPERIODS + 1)`-by-`nVariables`-by-1 array with the same initial value, `100`, for all indices. Each row of `S` is an observation of the state vector X_t at time t.

```
whos S
  Name  Size    Bytes  Class  Attributes
  S     250x6   12000  double

plot(T, S), xlabel('Trading Day'), ylabel('Price')
title('Single Path of Multi-Dimensional Market Model')
legend({'Canada' 'France' 'Germany' 'Japan' 'UK' 'US'}, ...
   'Location', 'Best')
```



Implementation 5: Using the SimByEuler Method

1 Because `simByEuler` is a valid simulation method, you can call it directly, overriding the `Simulation` parameter's current method or function (which in this case is `simByEuler`). The following statements produce the same price paths as generated in "Implementation 4: Using the Default Simulate Method" on page 5-34:

```
randn('state', 100)
[S,T] = SDE.simByEuler(nPeriods, 'DeltaTime', dt);
```

2 Simulate 10 trials with the same initial conditions:

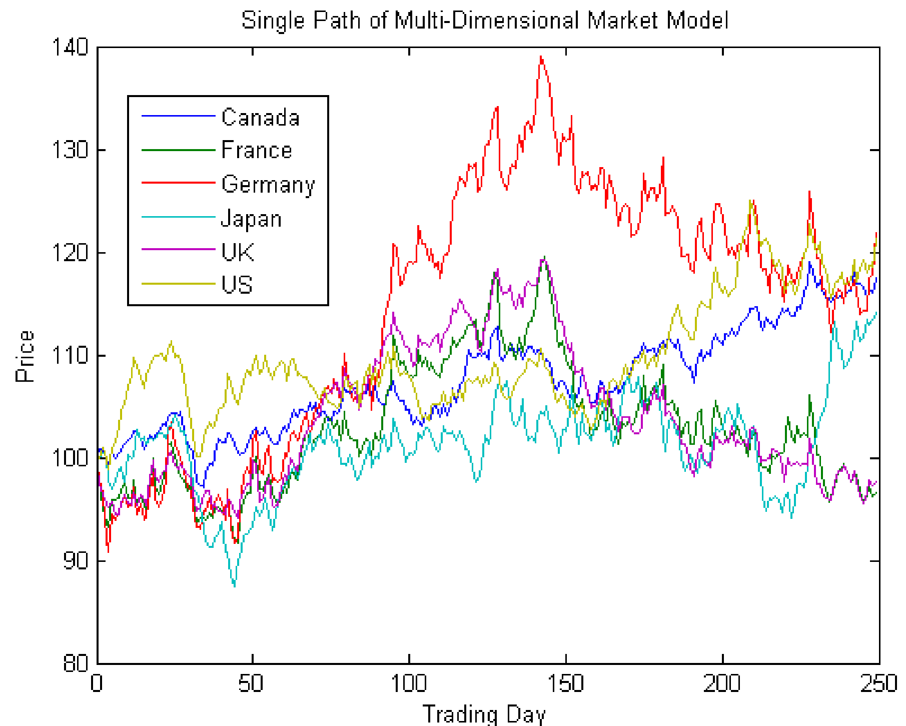
```
randn('state', 100)
[S,T] = SDE.simulate(nPeriods, 'DeltaTime', dt, 'nTrials', 10);
```

Now the output array `S` is an (`NPERIODS + 1`)-by-`nVariables`-by-`nTrials` time-series array:

```
whos S
  Name       Size          Bytes   Class     Attributes
  S          250x6x10      120000  double
```

whose first realization is identical to the single paths just plotted:

```
plot(T, S(:,:,1)), xlabel('Trading Day'), ylabel('Price')
title('First Path of Multi-Dimensional Market Model')
legend({'Canada' 'France' 'Germany' 'Japan' 'UK' 'US'},...
    'Location', 'Best')
```



Implementation 6: Using GBM Simulation Methods

Finally, consider simulation using `GBM` simulation methods. Separable `GBM` models have two specific simulation methods:

- An overloaded Euler simulation method, designed for optimal performance
- A method that provides an approximate solution of the underlying stochastic differential equation, designed for accuracy

1. To illustrate the performance benefit of the overloaded Euler approximation method, increase the number of trials to `10000`:

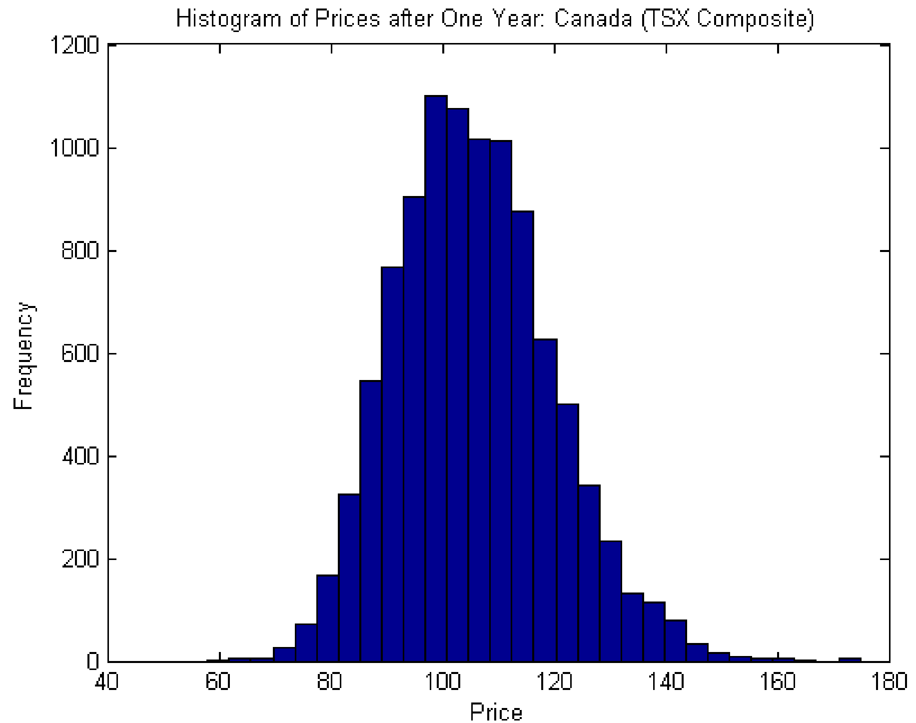
```
randn('state', 100)
[X,T] = GBM.simulate(nPeriods, 'DeltaTime', dt, ...
        'nTrials', 10000);
```

   The output `X` is a much larger time-series array:

```
whos X
  Name      Size           Bytes      Class     Attributes
  X         250x6x10000    120000000  double
```

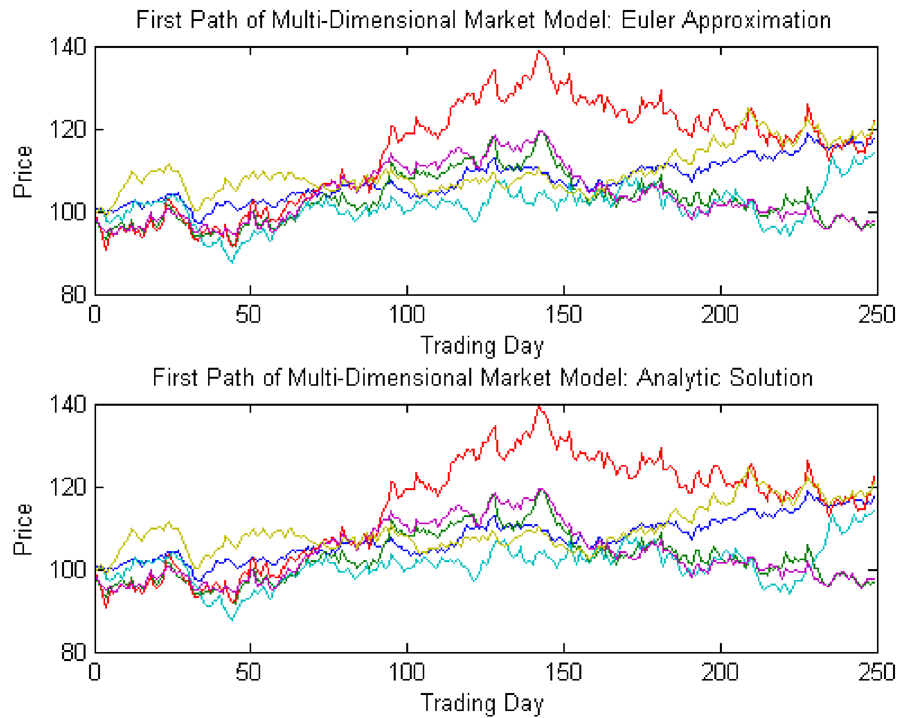
2. With this sample size, you can examine the terminal distribution of, for example, Canada's TSX Composite to verify qualitatively the log-normal character of the data:

```
hist(X(end,1,:), 30), xlabel('Price'), ylabel('Frequency')
  title('Histogram of Prices after One Year: ...
  Canada (TSX Composite)')
```



3 Simulate 10 trials of the solution and plot the first trial:

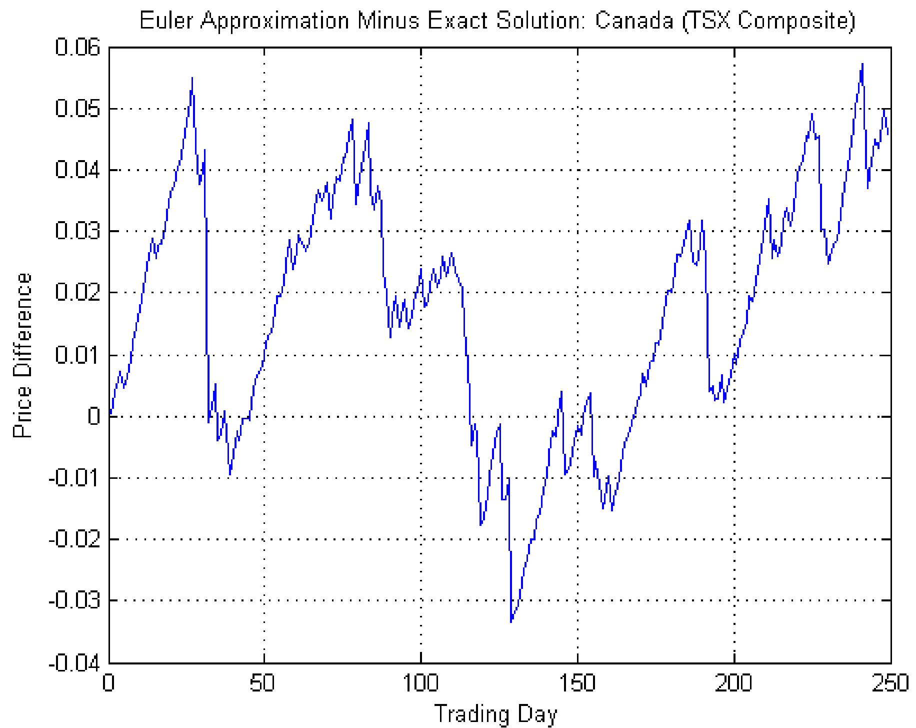
```
randn('state', 100)
 [X,T] = GBM.simBySolution(nPeriods,...
   'DeltaTime', dt, 'nTrials', 10);
subplot(2,1,1)
plot(T, S(:,:,1)), xlabel('Trading Day'),...
   ylabel('Price')
title('First Path of Multi-Dimensional Market Model: ...
   Euler Approximation')
subplot(2,1,2)
plot(T, X(:,:,1)), xlabel('Trading Day'),...
   ylabel('Price')
title('First Path of Multi-Dimensional Market Model:...
   Analytic Solution')
```



In this example, all parameters are constants, and `simBySolution` does indeed sample the exact solution. The details of a single index for any given trial show that the price paths of the Euler approximation and the exact solution are close, but not identical.

The following plot illustrates the difference between the two methods:

```
subplot(1,1,1)
plot(T, S(:,1,1) - X(:,1,1), 'blue'), grid('on')
xlabel('Trading Day'), ylabel('Price Difference')
title('Euler Approximation Minus Exact Solution:...
   Canada (TSX Composite)')
```



The `simByEuler` Euler approximation literally evaluates the stochastic differential equation directly from the equation of motion, for some suitable value of the `dt` time increment. This simple approximation suffers from discretization error. This error can be attributed to the discrepancy between the choice of the `dt` time increment and what in theory is a continuous-time parameter.

The discrete-time approximation improves as `DeltaTime` approaches zero. The Euler method is often the least accurate and most general method available. All models shipped in the simulation suite have this method.

In contrast, the `simBySolution` method provides a more accurate description of the underlying model. This method simulates the price paths by an approximation of the closed-form solution of separable models. Specifically, it applies an Euler approach to a transformed process, which in general is

not the exact solution to this GBM model. This is because the probability distributions of the simulated and true state vectors are identical only for piece-wise constant parameters.

When all model parameters are piece-wise constant over each observation period, the simulated process is exact for the observation times at which the state vector is sampled. Since all parameters are constants in this example, simBySolution does indeed sample the exact solution.

For an example of how to use simBySolution to optimize the accuracy of solutions, see “Optimizing Accuracy of Solutions” on page 5-74.

Stochastic Interpolation and the Brownian Bridge

All simulation methods require that you specify a time grid by specifying the number of periods (NPERIODS). You can also optionally specify a scalar or vector of strictly positive time increments (DeltaTime) and intermediate time steps (NSTEPS). These parameters, along with an initial sample time associated with the object (StartTime), uniquely determine the sequence of times at which the state vector is sampled. Thus, simulation methods allow you to traverse the time grid from beginning to end (that is, from left to right).

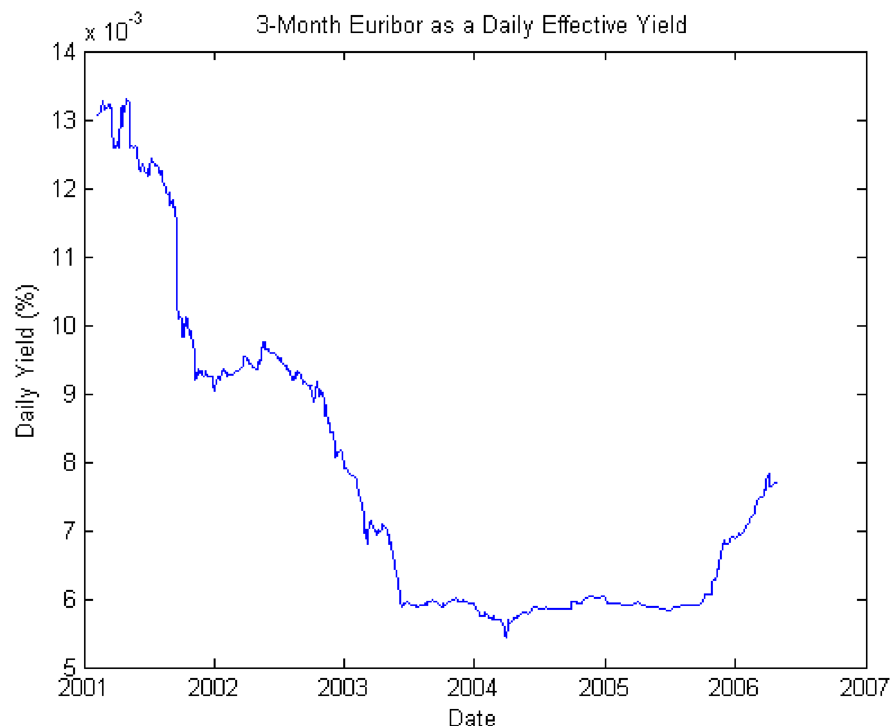
In contrast, interpolation methods allow you to traverse the time grid in any order, allowing both forward and backward movements in time. They allow you to specify a vector of interpolation times whose elements do not have to be unique.

Many references define the Brownian Bridge as a conditional simulation combined with a scheme for traversing the time grid, effectively merging two distinct algorithms. In contrast, the interpolation method offered here provides additional flexibility by intentionally separating the algorithms. In this method for moving about a time grid, you perform an initial Monte Carlo simulation to sample the state at the terminal time, and then successively sample intermediate states by stochastic interpolation. The first few samples determine the overall behavior of the paths, while later samples progressively refine the structure. Such algorithms are often called *variance reduction techniques*. This algorithm is particularly simple when the number of interpolation times is a power of 2. In this case, each interpolation falls midway between two known states, refining the interpolation using a method

like bisection. This example highlights the flexibility of refined interpolation by implementing this power-of-two algorithm.

1 Load a historical data set of three-month Euribor rates (`http://www.euribor.org/default.htm`), observed daily, and corresponding trading dates spanning the time interval from February 7, 2001 through April 24, 2006:

```
clear, clf
load SDE_Data
plot(SDE_Data.Dates, 100 * SDE_Data.Euribor3M)
datetick('x'), xlabel('Date'), ylabel('Daily Yield (%)')
title('3-Month Euribor as a Daily Effective Yield')
```



2 Now fit a simple univariate `Vasicek` model to the daily equivalent yields of the three-month Euribor data:

$$dX_t = S(L - X_t)dt + \sigma dW_t$$

Given initial conditions, the distribution of the short rate at some time T in the future is Gaussian with mean:

$$E(X_T) = X_0 e^{-ST} + L(1 - e^{-ST})$$

and variance:

$$Var(X_T) = \sigma^2(1 - e^{-ST}) / 2S$$

To calibrate this simple short rate model, rewrite it in more familiar regression format:

$$y_t = \alpha + \beta x_t + \varepsilon_t$$

where:

$$y_t = dX_t, \alpha = SLdt, \beta = -Sdt$$

perform an ordinary linear regression where the model volatility is proportional to the standard error of the residuals:

$$\alpha = \sqrt{Var(\varepsilon_t) / dt}$$

```
yields      = SDE_Data.Euribor3M;
regressors = [ones(length(yields) - 1, 1) yields(1:end-1)];
[coefficients, intervals, residuals] = ...
   regress(diff(yields), regressors);
dt    = 1;  % time increment = 1 day
speed = -coefficients(2)/dt;
level = -coefficients(1)/coefficients(2);
sigma =  std(residuals)/sqrt(dt);
```

3 Create an `HWV` object with an initial `StartState` set to the most recently observed short rate:

```
obj = hwv(speed, level, sigma, 'StartState', yields(end))
obj =
```

```
Class HWV: Hull-White/Vasicek
----------------------------------------
  Dimensions: State = 1, Brownian = 1
----------------------------------------
   StartTime: 0
  StartState: 7.70408e-005
 Correlation: 1
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
       Sigma: 4.77637e-007
       Level: 6.00424e-005
       Speed: 0.00228854
```

4 Assume, for example, that you simulate the fitted model over 64 (2^6) trading days, using a refined Brownian bridge with the power-of-two algorithm instead of the usual beginning-to-end Monte Carlo simulation approach. Furthermore, assume that the initial time and state coincide with those of the last available observation of the historical data, and that the terminal state is the expected value of the Vasicek model 64 days into the future. In this case, you can assess the behavior of various paths that all share the same initial and terminal states, perhaps to support pricing path-dependent interest rate options over a three-month interval.

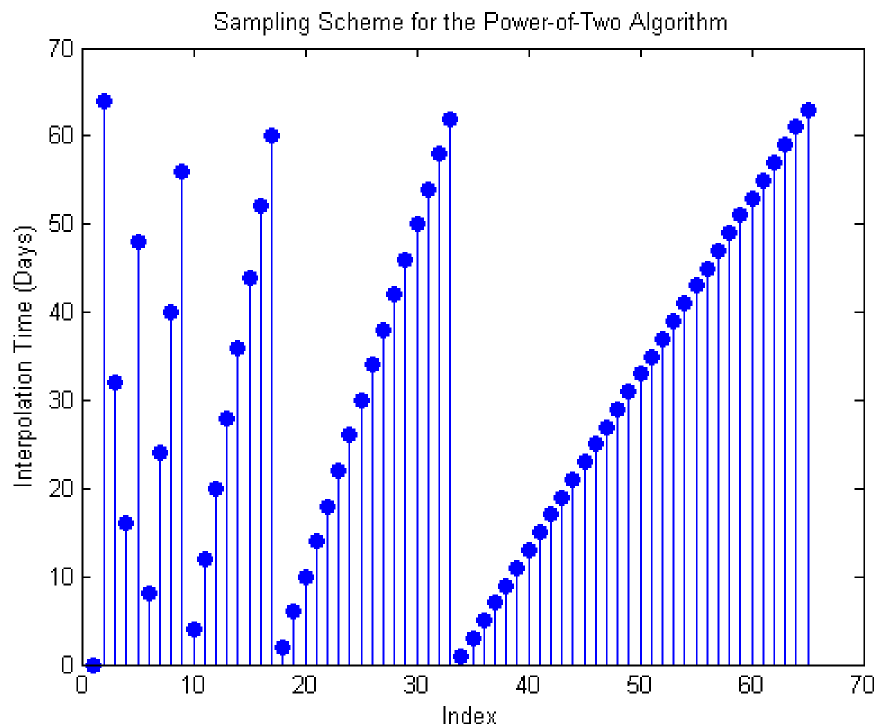
Create a vector of interpolation times to traverse the time grid by moving both forward and backward in time. Specifically, the first interpolation time is set to the most recent short rate observation time, the second interpolation time is set to the terminal time, and subsequent interpolation times successively sample intermediate states:

```
T      = 64;
times  = (1:T)';
t      = NaN(length(times) + 1, 1);
t(1)   = obj.StartTime;
t(2)   = T;
delta  = T;
jMax   = 1;
iCount = 3;

for k = 1:log2(T)
```

```
        i = delta / 2;
        for j = 1:jMax
            t(iCount) = times(i);
            i         = i + delta;
            iCount    = iCount + 1;
        end
        jMax  = 2 * jMax;
        delta = delta / 2;
    end
```

5 Examine the sequence of interpolation times generated by this algorithm:

```
stem(1:length(t), t, 'filled')
xlabel('Index'), ylabel('Interpolation Time (Days)')
title ('Sampling Scheme for the Power-of-Two Algorithm')
```



The first few samples are widely separated in time and determine the course structure of the paths. Later samples are closely spaced and progressively refine the detailed structure.

6 Now that you have generated the sequence of interpolation times, initialize a course time-series grid to begin the interpolation. The sampling process begins at the last observed time and state taken from the historical short rate series, and ends 64 days into the future at the expected value of the `Vasicek` model derived from the calibrated parameters:

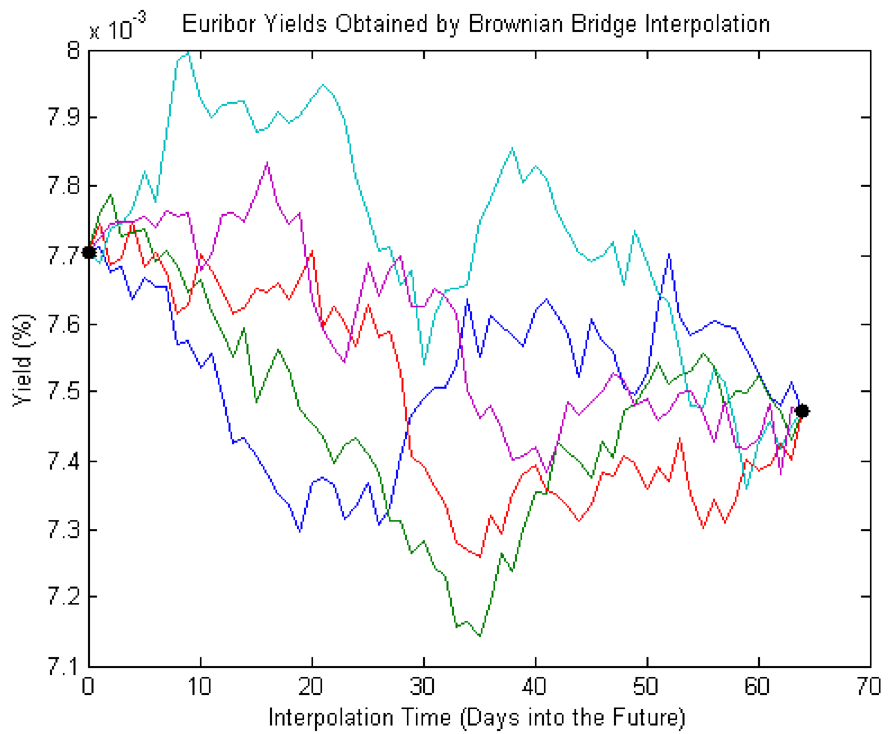
```
average = obj.StartState * exp(-speed * T) + ...
    level * (1 - exp(-speed * T));
X       = [obj.StartState ; average];
```

7 Generate 5 sample paths, setting the `Refine` input flag to `TRUE` to insert each new interpolated state into the time series grid as it becomes available. Perform interpolation on a trial-by-trial basis. Because the input time series `X` has five trials (where each page of the 3-dimensional time series represents an independent trial), the interpolated output series `Y` also has five pages:

```
nTrials = 5;
randn('state', 0)
Y = obj.interpolate(t, X(:,:,ones(nTrials,1)), ...
  'Times',[obj.StartTime  T], 'Refine', true);
```

8 Plot the resulting sample paths. Because the interpolation times do not monotonically increase, sort the times and reorder the corresponding short rates:

```
[t,i] = sort(t);
Y     = squeeze(Y);
Y     = Y(i,:);
plot(t, 100 * Y), hold('on')
plot(t([1 end]), 100 * Y([1 end],1),'. black', ...
     'MarkerSize', 20)
xlabel('Interpolation Time (Days into the Future)')
ylabel('Yield (%)'), hold('off')
title ('Euribor Yields Obtained by Brownian Bridge ...
    Interpolation')
```



The short rates in this plot represent alternative sample paths that share the same initial and terminal values. They illustrate a special, though simplistic, case of a broader sampling technique known as *stratified sampling*. For a more sophisticated example of stratified sampling, see "User-Specified Random Number Generation: Stratified Sampling" on page 5-63.

Although this simple example simulated a univariate `Vasicek` interest rate model, it applies to problems of any dimensionality.

Inducing Dependence and Correlation

This example illustrates two techniques that induce dependence between individual elements of a state vector. It also illustrates the interaction between `Sigma` and `Correlation`.

The first technique generates correlated Gaussian variates to form a Brownian motion process with dependent components. These components are then weighted by a diagonal volatility or exposure matrix `Sigma`.

The second technique generates independent Gaussian variates to form a standard Brownian motion process, which are then weighted by the lower Cholesky factor of the desired covariance matrix. Although these techniques can be used on many models, the relationship between them is most easily illustrated by working with a separable `GBM` model (see "Implementing Multidimensional Equity Market Models" on page 5-29). The market model to simulate is:

$$dX_t = \mu X_t dt + \sigma X_t dW_t$$

where μ is a diagonal matrix.

1 Load the `SDE_Data` data set:

```
load SDE_Data, SDE_Data
SDE_Data =
        Dates: [1359x1 double]
       Canada: [1359x1 double]
       France: [1359x1 double]
      Germany: [1359x1 double]
        Japan: [1359x1 double]
           UK: [1359x1 double]
           US: [1359x1 double]
    Euribor3M: [1359x1 double]

prices  = [SDE_Data.Canada  SDE_Data.France...
      SDE_Data.Germany  SDE_Data.Japan...
      SDE_Data.UK       SDE_Data.US];
```

2 Convert the daily prices to returns:

```
returns =  price2ret(prices);
```

3 Specify `Sigma` and `Correlation` using the first technique:

   a Using the first technique, specify `Sigma` as a diagonal matrix of asset return standard deviations:

```
expReturn   = diag(mean(returns));  % expected return vector
sigma       = diag(std(returns));   % volatility of returns
```

b Specify `Correlation` as the sample correlation matrix of those returns. In this case, the components of the Brownian motion are dependent:

```
correlation = corrcoef(returns);
GBM1        = gbm(expReturn, sigma, 'Correlation', ...
      correlation);
```

4 Specify `Sigma` and `Correlation` using the second technique:

a Using the second technique, specify `Sigma` as the lower Cholesky factor of the asset return covariance matrix:

```
covariance = cov(returns);
sigma      = cholcov(covariance)';
```

b Set `Correlation` to an identity matrix:

```
GBM2       = gbm(expReturn, Sigma);
```

Here, `Sigma` captures both the correlation and magnitude of the asset return uncertainty. In contrast to the first technique, the components of the Brownian motion are independent. Also, this technique accepts the default assignment of an identity matrix to `Correlation`, and is more straightforward.

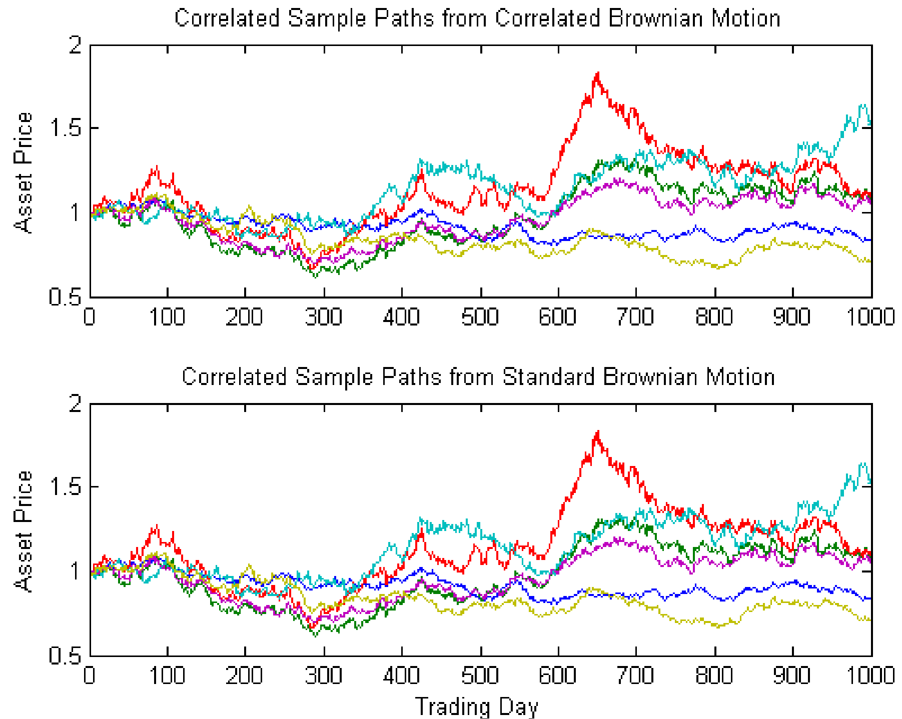
5 Simulate a single trial of 1000 observations (roughly four years of daily data) using both techniques. By default, all state variables start at `1`:

```
randn('state', 0)
[X1,T] = GBM1.simByEuler(1000);  % correlated Brownian motion
randn('state', 0)
[X2,T] = GBM2.simByEuler(1000);  % standard Brownian motion
```

6 When based on the same initial random number state, each technique generates identical asset price paths:

```
subplot(2,1,1), plot(T, X1)
title('Correlated Sample Paths ...
   from Correlated Brownian Motion')
```

```
ylabel('Asset Price')
subplot(2,1,2), plot(T, X2)
title('Correlated Sample Paths ...
   from Standard Brownian Motion')
xlabel('Trading Day'), ylabel('Asset Price')
```



Incorporating Dynamic Behavior

As previously discussed, object parameters may be evaluated as if they are MATLAB® functions accessible by a common interface. This accessibility provides the impression of dynamic behavior regardless of whether the underlying parameters are truly time-varying. Furthermore, because parameters are accessible by a common interface, seemingly simple linear constructs may in fact represent complex, nonlinear designs.

For example, consider a univariate geometric Brownian motion (GBM) model of the form:

$$dX_t = \mu(t)X_t dt + \sigma(t)X_t dW_t$$

In this model, the return, $\mu(t)$, and volatility, $\sigma(t)$, are generally dynamic parameters of time alone. However, when creating a `GBM` object to represent the underlying model, such dynamic behavior must be accessible by the common (t, X_t) interface. This reflects the fact that `GBM` models (and others) are restricted parameterizations that derive from the general `SDE` class.

As a convenience, you can specify parameters of restricted models, such as `GBM` models, as traditional MATLAB arrays of appropriate dimension. In this case, such arrays represent a static special case of the more general dynamic situation accessible by the (t, X_t) interface.

Moreover, when you enter parameters as functions, object constructors can verify that they return arrays of correct size by evaluating them at the initial time and state. Otherwise, object constructors have no knowledge of any particular functional form.

The following example illustrates a technique that includes dynamic behavior by mapping a traditional MATLAB time-series array to a callable function with a (t, X_t) interface. It also compares the function with an otherwise identical model with constant parameters.

Because time-series arrays represent dynamic behavior that must be captured by functions accessible by the (t, X_t) interface, you need utilities to convert traditional time-series arrays into callable functions of time and state. The following example shows how to do this using the conversion function `ts2func` (time series to function).

1 Load a daily historical data set of 3-month Euribor rates and closing index levels of France's CAC 40 spanning the time interval February 7, 2001 to April 24, 2006:

```
clear
load SDE_Data
```

2 Simulate risk-neutral sample paths of the CAC 40 index using a geometric Brownian motion (GBM) model:

$$dX_t = r(t)X_t dt + \sigma X_t dW_t$$

where *r(t)* represents evolution of the risk-free rate of return.

Furthermore, assume that you need to annualize the relevant information derived from the daily data (annualizing the data is optional, but is useful for comparison to other examples), and that each calendar year comprises 250 trading days:

```
dt      = 1 / 250;
returns = price2ret(SDE_Data.France);
sigma   = std(returns) * sqrt(250);
yields  = SDE_Data.Euribor3M;
yields  = 360 * log(1 + yields);
```

3 Compare the resulting sample paths obtained from two risk-neutral historical simulation approaches, where the daily Euribor yields serve as a proxy for the risk-free rate of return.

a The first approach specifies the risk-neutral return as the sample average of Euribor yields, and therefore assumes a constant (non-dynamic) risk-free return:

```
nPeriods = length(yields);  % Simulated observations

randn('state', 25)
obj    = gbm(mean(yields), diag(sigma), 'StartState', 100)
[X1,T] = obj.simulate(nPeriods, 'DeltaTime', dt);

obj =
   Class GBM: Generalized Geometric Brownian Motion
   ------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------------
      StartTime: 0
      StartState: 100
      Correlation: 1
      Drift: drift rate function F(t,X(t))
```

```
Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
Return: 0.0278117
Sigma: 0.231875
```

b In contrast, the second approach specifies the risk-neutral return as the historical time series of Euribor yields. It therefore assumes a dynamic, yet deterministic, rate of return; this example does not illustrate stochastic interest rates. To illustrate this dynamic effect, use the `ts2func` utility:

```
r = ts2func(yields, 'Times', (0:nPeriods - 1)');
```

`ts2func` packages a specified time series array inside a callable function of time and state, and synchronizes it with an optional time vector. For instance:

```
r(0,100)
ans =
    0.0470
```

evaluates the function at ($t = 0, X_t = 100$) and returns the first observed Euribor yield. However, you can also evaluate the resulting function at any intermediate time t and state X_t:

```
r(7.5,200)
ans =
    0.0472
```

Furthermore, the following command produces the same result when called with time alone:

```
r(7.5)
ans =
    0.0472
```

The equivalence of these last two commands highlights some important features.

When you specify parameters as functions, they must evaluate properly when passed a scalar, real-valued sample time (t), and a *NVARS*-by-1

state vector (X_t). They must also generate an array of appropriate dimensions, which in the first case is a scalar constant, and in the second case is a scalar, piece-wise constant function of time alone.

You are not required to use either time (t) or state (X_t). In the current example, the function evaluates properly when passed time followed by state, thereby satisfying the minimal requirements. The fact that it also evaluates correctly when passed only time simply indicates that the function does not require the state vector X_t. The important point to make is that it works when you pass it (t, X_t).

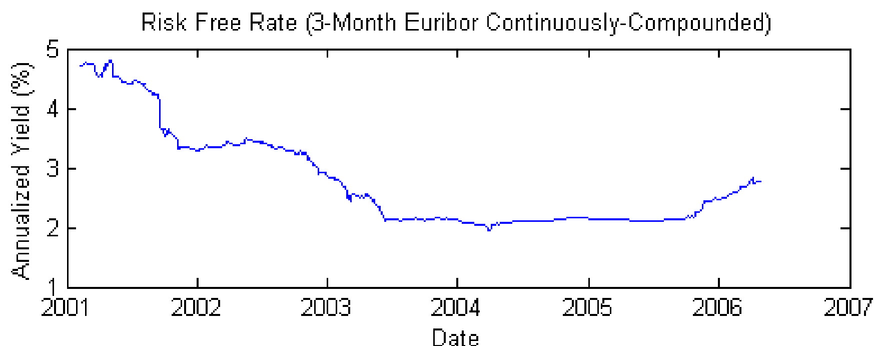
Furthermore, the `ts2func` function performs a zero-order-hold (ZOH) piece-wise constant interpolation. The notion of piece-wise constant parameters is pervasive throughout the SDE architecture, and is discussed in more detail in "Optimizing Accuracy of Solutions" on page 5-74.

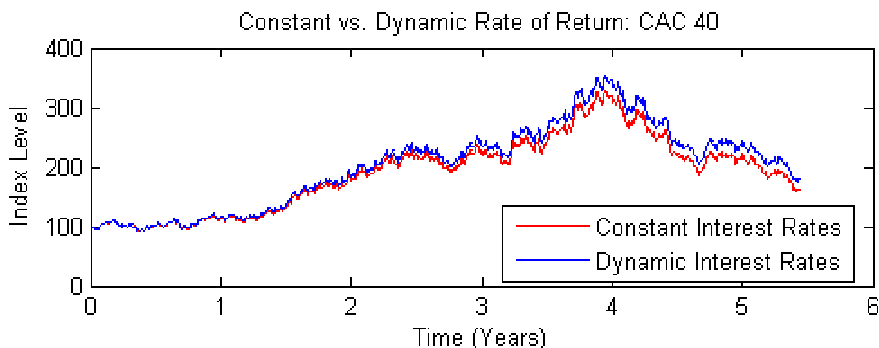
4 Complete the comparison by performing the second simulation using the same initial random number state:

```
randn('state', 25)
obj = gbm(r, diag(sigma), 'StartState', 100)
X2  = obj.simulate(nPeriods, 'DeltaTime', dt);

obj =
   Class GBM: Generalized Geometric Brownian Motion
   ------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------------
      StartTime: 0
     StartState: 100
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: function ts2func/vector2Function
          Sigma: 0.231875
```

5 Plot the series of risk-free reference rates to compare the two simulation trials:

```
subplot(2,1,1)
plot(SDE_Data.Dates, 100 * yields)
datetick('x'), xlabel('Date'), ...
    ylabel('Annualized Yield (%)')
title('Risk Free Rate ...
   (3-Month Euribor Continuously-Compounded)')
subplot(2,1,2)
plot(T, X1, 'red', T, X2, 'blue')
xlabel('Time (Years)'), ylabel('Index Level')
title('Constant vs. Dynamic Rate of Return: CAC 40')
legend({'Constant Interest Rates'...
'Dynamic Interest Rates'}, 'Location', 'Best')
```





The paths are close but not exact. The blue line in the last plot uses all the historical Euribor data, and illustrates a single trial of a historical simulation.

End-of-Period Processes

Ensuring Positive State Variables

All simulation and interpolation methods allow you to specify a sequence of functions, or background processes, to evaluate at the end of every sample time period. This period includes any intermediate time steps determined by the optional NSTEPS input, as discussed in "Optimizing Accuracy of Solutions" on page 5-74. These functions are specified as callable functions of time and state, and must return an updated state vector X_t:

$$X_t = f(t, X_t)$$

You must specify multiple processing functions as a cell array of functions. These functions are invoked in the order in which they appear in the cell array.

Processing functions are not required to use time (t) or state (X_t). They are also not required to update or change the input state vector. In fact, simulation and interpolation methods have no knowledge of any implementation details, and in this respect, they only adhere to a published interface.

Such processing functions provide a powerful modeling tool that can solve a variety of problems. Such functions allow you to, for example, specify boundary conditions, accumulate statistics, plot graphs, and price path-dependent options.

Except for Brownian motion (BM) models, the individual components of the simulated state vector typically represent variables whose real-world counterparts are inherently positive quantities, such as asset prices or interest rates. However, by default, most of the simulation and interpolation methods provided here model the transition between successive sample times as a scaled (possibly multivariate) Gaussian draw. Consequently, when approximating a continuous-time process in discrete time, the state vector may not remain positive. The only exception is the simBySolution logarithmic transform of separable geometric Brownian motion models. Moreover, by default, none of the simulation and interpolation methods make adjustments to the state vector. Therefore, you are responsible for ensuring that all components of the state vector remain positive as appropriate.

Fortunately, specifying non-negative states ensures a simple end-of-period processing adjustment. Although this adjustment is widely applicable, it is revealing when applied to a univariate CIR square-root diffusion model:

$$dX_t = 0.25(0.1 - X_t)dt + 0.2X_t^{\frac{1}{2}}dW_t = S(L - X_t)dt + \sigma X_t^{\frac{1}{2}}dW_t$$

Perhaps the primary appeal of univariate CIR models where:

$$2SL \geq \sigma^2$$

is that the short rate remains positive. However, the positivity of short rates only holds for the underlying continuous-time model.

1 To illustrate the latter statement, simulate daily short rates of the CIR model over one calendar year (approximately 250 trading days):

```
randn('state', 10)
obj   = cir(0.25, @(t,X) 0.1, 0.2, 'StartState', 0.02);
[X,T] = obj.simByEuler(250, 'DeltaTime', ...
    1/250, 'nTrials', 5);
```

Interest rates can become negative if the resulting paths are simulated in discrete time. Moreover, since CIR models incorporate a square root diffusion term, the short rates might even become complex:

```
[T(200:210)         X(200:210,1,5)]

ans =
   0.7960              0.0023
   0.8000              0.0023
   0.8040              0.0022
   0.8080              0.0010
   0.8120              0.0005
   0.8160              0.0003
   0.8200             -0.0001
   0.8240             -0.0000 - 0.0002i
   0.8280              0.0002 - 0.0003i
   0.8320              0.0005 - 0.0004i
```

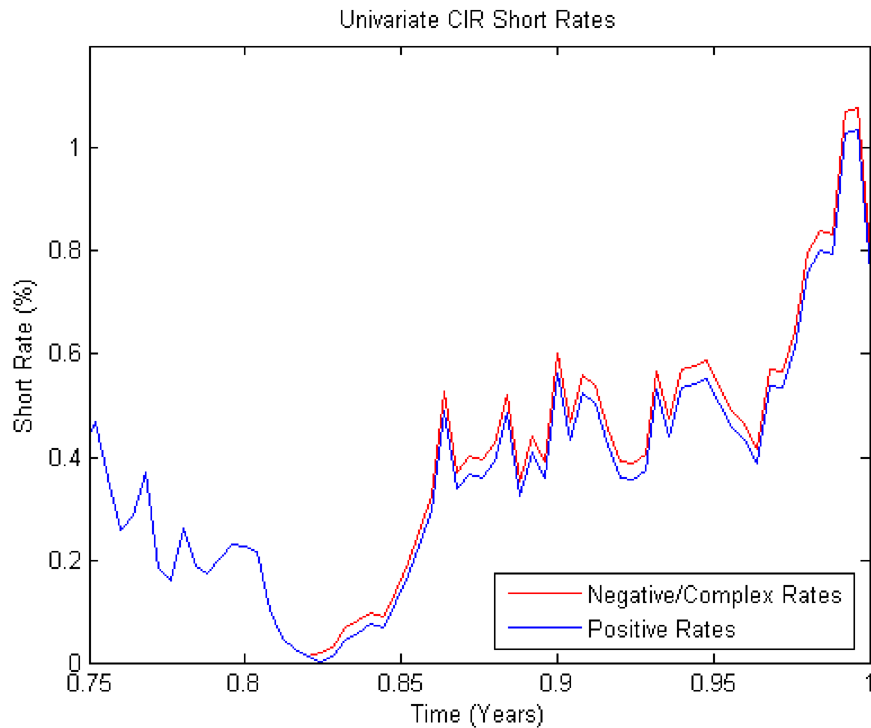
```
    0.8360                 0.0007 - 0.0004i
```

2 Repeat the simulation, this time specifying a processing function that takes the absolute magnitude of the short rate at the end of each period. You can access the processing function by time and state (t, X_t), but it only uses the state vector of short rates X_t:

```
randn('state', 10)
[Y,T] = obj.simByEuler(250, 'DeltaTime', 1/250, ...
   'nTrials', 5, 'Processes', @(t,X) abs(X));
```

3 Graphically compare the magnitude of the unadjusted path (with negative and complex numbers!) to the corresponding path kept positive by using an end-of-period processing function over the time span of interest:

```
clf
plot(T, 100 * abs(X(:,1,5)), 'red', T, ...
  100 * Y(:,1,5), 'blue')
axis([0.75 1 0 1.2])
xlabel('Time (Years)'), ylabel('Short Rate (%)')
title('Univariate CIR Short Rates')
legend({'Negative/Complex Rates' 'Positive Rates'}, ...
  'Location', 'Best')
```



Black-Scholes Option Pricing

As discussed in "Ensuring Positive State Variables" on page 5-57, all simulation and interpolation methods allow you to specify one or more functions of the form:

$$X_t = f(t, X_t)$$

to evaluate at the end of every sample time.

The previous example illustrated a simple, common end-of-period processing function to ensure non-negative interest rates. This example illustrates a processing function that allows you to avoid simulation outputs altogether.

Consider pricing European stock options by Monte Carlo simulation within a Black-Scholes-Merton framework. Assume that the stock has the following characteristics:

- The stock currently trades at 100.
- The stock pays no dividends.
- The stock's volatility is 50% per annum.
- The option strike price is 95.
- The option expires in three months.
- The risk-free rate is constant at 10% per annum.

To solve this problem, model the evolution of the underlying stock by a univariate geometric Brownian motion (GBM) model with constant parameters:

$$dX_t = 0.1X_t dt + 0.5X_t dW_t$$

Furthermore, assume that the stock price is simulated daily, and that each calendar month comprises 21 trading days:

```
strike   = 95;
rate     = 0.1;
sigma    = 0.5;
dt       = 1 / 252;
nPeriods = 63;
T        = nPeriods * dt;
obj = gbm(rate, sigma, 'StartState', 100);
```

The goal is to simulate independent paths of daily stock prices, and calculate the price of European options as the risk-neutral sample average of the discounted terminal option payoff at expiration 63 days from now. This example calculates option prices by two approaches:

- A Monte Carlo simulation that explicitly requests the simulated stock paths as an output. The output paths are then used to price the options.
- An end-of-period processing function, accessible by time and state, that records the terminal stock price of each sample path. This processing

function is implemented as a nested function with access to shared information. For more information, see the demo `blackScholesExample.m`.

1. Before simulation, invoke the example file to access the end-of-period processing function:

```
nTrials = 10000; % Number of independent trials (i.e., paths)
f = blackScholesExample(nPeriods, nTrials)
f =
    BlackScholes: @blackScholesExample/saveTerminalStockPrice
       CallPrice: @blackScholesExample/getCallPrice
        PutPrice: @blackScholesExample/getPutPrice
```

2. Simulate 10000 independent trials (sample paths). Request the simulated stock price paths as an output, and specify an end-of-period processing function:

```
randn('state', 0)
X = obj.simBySolution(nPeriods, 'DeltaTime', dt, ...
  'nTrials', nTrials, 'Processes', f.BlackScholes);
```

3. Calculate the option prices directly from the simulated stock price paths. Because these are European options, ignore all intermediate stock prices:

```
call = mean(exp(-rate * T) * max(squeeze(X(end,:,:)) ...
   - strike, 0))
put  = mean(exp(-rate * T) * max(strike - ...
   squeeze(X(end,:,:)), 0))
call =
   13.9964
put  =
    6.2028
```

4. Price the options indirectly by invoking the nested functions:

```
f.CallPrice(strike, rate)
f.PutPrice (strike, rate)
ans =
   13.9964
ans =
    6.2028
```

For reference, the theoretical call and put prices computed from the Black-Scholes option formulas are `13.6953` and `6.3497`, respectively.

5. Although steps 3 and 4 produce the same option prices, the latter approach works directly with the terminal stock prices of each sample path. Therefore, it is much more memory efficient. In this example, there is no compelling reason to request an output.

User-Specified Random Number Generation: Stratified Sampling

Simulation methods allow you to specify a noise process directly, as a callable function of time and state:

$$z_t = Z(t, X_t)$$

Stratified sampling is a variance reduction technique that constrains a proportion of sample paths to specific subsets (or *strata*) of the sample space.

This example specifies a noise function to stratify the terminal value of a univariate equity price series. Starting from known initial conditions, the function first stratifies the terminal value of a standard Brownian motion, and then samples the process from beginning to end by drawing conditional Gaussian samples using a Brownian bridge.

The stratification process assumes that each path is associated with a single stratified terminal value such that the number of paths is equal to the number of strata. This technique is called *proportional sampling*. This example is similar to, yet more sophisticated than, the one discussed in "Stochastic Interpolation and the Brownian Bridge" on page 5-42. Since stratified sampling requires knowledge of the future, it also requires more sophisticated time synchronization; specifically, the function in this example requires knowledge of the entire sequence of sample times. For more information, see the demo `stratifiedExample.m`.

The function implements proportional sampling by partitioning the unit interval into bins of equal probability by first drawing a random number uniformly distributed in each bin. The inverse cumulative distribution function of a standard *N(0,1)* Gaussian distribution then transforms these

stratified uniform draws. Finally, the resulting stratified Gaussian draws are scaled by the square root of the terminal time to stratify the terminal value of the Brownian motion.

The noise function does not return the actual Brownian paths, but rather the Gaussian draws $Z(t,X_t)$ that drive it.

This example first stratifies the terminal value of a univariate, zero-drift, unit-variance-rate Brownian motion (BM) model:

$$dX_t = dW_t$$

1 Assume that 10 paths of the process are simulated daily over a three-month period. Also assume that each calendar month and year consist of 21 and 252 trading days, respectively:

```
randn('state', 10), rand('twister', 0)
dt       = 1 / 252;        % 1 day = 1/252 years
nPeriods = 63;             % 3 months = 63 trading days
 T       = nPeriods * dt;  % 3 months = 0.25 years
nPaths   = 10;            % # of simulated paths
obj      = bm(0, 1, 'StartState', 0);
sampleTimes = cumsum([obj.StartTime ...
   dt(ones(nPeriods,1))]);
z        = stratifiedExample(nPaths, sampleTimes)
z        = @stratifiedExample/stratifiedSampling
```

2 Simulate the standard Brownian paths by explicitly passing the stratified sampling function to the simulation method:

```
X = obj.simulate(nPeriods, 'DeltaTime', dt, ...
'nTrials', nPaths, 'Z', z);
```

3 For convenience, reorder the output sample paths by reordering the 3-dimensional output to a 2-dimensional equivalent array:

```
X = squeeze(X);
```

4 Verify the stratification:

a Recreate the uniform draws with proportional sampling:

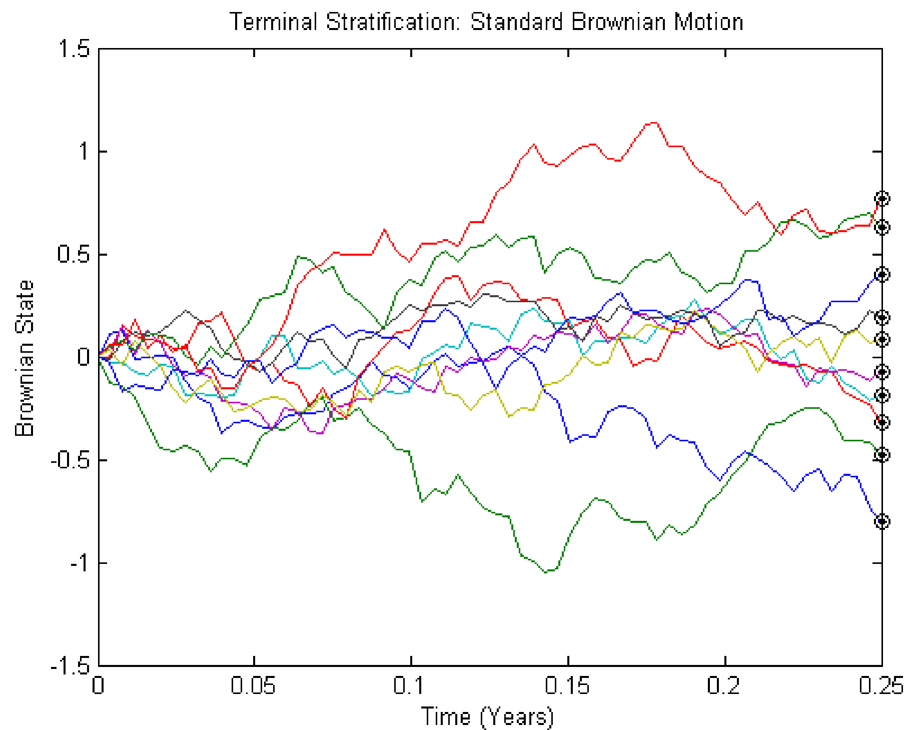
```
rand('twister', 0)
U  = ((1:nPaths)' - 1 + rand(nPaths,1))/nPaths;
```

b Transform them to obtain the terminal values of standard Brownian motion:

```
WT = norminv(U) * sqrt(T);  % Stratified Brownian motion.
```

c Plot the terminal values and output paths on the same figure:

```
plot(sampleTimes, X), hold('on')
xlabel('Time (Years)'), ylabel('Brownian State')
title('Terminal Stratification: Standard Brownian Motion')
plot(T, WT, '. black', T, WT, 'o black')
hold('off')
```



The last value of each sample path (the last row of the output array X) coincides with the corresponding element of the stratified terminal value of the Brownian motion. This occurs because the simulated model and the noise generation function both represent the same standard Brownian motion.

However, you can use the same stratified sampling function to stratify the terminal price of constant-parameter geometric Brownian motion models. In fact, you can use the stratified sampling function to stratify the terminal value of any constant-parameter model driven by Brownian motion if the model's terminal value is a monotonic transformation of the terminal value of the Brownian motion.

To illustrate this, load the SDE_data data set and simulate risk-neutral sample paths of the FTSE 100 index using a geometric Brownian motion (GBM) model with constant parameters:

$$dX_t = rX_t dt + \sigma X_t dW_t$$

where the average Euribor yield represents the risk-free rate of return.

1 Assume that the relevant information derived from the daily data is annualized, and that each calendar year comprises 252 trading days:

```
returns = price2ret(SDE_Data.UK);
sigma   = std(returns) * sqrt(252);
rate    = SDE_Data.Euribor3M;
rate    = mean(360 * log(1 + rate));
```

2 Create the GBM model, assuming the FTSE 100 starts at 100:

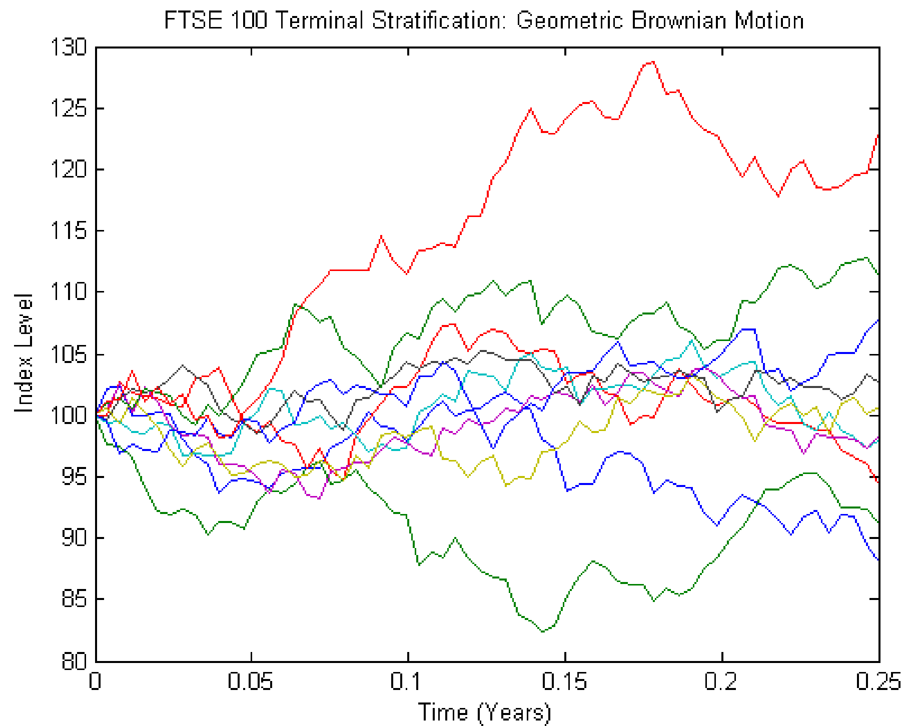
```
obj = gbm(rate, sigma, 'StartState', 100);
```

3 Determine the sample time and simulate the price paths.

In what follows, NSTEPS specifies the number of intermediate time steps within each time increment DeltaTime. Each increment DeltaTime is partitioned into NSTEPS subintervals of length DeltaTime/nSteps each, refining the simulation by evaluating the simulated state vector at NSTEPS–1 intermediate points. This refinement improves accuracy

by allowing the simulation to more closely approximate the underlying continuous-time process without storing the intermediate information:

```
nSteps      = 1;
sampleTimes = cumsum([obj.StartTime ;
dt(ones(nPeriods * nSteps,1))/nSteps]);
z           = stratifiedExample(nPaths, sampleTimes);
randn('state', 10), rand('twister', 0)
[Y, Times]  = obj.simBySolution(nPeriods, 'nTrials, nPaths, ...
  'DeltaTime', dt, 'nSteps', nSteps,  'Z', z);
Y = squeeze(Y);   % Reorder to a 2-D array
plot(Times, Y)
xlabel('Time (Years)'), ylabel('Index Level')
title('FTSE 100 Terminal Stratification: ...
Geometric Brownian Motion')
```



Although the terminal value of the Brownian motion shown in the latter plot is normally distributed, and the terminal price in the previous plot is lognormally distributed, the corresponding paths of each graph are similar.

Creating User-Specified Functions

In this section...

"Evaluating Object Parameters, Noise, and End-of-Period Processing Functions" on page 5-69

"Random Number Generation Functions vs. End-of-Period Processing Functions" on page 5-70

Evaluating Object Parameters, Noise, and End-of-Period Processing Functions

Several examples in this documentation emphasize the evaluation of object parameters as functions accessible by a common interface. In fact, you can evaluate object parameters by passing to them time and state, regardless of whether the underlying user-specified parameter is a function. However, it is helpful to compare the behavior of object parameters that are specified as functions to that of user-specified noise and end-of-period processing functions.

Object parameters that are specified as functions are evaluated in the same way as user-specified random number (noise) generation functions. (For more information, see "Random Number Generation Functions vs. End-of-Period Processing Functions" on page 5-70.) Object parameters that are specified as functions are inputs to model object constructors. User-specified noise and processing functions are *optional* inputs to model object constructors.

Because class constructors offer unique interfaces, and simulation methods of any given model have different implementation details, models often call parameter functions for validation purposes a different number of times, or in a different order, during object creation, simulation, and interpolation.

Therefore, although parameter functions, user-specified noise generation functions, and end-of-period processing functions all share the same interface and are validated at the same initial time and state (`obj.StartTime` and `obj.StartState`), parameter functions are not guaranteed to be invoked only once before simulation as noise generation and end-of-period processing functions are. In fact, parameter functions might not even be invoked the same number of times during a given Monte Carlo simulation process.

In most applications in which you specify parameters as functions, they are simple, deterministic functions of time and/or state. There is no need to count periods, count trials, or otherwise accumulate information or synchronize time.

However, if parameter functions require more sophisticated bookkeeping, the correct way to determine when a simulation has begun (or equivalently, to determine when model validation is complete) is to determine when the input time and/or state differs from the initial time and state (`obj.StartTime` and `obj.StartState`, respectively). Because the input time is a known scalar, detecting a change from the initial time is likely the best choice in most situations. This is a general mechanism that you can apply to any type of user-defined function.

Random Number Generation Functions vs. End-of-Period Processing Functions

It is useful to compare the evaluation rules of user-specified noise generation functions to those of end-of-period processing functions. These functions have the following in common:

- They both share the same general interface, returning a column vector of appropriate length when evaluated at the current time and state:

  $$X_t = f(t, X_t)$$

  $$z_t = Z(t, X_t)$$

- Before simulation, the simulation method itself calls each function is once to validate the size of the output at the initial time and state, `obj.StartTime` and `obj.StartState`, respectively.
- During simulation, the simulation method calls each function the same number of times:`NPERIODS * NSTEPS`.

However, there is an important distinction regarding the timing, between these two types of functions. It is most clearly drawn directly from the generic `SDE` model:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

This equation is expressed in continuous time, but the simulation methods approximate the model in discrete time as:

$$X_{t+\Delta t} = X_t + F(t, X_t)dt + G(t, X_t)\sqrt{\Delta t}Z(t, X_t)$$

where $\Delta t > 0$

is a small (and not necessarily equal) period or time increment into the future. This equation is often referred to as an *Euler approximation*. All functions on the right-hand side are evaluated at the current time and state (t, X_t).

In other words, over the next small time increment, the simulation evolves the state vector based only on information available at the current time and state. In this sense, you can think of the noise function as a beginning-of-period function, or as a function evaluated from the left. This is also true for any user-supplied drift or diffusion function. For more details, see "Evaluating Object Parameters, Noise, and End-of-Period Processing Functions" on page 5-69.

In contrast, user-specified end-of-period processing functions are applied only at the end of each simulation period or time increment. For more information about processing functions, see "Black-Scholes Option Pricing" on page 5-60.

Therefore, all simulation methods evaluate noise generation functions as:

$$z_t = Z(t, X_t)$$

for $t = t_0 + \Delta t, t_0 + 2\Delta t, ..., T$

Yet simulation methods evaluate end-of-period processing functions as:

$$X_t = f(t, X_t)$$

for $t = t_0 + \Delta t, t_0 + 2\Delta t, ..., T$

where t_0 and T are the initial time (taken from the object) and the terminal time (derived from inputs to the simulation method), respectively. These evaluations occur on all sample paths. Therefore, during simulation, noise functions are never evaluated at the final (terminal) time, and end-of-period processing functions are never evaluated at the initial (starting) time.

Managing Memory, Performance, and Solution Accuracy

In this section...

"Managing Memory" on page 5-72

"Enhancing Performance" on page 5-73

"Optimizing Accuracy of Solutions" on page 5-74

Managing Memory

There are two general approaches for managing memory when solving most problems supported by the SDE engine:

- Perform a traditional simulation to simulate the underlying variables of interest, specifically requesting and then manipulating the output arrays.

  This approach is straightforward and the best choice for small or medium-sized problems. Since its outputs are arrays, it is convenient to manipulate simulated results in the MATLAB® matrix-based language. However, as the scale of the problem increases, the benefit of this approach decreases, because the output arrays must store large quantities of possibly extraneous information.

  For example, consider pricing a European option in which the terminal price of the underlying asset is the only value of interest. To ease the memory burden of the traditional approach, reduce the number of simulated periods specified by the required input `NPERIODS` and specify the optional input `NSTEPS`. This enables you to manage memory without sacrificing accuracy (see "Optimizing Accuracy of Solutions" on page 5-74).

  In addition, simulation methods can determine the number of output arguments and allocate memory accordingly. Specifically, all simulation methods support the same output argument list:

  ```
  [Paths, Times, Z]
  ```

  where `Paths` and `Z` can be large, 3-dimensional time-series arrays. However, the underlying noise array is typically unnecessary, and is only stored if requested as an output. In other words, `Z` is stored only at your request; do not request it if you do not need it.

If you need the output noise array `Z` but do not need the `Paths` time-series array, you can avoid storing `Paths` by using the optional input flag `StorePaths`, which all simulation methods support. By default, `Paths` is stored (`StorePaths` = `true`). However, setting `StorePaths` to `false` returns `Paths` as an empty matrix.

- Specify one or more end-of-period processing functions to manage and store only the information of interest, avoiding simulation outputs altogether.

  This approach requires you to specify one or more end-of-period processing functions, and is often the preferred approach for large-scale problems. This approach allows you to avoid simulation outputs altogether. Since no outputs are requested, the 3-dimensional time-series arrays `Paths` and `Z` are not stored.

  This approach often requires more effort, but is far more elegant and allows you to customize tasks and dramatically reduce memory usage.

Enhancing Performance

The following approaches improve performance when solving SDE problems:

- Specifying model parameters as traditional MATLAB arrays and functions, in various combinations. This provides a flexible interface that can support virtually any general nonlinear relationship. However, while functions offer a convenient and elegant solution for many problems, simulations typically run faster when you specify parameters as double-precision vectors or matrices. Thus, it is a good practice to specify model parameters as arrays when possible.
- Using Brownian motion (`BM`) and geometric Brownian motion (`GBM`) models that provide overloaded Euler simulation methods take advantage of separable, constant-parameter models. These specialized methods are exceptionally fast, but are only available to models with constant parameters that are simulated without user-specified end-of-period processing and noise generation functions.
- Replace the simulation of a constant-parameter, univariate model derived from the `SDEDDO` class with that of a diagonal multivariate model, treating the multivariate model as a portfolio of univariate models. This increases the dimensionality of the model and enhances performance by decreasing the effective number of simulation trials.

Note This technique is only applicable to constant-parameter univariate models without user-specified end-of-period processing and noise generation functions.

- Take advantage of the fact that simulation methods are designed to detect the presence of `NaN` (not a number) conditions returned from end-of-period processing functions. A `NaN` represents the result of an undefined numerical calculation, and any subsequent calculation based on a `NaN` produces another `NaN`. This helps improve performance in certain situations. For example, consider simulating paths of the underlier of a knock-out barrier option (an option that becomes worthless as soon as the price of the underlying asset crosses some prescribed barrier). A user-defined end-of-period function could detect a barrier crossing and return a `NaN` to signal early termination of the current trial.

Optimizing Accuracy of Solutions

About Precision and Error

The simulation architecture does not, in general, simulate *exact* solutions to any SDE. Instead, the simulation architecture provides a discrete-time approximation of the underlying continuous-time process, a simulation technique often known as an *Euler approximation*.

In the most general case, a given simulation derives directly from an SDE. Therefore, the simulated discrete-time process approaches the underlying continuous-time process only in the limit as the time increment dt approaches zero. In other words, the simulation architecture places more importance on ensuring that the probability distributions of the discrete-time and continuous-time processes are close, than on the pathwise proximity of the processes.

Before illustrating techniques to improve the approximation of solutions, it is helpful to understand the source of error. Throughout this architecture, all simulation methods assume that model parameters are piece-wise constant over any time interval of length dt. In fact, the methods even evaluate dynamic parameters at the beginning of each time interval and hold them

fixed for the duration of the interval. This sampling approach introduces *discretization error*.

However, there are certain models for which the piece-wise constant approach provides exact solutions:

- "Creating Brownian Motion (BM) Models" on page 5-21 with constant parameters, simulated by Euler approximation (`simByEuler`).
- "Creating Geometric Brownian Motion (GBM) Models" on page 5-23 with constant parameters, simulated by closed-form solution (`simBySolution`).
- "Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 5-26 with constant parameters, simulated by closed-form solution (`simBySolution`)

More generally, you can simulate the exact solutions for these models even if the parameters vary with time, if they vary in a piece-wise constant way such that parameter changes coincide with the specified sampling times. However, such exact coincidence is unlikely; therefore, the previously discussed constant parameter condition is commonly used in practice.

One obvious way to improve accuracy involves sampling the discrete-time process more frequently. This decreases the time increment (dt), causing the sampled process to more closely approximate the underlying continuous-time process. Although decreasing the time increment is universally applicable, however, there is a tradeoff among accuracy, run-time performance, and memory usage.

To manage this tradeoff, specify an optional input argument, `NSTEPS`, for all simulation methods. `NSTEPS` indicates the number of intermediate time steps within each time increment dt, at which the process is sampled but not reported.

It is important and convenient at this point to emphasize the relationship of the inputs `NSTEPS`, `NPERIODS`, and `DeltaTime` to the output vector `Times`, which represents the actual observation times at which the simulated paths are reported.

- NPERIODS, a required input, indicates the number of simulation periods of length DeltaTime, and determines the number of rows in the simulated 3-dimensional Paths time-series array (if an output is requested).
- DeltaTime is optional, and indicates the corresponding NPERIODS-length vector of positive time increments between successive samples. It represents the familiar *dt* found in stochastic differential equations. If DeltaTime is unspecified, the default value of 1 is used.
- NSTEPS is also optional, and is only loosely related to NPERIODS and DeltaTime. NSTEPS specifies the number of intermediate time steps within each time increment DeltaTime.

  Specifically, each time increment DeltaTime is partitioned into NSTEPS subintervals of length DeltaTime/NSTEPS each, and refines the simulation by evaluating the simulated state vector at (NSTEPS - 1) intermediate times. Although the output state vector (if requested) is not reported at these intermediate times, this refinement improves accuracy by causing the simulation to more closely approximate the underlying continuous-time process. If NSTEPS is unspecified, the default is 1 (to indicate no intermediate evaluation).
- The output Times is an NPERIODS + 1-length column vector of observation times associated with the simulated paths. Each element of Times is associated with a corresponding row of Paths.

The following example illustrates this intermediate sampling by comparing the difference between a closed-form solution and a sequence of Euler approximations derived from various values of NSTEPS.

Example: Improving SDE Solution Accuracy by Increasing Sampling of the Discrete-Time Process

Consider a univariate geometric Brownian motion (GBM) model with constant parameters:

$$dX_t = 0.1 X_t dt + 0.4 X_t dW_t$$

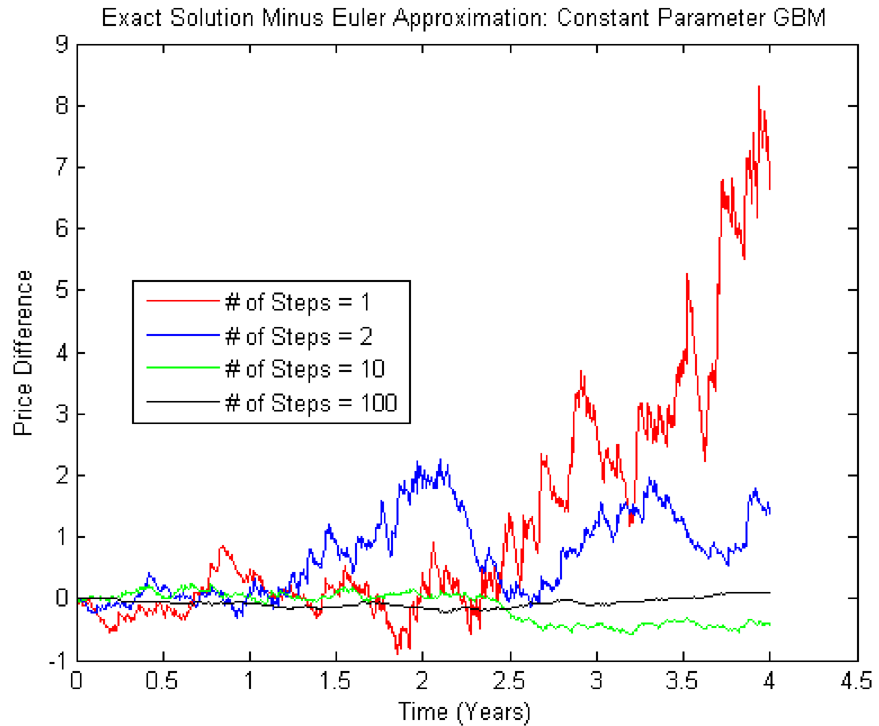
Assume that the expected rate of return and volatility parameters are annualized, and that a calendar year comprises 250 trading days.

1. Simulate approximately four years of univariate prices for both the exact solution and the Euler approximation for various values of NSTEPS:

```
nPeriods  = 1000;
dt        = 1 / 250;
obj   = gbm(0.1, 0.4, 'StartState', 100);
randn('state', 25)
[X,T]   = obj.simBySolution(nPeriods, 'DeltaTime', dt);
randn('state', 25)
[Y,T]   = obj.simByEuler(nPeriods, 'DeltaTime', dt);
clf, plot(T, X - Y, 'red'), hold('on')
randn('state', 25)
[X,T]   = obj.simBySolution(nPeriods, 'DeltaTime',...
    dt, 'nSteps', 2);
randn('state', 25)
[Y,T]   = obj.simByEuler(nPeriods, 'DeltaTime', ...
    dt, 'nSteps', 2);
plot(T, X - Y, 'blue')
randn('state', 25)
[X,T]   = obj.simBySolution(nPeriods, 'DeltaTime', ...
    dt, 'nSteps', 10);
randn('state', 25)
[Y,T]   = obj.simByEuler(nPeriods, 'DeltaTime', ...
    dt, 'nSteps', 10);
plot(T, X - Y, 'green')
randn('state', 25)
[X,T]   = obj.simBySolution(nPeriods, 'DeltaTime', ...
    dt, 'nSteps', 100);
randn('state', 25)
[Y,T]   = obj.simByEuler(nPeriods, 'DeltaTime', ...
    dt, 'nSteps', 100);
plot(T, X - Y, 'black'), hold('off')
```

2 Compare the error (the difference between the exact solution and the Euler approximation) graphically:

```
xlabel('Time (Years)'), ylabel('Price Difference')
title('Exact Solution Minus Euler Approximation: ...
  Constant Parameter GBM')
legend({'# of Steps = 1'  '# of Steps = 2'    ...
       '# of Steps = 10' '# of Steps = 100'}, ...
  'Location', 'Best')
hold('off')
```



As expected, the simulation error decreases as the number of intermediate time steps increases. Because the intermediate states are not reported, all simulated time series have the same number of observations regardless of the actual value of `NSTEPS`:

```
whos T X Y
  Name         Size          Bytes  Class    Attributes

  T           1001x1         8008  double
  X           1001x1         8008  double
  Y           1001x1         8008  double
```

Furthermore, since the previously simulated exact solutions are correct for any number of intermediate time steps, additional computations are not needed for this example. In fact, this assessment is generally correct.

The exact solutions are sampled at intermediate times to ensure that the simulation uses the same sequence of Gaussian random variates in the same order. Without this assurance, there is no way to compare simulated prices on a pathwise basis. However, there might be valid reasons for sampling exact solutions at closely spaced intervals, such as pricing path-dependent options.

Estimation

Maximum Likelihood Estimation (p. 6-2)	Performing maximum likelihood estimation using `garchfit`
Initial Parameter Estimates (p. 6-4)	How to use user-supplied and automatically generated initial parameter estimates
Presample Observations (p. 6-12)	Computing presample data for conditional mean and supported variance models
Termination Criteria and Optimization Results (p. 6-15)	Optimization parameters that affect the optimization process
Examples: Specifying Your Own Presample Data to Estimate ARMA(R,M) Parameters (p. 6-21)	Illustrates presample data, transient effects, and lower bound constraints.

Maximum Likelihood Estimation

This section explains how the `garchfit` estimation engine uses maximum likelihood to estimate the parameters needed to fit the specified models to a given univariate return series.

Given an observed univariate time series and the conditional mean and variance models described in “Conditional Mean and Variance Models” on page 2-7, `garchfit` does the following:

- Infers the innovations (residuals) from the return series.
- Estimates, by maximum likelihood, the parameters needed to fit the specified models to the return series.

Given a vector of initial parameter estimates, as described in “Initial Parameter Estimates” on page 6-4, the `garchfit` function calls the Optimization Toolbox™ `fmincon` function to perform constrained optimization of a scalar function of several variables; that is, the log-likelihood function. This technique is called *constrained nonlinear optimization* or *nonlinear programming*. In turn, `fmincon` calls the appropriate log-likelihood objective function to estimate the model parameters using maximum likelihood estimation (MLE).

The chosen log-likelihood objective function proceeds as follows:

- Given the vector of current parameter values and the observed data `Series`, the log-likelihood function infers the process innovations (residuals) by *inverse filtering*. This inference operation rearranges the conditional mean equation to solve for the current innovation ε_t:

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

  This equation is a whitening filter, transforming a (possibly) correlated process y_t into an uncorrelated white noise process ε_t.
- The log-likelihood function then uses the inferred innovations ε_t to infer the corresponding conditional variances σ_t^2 via recursive substitution into

the previous model-dependent conditional variance equations Equation 2-4, Equation 2-5, and Equation 2-6.

- Finally, the function uses the inferred innovations and conditional variances to evaluate the appropriate log-likelihood objective function. If ε_t is Gaussian, the log-likelihood function is

$$LLF = -\frac{T}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{T}\log\sigma_t^2 - \frac{1}{2}\sum_{t=1}^{T}\varepsilon_t^2\sigma_t^2 \tag{6-1}$$

If ε_t is Student's t, the log-likelihood function is

$$LLF = T\log\{\frac{\Gamma[\frac{(\nu+1)}{2}]}{\pi^{\frac{1}{2}}\Gamma(\frac{\nu}{2})}(\nu-2)^{-\frac{1}{2}}\} - \frac{1}{2}\sum_{t=1}^{T}\log\sigma_t^2 - \frac{\nu+1}{2}\sum_{t=1}^{T}\log[1+\frac{\varepsilon_t^2}{\sigma_t^2(\nu-2)}] \tag{6-2}$$

where T is the sample size, that is, the number of rows in the series $\{y_t\}$. The degrees of freedom ν must be greater than 2.

The conditional mean equation, Equation 2-2, and the conditional variance equations, Equation 2-4, Equation 2-5, and Equation 2-6, are recursive, and generally require presample observations to initiate inverse filtering. For this reason, the objective functions shown here are referred to as *conditional log-likelihood functions*. Evaluation of the log-likelihood function is conditioned, or based, on a set of presample observations. For more information about the methods used to specify these presample observations, see "Presample Observations" on page 6-12.

The iterative numerical optimization repeats the previous three steps until it satisfies suitable termination criteria. For more information, see "Termination Criteria and Optimization Results" on page 6-15 .

Initial Parameter Estimates

In this section...

"User-Specified Initial Estimates" on page 6-4

"Automatically Generated Initial Estimates" on page 6-6

"Parameter Bounds" on page 6-10

User-Specified Initial Estimates

The constrained nonlinear optimizer, `fmincon`, requires a vector of initial parameter estimates. The `garchfit` function computes initial parameter estimates if you provide none. At times, however, it might be helpful to compute and specify your own initial guesses to avoid convergence problems. You can specify complete initial estimates for either or both the conditional mean equation and the conditional variance equation.

For the conditional mean estimates to be complete, specify the following parameters:

- `C`
- `AR`
- `MA`

These must be consistent with the orders you specified for `R` and `M`. The length of `AR` must be `R`, and the length of `MA` must be `M`. If you provide a regression matrix X, you must also specify the `Regress` parameter. `C`, `AR`, `MA`, and `Regress` correspond respectively to C, Φ_j, θ_i, and β_k in Equation 2-2.

Note Set `C = NaN` (Not-a-Number) to remove the constant C from the conditional mean model. This fixes `C = 0` without providing initial parameter estimates for the remaining parameters. In this case, the value of `FixC` has no effect.

For the conditional variance estimates to be complete, specify these specification structure parameters for all conditional variance models:

- K
- GARCH
- ARCH

These must be consistent with the orders you specified for P and Q. The length of GARCH must be P, and the length of ARCH must be Q. You must also specify the Leverage parameter for GJR and EGARCH conditional variance models. The parameters K, GARCH, ARCH, and Leverage correspond respectively to κ, G_i, A_j, andL_j in Equation 2-4, Equation 2-5, and Equation 2-6.

You can use garchset to create the necessary specification structure, Spec, or you can modify the Coeff structure returned by a previous call to garchfit.

If you provide initial parameter estimates for a model equation, you must provide *all* the estimated constants and coefficients consistent with the specified model orders. For example, for an ARMA(2,2) model with no regression matrix, you must specify the parameters C, AR, and MA. If you specify only MA, the specification is *incomplete*, and garchfit ignores the MA you specified and automatically generates all the requisite initial estimates.

The following specification structure provides C and AR as initial parameter estimates, but does not provide MA, even though M = 1. In this case, garchfit ignores the C and AR fields, computes initial parameter estimates, and overwrites existing parameters in the incomplete conditional mean specification.

```
spec = garchset('R',1,'M',1,'C',0,'AR',0.5,...
                'P',1,'Q',1,'K',0.0005,'GARCH',0.8,'ARCH',0.1)
spec =

         Comment: 'Mean: ARMAX(1,1,?); Variance: GARCH(1,1)'
    Distribution: 'Gaussian'
               R: 1
               M: 1
               C: 0
              AR: 0.5000
              MA: []
   VarianceModel: 'GARCH'
               P: 1
```

```
    Q: 1
    K: 5.0000e-004
GARCH: 0.8000
 ARCH: 0.1000
```

However, the structure explicitly sets all fields in the conditional variance model. Therefore, `garchfit` uses the specified values of `K`, `GARCH`, and `ARCH` as initial estimates, subject to further refinement.

Automatically Generated Initial Estimates

`garchfit` automatically generates initial estimates if you provide incomplete or no initial coefficient estimates for a conditional mean or variance model. It first estimates the conditional mean parameters as needed, and then estimates the conditional variance parameters as needed. Again, `garchfit` ignores incomplete initial estimates. It estimates initial conditional mean parameters using standard statistical time-series techniques, dependent upon the parametric form of the conditional mean equation.

Conditional Mean Models Without a Regression Component

ARMA Models. Initial parameter estimates of general ARMA(R,M) conditional mean models are estimated by the three-step method outlined in Box, Jenkins, and Reinsel [10], Appendix A6.2.

- `garchfit` estimates the autoregressive coefficients, Φ_j, by computing the sample autocovariance matrix and solving the Yule-Walker equations.
- Using these estimated coefficients, `garchfit` filters the observed `Series` to obtain a pure moving average process.
- `garchfit` computes the autocovariance sequence of the moving average process, and uses it to iteratively estimate the moving average coefficients, θ_i. This also provides an estimate of the unconditional variance of the innovations.

Conditional Mean Models with a Regression Component

ARX Models (No Moving Average Terms Allowed). Ordinary least squares regression generates initial estimates of the autoregressive coefficients, Φ_j, and the regression coefficients, β_k, of the explanatory data matrix X.

For more information, see Chapter 8, “Regression Components”.

ARMAX Models (Moving Average Terms Included). Initial parameter estimation of the general ARMAX conditional mean models requires two steps:

- `garchfit` estimates an ARX model by ordinary least squares.
- `garchfit` estimates an MA(M) = ARMA(0,M) model, as described in “Conditional Mean Models Without a Regression Component” on page 6-6.

Conditional Variance Models

Unlike conditional mean parameters, initial estimates of conditional variance parameters are based on empirical analysis of financial time series. The approach is dependent upon the conditional variance model you select.

GARCH(P,Q) Models. For GARCH models, `garchfit` assumes that the sum of the G_i, (i = 1, ...,P) and the A_j, (j = 1, ...,Q is close to 1. Specifically, for a general GARCH(P,Q) model (Equation 2-4), `garchfit` assumes that

$$G_1 + \dots + G_P + A_1 + \dots + A_Q = 0.9$$

If $P > 0$ (lagged conditional variances are included), then `garchfit` equally allocates 0.85 out of the available 0.90 to the P GARCH coefficients. It allocates the remaining 0.05 equally among the Q ARCH coefficients.

$P = 0$ specifies an ARCH(Q) model in which `garchfit` allocates 0.90 equally to the Q ARCH terms.

The following examples clarify this approach.

Initial estimates of the GARCH(1,1) model are expressed as follows:

$$\sigma_t^2 = \kappa + 0.85\sigma_{t-1}^2 + 0.05\varepsilon_{t-1}^2$$

A GARCH(2,1) model is initially expressed as:

$$\sigma_t^2 = \kappa + 0.425\sigma_{t-1}^2 + 0.425\sigma_{t-2}^2 + 0.05\varepsilon_{t-1}^2$$

An ARCH(1) model is initially expressed as:

$$\sigma_t^2 = \kappa + 0.9\varepsilon_{t-1}^2$$

An ARCH(2) model is initially expressed as:

$$\sigma_t^2 = \kappa + 0.45\varepsilon_{t-1}^2 + 0.45\varepsilon_{t-2}^2$$

Finally, `garchfit` estimates the constant κ of the conditional variance model by first estimating the unconditional, or time-independent, variance of $\{\varepsilon_t\}$:

$$\sigma^2 = \frac{1}{T}\sum_{t=1}^{T}\varepsilon_t^2$$

In terms of the parameters, this can also be expressed as:

$$\sigma^2 = \frac{\kappa}{1 - \sum_{t=1}^{P} G_i - \sum_{j=1}^{Q} A_j} = \frac{\kappa}{1 - (0.85 + 0.05)}$$

and so

$$\kappa = \sigma^2(1 - (0.85 + 0.05)) = 0.1\sigma^2$$

GJR(P,Q) Models. `garchfit` treats a GJR(P,Q) model, described in Equation 2-5, as an extension of an equivalent GARCH(P,Q) model with zero leverage terms. Thus, initial parameter estimates of GJR models are identical to those of equivalent order GARCH models (see "GARCH(P,Q) Models" on page 6-7), with the additional assumption that all leverage terms are zero:$L_i = 0,\ 1 \le i \le Q$.

EGARCH(P,Q) Models. For EGARCH models, `garchfit` assumes that the sum of the G_i, ($i = 1, ..., P$ is 0.9, and the sum of A_j, ($j = 1, ..., Q$ is 0.2. Specifically, for a general EGARCH(P,Q) model (Equation 2-6), `garchfit` assumes that:

$$G_1 + G_2 + \ldots + G_P = 0.9$$

$$A_1 + A_2 + \ldots + A_Q = 0.2$$

and

$$L_i = 0,\ 1 \le i \le Q$$

If $P > 0$ (lagged conditional variances are included), then `garchfit` equally allocates the available weight of 0.9 to the P GARCH coefficients. It equally allocates the available weight of 0.2 to the Q ARCH coefficients.

In EGARCH models, the standardized innovation, z_t, serves as the forcing variable for both the conditional variance and the error. Thus, theG_i terms captured volatility clustering (that is, persistence). In other words, EGARCH models make no allowance for the relationship between theG_i and A_j coefficients regarding initial parameter estimates. Because of this, EGARCH(0,Q) models ignore the persistence effect commonly associated with financial returns, and are unusual. Some examples clarify the approach.

The EGARCH(1,1) model is by far the most common, and initial estimates are expressed as:

$$\log\sigma_t^2 = \kappa + 0.9\log\sigma_{t-1}^2 + 0.2[\,|z_{t-1}| - E(|z_{t-1}|)]$$

Initial estimates for an EGARCH(2,2) model are expressed as

$$\log \sigma_t^2 = \kappa + 0.45 \log \sigma_{t-1}^2 + 0.45 \log \sigma_{t-2}^2 + 0.1[\,|z_{t-1}| - E(|z_{t-1}|)]$$
$$+0.1[\,|z_{t-2}| - E(|z_{t-2}|)]$$

An EGARCH(0,1) model would be initially expressed as

$$\log \sigma_t^2 = \kappa + 0.2[\,|z_{t-1}| - E(|z_{t-1}|)]$$

As you can see, initial parameter estimates for EGARCH models are most effective when $P > 0$.

Finally, you can estimate the constant κ of an EGARCH conditional variance model by noting the approximate relationship between the unconditional variance of the innovations process, σ^2, and the G_i parameters of an EGARCH(1,1) model:

$$\kappa = (1 - G_1)\log \sigma^2 = (1 - 0.9)\log \sigma^2 = 0.1 \log \sigma^2$$

Parameter Bounds

`garchfit` bounds some model parameters to provide stability in the optimization process. See the example "Active Lower Bound Constraint" on page 6-30 for more information on overriding these bounds in the unlikely event they become active.

Conditional Mean Model

For the conditional mean model, Equation 2-2, `garchfit` bounds the conditional mean constant C and the conditional mean regression coefficients β_k, if any, in the interval [-10,10]. However, if the coefficient estimates that you specify or that `garchfit` generates are outside this interval, `garchfit` sets the appropriate lower or upper bound equal to the estimated coefficient.

GARCH(P,Q) and GJR(P,Q) Conditional Variance Models

For GARCH(P,Q) and GJR(P,Q) conditional variance models, represented by Equation 2-3 and Equation 2-4, `garchfit` uses 5 as an upper bound for the conditional variance constant κ. If the initial estimate is greater than 5, `garchfit` uses the estimated value as the upper bound.

EGARCH(P,Q) Conditional Variance Model

For EGARCH(P,Q) conditional variance models, represented by Equation 2-5, `garchfit` places arbitrary bounds on the conditional variance constant κ, such that $-5 \le \kappa \le 5$. If the magnitude of the initial estimate is greater than 5, `garchfit` adjusts the bounds accordingly.

Presample Observations

In this section...

"Calculating Presample Data" on page 6-12
"User-Specified Presample Observations" on page 6-12
"Automatically Generated Presample Observations" on page 6-13

Calculating Presample Data

This section shows how `garchfit` automatically generates presample data for the conditional mean model and each of the supported conditional variance models. It also shows how to specify your own presample data. "Maximum Likelihood Estimation" on page 6-2 discusses presample data required to initiate inverse filtering and evaluate the conditional log-likelihood objective function.

User-Specified Presample Observations

Use the time-series column vector inputs `PreInnovations`, `PreSigmas`, and `PreSeries` to explicitly specify all required presample data. The following table summarizes the minimum number of rows required to successfully initiate the optimization process.

Garchfit Input Argument	Minimum Number of Rows GARCH(P,Q), GJR(P,Q)	EGARCH(P,Q)
`PreInnovations`	`max(M,Q)`	`max(M,Q)`
`PreSigmas`	`P`	`max(P,Q)`
`PreSeries`	`R`	`R`

If you specify at least one, but fewer than three, sets of presample data, `garchfit` does not attempt to derive presample observations for those you omit. When specifying your own presample data, include all data required for the given conditional mean and variance models. See the example "Specifying Presample Data" on page 6-21.

Automatically Generated Presample Observations

If you do not specify presample data, `garchfit` automatically generates the required presample data.

Conditional Mean Models

For conditional mean models with an autoregressive component, `garchfit` assigns the `R` required presample observations (`PreSeries`) of y_t, the sample mean of `Series`. For models with a moving-average component, it sets the `M` required presample observations (`PreInnovations`) of ε_t to their expected value of zero. With this presample data, `garchfit` can infer the entire sequence of innovations for any general ARMAX conditional mean model, regardless of the conditional variance model you select.

`garchfit` attempts to eliminate the effect of transients in the presample data it generates. This effect parallels that in the simulation process described in "Automatically Generating Presample Data" on page 4-7. For an example of transient effects in the estimation process, see "Presample Data and Transient Effects" on page 6-24.

GARCH(P,Q) Models

Once `garchfit` computes the innovations, it assigns the sample mean of the squared innovations

$$\sigma^2 = \frac{1}{T}\sum_{t=1}^{T}\varepsilon_t^2$$

to the required `P` and `Q` presample observations of σ_t^2 and ε_t^2, respectively. See Hamilton [22] and Bollerslev [6].

GJR(P,Q) Models

`garchfit` also assigns the average squared innovation to all required presample observations of σ_t^2 and ε_t^2. In addition, `garchfit` weights the `Q` presample observations of ε_t^2 associated with the leverage terms by 0.5 (that is, the probability of a negative past residual).

EGARCH(P,Q) Models

`garchfit` also assigns the average squared innovation to all `P` presample observations of σ_t^2. In addition, it sets all `Q` presample observations of the standardized innovations $z_t = (\frac{\varepsilon_t}{\sigma_t})$ to zero and $| z_t | = (\frac{| \varepsilon_t |}{\sigma_t})$ to the mean absolute deviation. This has the effect of setting all `Q` presample ARCH and leverage terms to zero.

Termination Criteria and Optimization Results

In this section...

"Optimization Parameters" on page 6-15

"MaxIter and MaxFunEvals" on page 6-15

"TolCon, TolFun, and TolX" on page 6-16

"Convergence" on page 6-17

"Optimization Results" on page 6-17

"Constraint Violation Tolerance" on page 6-18

Optimization Parameters

Listed below, in order of importance, are several fields in the specification structure that allow you to influence the optimization process.

`TolCon`	Termination tolerance on the constraint violation
`TolFun`	Termination tolerance on the function value
`TolX`	Termination tolerance on the parameter estimates
`MaxFunEvals`	Maximum number of function evaluations allowed
`MaxIter`	Maximum number of iterations allowed

For more information about these parameters, see:

- "Tolerances and Stopping Criteria" in the Optimization Toolbox™ documentation.
- The `garchset` function reference page.

MaxIter and MaxFunEvals

`MaxIter` is the maximum number of iterations allowed in the estimation process. Each iteration involves an optimization phase in which `garchfit` 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 that 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, so the default value of `MaxFunEvals` for the default model 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 that the estimation terminated prematurely.

The fields `MaxFunEvals` and `MaxIter` are purely mechanical in nature. Although you may encounter situations in which `MaxFunEvals` or `MaxIter` is reached, this is rather uncommon. Increasing `MaxFunEvals` or `MaxIter` may allow successful convergence. However, reaching `MaxFunEvals` or `MaxIter` is usually an indication that your model poorly describes the data. In particular, it often indicates that the model is too complicated. Finally, although `MaxFunEvals` and `MaxIter` can cause the function to stop before a solution is found, they do not affect the solution once it is found.

TolCon, TolFun, and TolX

The fields `TolCon`, `TolFun`, and `TolX` are tolerance-related parameters. They directly influence how and when convergence is achieved, and can also affect the solution.

- `TolCon` is the termination tolerance placed on constraint violations. It represents the maximum value by which parameter estimates can violate a constraint and still allow successful convergence. For information about these constraint violations, see "Conditional Mean and Variance Models" on page 2-7.
- `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`. For more information, see "Optimization Results" on page 6-17.
- `TolX` is the termination tolerance placed on the estimated parameter values. Like `TolFun`, successful convergence occurs when the parameter

values change by less than TolX. For more information, see "Optimization Results" on page 6-17.

Convergence

TolFun, and TolX have the same default value, 1e-006. The TolCon default is 1e-007. If the estimation shows little or no progress, or shows progress but stops early, increase one or more of these parameter values. For example, increasing the values from 1e-006 to 1e-004 may allow the estimation to converge. If the estimation appears to converge to a suboptimal solution, decrease one or more of these parameter values. Decreasing the values from 1e-006 to 1e-007 may provide more accurate parameter estimates.

Note You can avoid many convergence difficulties by performing a pre-fit analysis. "Example: Analysis and Estimation Using the Default Model" on page 2-16 describes graphical techniques, such as plotting the return series, and examining the ACF and PACF. It also discusses some preliminary tests, including Engle's ARCH test and the *Q*-test. Chapter 10, "Model Selection and Analysis" discusses other tests to help you determine the appropriateness of a specific GARCH model. It also explains how equality constraints can help you assess parameter significance. "Limitations of GARCH Modeling" on page 1-4 mentions some limitations of GARCH models that could affect convergence.

Optimization Results

Unlike MaxIter and MaxFunEvals, the tolerance fields TolCon, TolFun, and TolX affect optimization results. (See "TolCon, TolFun, and TolX" on page 6-16.) Assuming that you have selected iterative display, a message like the following appears upon successful termination:

```
Optimization terminated successfully:
Magnitude of directional derivative in search direction
less than 2*options.TolFun and maximum constraint violation
is less than options.TolCon

Optimization terminated successfully:
Search direction less than 2*options.TolX and
maximum constraint violation is less than options.TolCon
```

```
Optimization terminated successfully:
First-order optimality measure less than options.TolFun and
maximum constraint violation is less than options.TolCon
```

Increasing `TolFun` or `TolX` from the default of `1e-6` to, for example, `1e-5`, relaxes one or both of the first two termination criteria. This often results in a less accurate solution. Similarly, decreasing `TolFun` or `TolX` to, for example, `1e-7` restricts one or both of the first two termination criteria. This often results in a more accurate solution, but may also require more iterations. However, experience has shown that the default value of `1e-6` for `TolFun` and `TolX` is almost always sufficient. Changing these values is unlikely to significantly affect the estimation results for GARCH modeling. Thus, it is recommended that you accept the default values for `TolFun` and `TolX`.

The default value of `TolCon` is `1e-7`. Changing the value of `TolCon` can significantly affect the solution in situations in which a constraint is active. `TolCon` is the most important optimization-related field for the GARCH Toolbox™ software. Additional discussion of its significance and use is helpful.

When `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 [22], page 142.) This is because statistical inference relies on the log-likelihood function's 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.

Constraint Violation Tolerance

At each step in the optimization process, `garchfit` evaluates the constraints described in "Conditional Mean and Variance Models" on page 2-7 against the current intermediate solution vector. For each user-specified equality constraint, it determines whether there is a violation whose absolute value is greater than `TolCon`. For each inequality constraint (including lower and upper bounds), it determines whether the inequality is violated by more than the value of `TolCon`. If either the `TolFun` or `TolX` exit condition is satisfied, and if the maximum of any violations is less than the value of `TolCon`, then the optimization terminates successfully. (See "TolCon, TolFun, and TolX" on page 6-16.)

Strict Inequality Constraints

The Optimization Toolbox `fmincon` numerical optimizer defines inequality constraints as a less than or equal to condition. However, many GARCH Toolbox inequality constraints are strict inequalities that specifically exclude exact equality. For this reason, the GARCH Toolbox interpretation of `TolCon` differs from the Optimization Toolbox interpretation.

`TolCon` applies to both strict inequalities and those that are not strict, but `garchfit` provides special handling for strict inequalities. Specifically, `garchfit` associates each strict inequality constraint with its theoretical bound, or limit. However, to avoid the possibility of violating strict inequality constraints, `garchfit` defines the *actual bound* for each such constraint as the theoretical bound offset by `2*TolCon`. The optimization can successfully terminate if the actual bound is violated by as much as `TolCon`. Consequently, any given strict inequality constraint is allowed to approach its theoretical bound to within `TolCon`.

Single Parameter Strict Inequality Constraints

It is possible for an estimate of a strict inequality constraint that involves a single parameter to terminate a distance `TolCon` from its theoretical bound. However, experience has shown that this is unlikely. Examples of such constraints are:

- The conditional variance constant $\kappa > 0$ for the GARCH(P,Q) and GJR(P,Q) models
- The degrees of freedom $\nu > 2$ for the Student's t distribution

Typically, when the lower or upper bound of such a single-parameter inequality constraint is active, the estimate remains `2*TolCon` from the bound.

It is unlikely that an estimate of a single parameter constraint will terminate a distance `TolCon` from its theoretical bound. However, the `garchfit` approach for handling strict inequalities still allows for this condition.

As an illustration, assume `TolCon = 1e-7` (its default value), and consider the default GARCH(1,1) model:

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

with constraints

$\kappa > 0$

$G_1 + A_1 < 1$

$G_1 \geq 0$

$A_1 \geq 0.$

When the lower bound constraint $\kappa > 0$ is active, the estimated value of κ is typically

$\kappa = 2e^{-7}$ = 2*`TolCon`.

Relaxing Constraint Tolerance Limits

Experience has shown that relaxing `TolCon` is more likely to remove an active constraint in some cases than in others. For inequality constraints with a single parameter, such as those discussed in “Single Parameter Strict Inequality Constraints” on page 6-19, decreasing `TolCon` may relax the constraint such that it is no longer active. The example “Active Lower Bound Constraint” on page 6-30 explains how to identify such a condition by examining the summary output structure.

This is not generally true for linear inequality constraints with multiple parameters. An example is $G_1 + A_1 < 1$. When this constraint is active, the estimated values of G_1 and A_1 are typically such that $G_1 + A_1$ = 0.9999999 = 1.0 – `TolCon`. Decreasing `TolCon` to, say, `1e-8`, allows $G_1 + A_1$ to approach 1.0 more closely, but the linear inequality constraint is likely to remain active.

Examples: Specifying Your Own Presample Data to Estimate ARMA(R,M) Parameters

In this section...

"Specifying Presample Data" on page 6-21

"Presample Data and Transient Effects" on page 6-24

"Alternative Technique for Estimating ARMA(R,M) Parameters" on page 6-30

"Active Lower Bound Constraint" on page 6-30

"Determining Convergence Status" on page 6-34

Specifying Presample Data

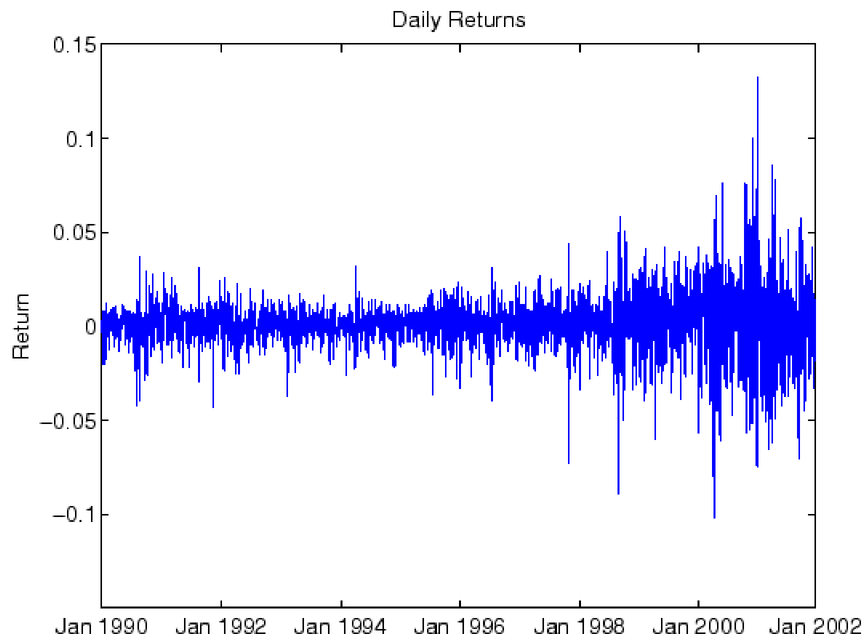
This example shows you how to specify your own presample data to initiate the estimation process. It highlights the formal column-oriented nature of the presample time-series inputs.

1. Load the nasdaq data set and convert prices to returns:

```
load garchdata
nasdaq = price2ret(NASDAQ);
```

2. Segment the NASDAQ data to compare estimation results from a relatively stable period to those from a period of relatively high volatility:

```
plot(nasdaq)
axis([0 length(nasdaq) -0.15 0.15])
set(gca,'XTick',[1 507 1014 1518 2025 2529 3027])
set(gca,'XTickLabel',{'Jan 1990' 'Jan 1992' 'Jan 1994' ...
    'Jan 1996' 'Jan 1998' 'Jan 2000' 'Jan 2002'})
ylabel('Return')
title('Daily Returns')
```



The NASDAQ returns show a distinct increase in volatility starting, approximately, in December 1997. This is roughly the 2000th observation.

3 Create a specification structure to model the NASDAQ returns as an MA(1) process with GJR(1,1) residuals:

```
spec = garchset('VarianceModel','GJR','M',1,'P',1,'Q',1,...
       'Display','off');
```

4 Estimate the parameters, standard errors, and inferred residuals and standard deviations using the first 2000 observations, allowing `garchfit` to automatically generate the necessary presample observations. Then display the estimated coefficients and errors.

```
[coeff,errors,LLF,eFit,sFit] = garchfit(spec,nasdaq(1:2000));
garchdisp(coeff,errors)

  Mean: ARMAX(0,1,0); Variance: GJR(1,1)

  Conditional Probability Distribution: Gaussian
```

```
Number of Model Parameters Estimated: 6

                                  Standard          T
  Parameter       Value             Error       Statistic
 -----------   -----------     ------------   -----------
          C    0.00056403      0.00023455         2.4048
      MA(1)    0.25006         0.024165          10.3480
          K    1.1907e-005     1.528e-006         7.7931
   GARCH(1)    0.69447         0.033664          20.6295
    ARCH(1)    0.024937        0.017695           1.4093
Leverage(1)    0.24541         0.030517           8.0420
```

5 This conditional mean model has no regression component. Therefore, you can obtain the same estimation results by calling `garchfit` with an empty regression matrix, `X = []`, as a placeholder for the third input:

```
[coeff,errors,LLF,eFit,sFit] = garchfit(spec,...
       nasdaq(1:2000),[]);
```

6 Specify your own presample data by specifying additional inputs. Because the inputs `PreInnovations`, `PreSigmas`, and `PreSeries` represent time series in a formal sense, provide required presample data in the form of column vectors of sufficient length.

From the table in “Presample Observations” on page 6-12:

- The length of `PreInnovations` must be at least `max(M,Q) = 1`.
- The length of `PreSigmas` must be at least `P = 1`.
- and `PreSeries` can be empty or unspecified altogether because `R = 0`.

Estimate the same model from the later high-volatility period, using the inferred residuals and standard deviations from the previous period as the presample data:

```
[coeff,errors] = garchfit(spec,nasdaq(2001:end),[],eFit,sFit);
garchdisp(coeff, errors)

  Mean: ARMAX(0,1,0); Variance: GJR(1,1)

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 6
```

```
                              Standard         T
  Parameter       Value        Error       Statistic
 -----------   -----------   ------------   -----------
           C    0.00065398    0.00060488        1.0812
       MA(1)    0.012699      0.035131          0.3615
           K    1.7845e-005   3.9153e-006       4.5578
    GARCH(1)    0.85799       0.026246         32.6906
     ARCH(1)    0.016147      0.022595          0.7146
 Leverage(1)    0.17433       0.033234          5.2455
```

Comparing the estimation results from the two periods reveals a marked difference. The last input, `PreSeries`, is not needed and is left unspecified.

7 Since the example uses only the most recent observations of `PreInnovations`, `PreSigmas`, and `PreSeries`, any of the following calls to `garchfit` produce identical estimation results:

```
[coeff,errors] = garchfit(spec,nasdaq(2001:end),[],...
       eFit(end),sFit(end));
[coeff,errors] = garchfit(spec,nasdaq(2001:end),[],...
       eFit(end),sFit(end),nasdaq(1:2000));
[coeff,errors] = garchfit(spec,nasdaq(2001:end),...
       [],eFit,sFit,nasdaq(1999:2000));
```

The first equivalent call passes in the minimum required presample observations of past residuals and standard deviations. In this case, it passes the last inferred observation of each. The last two equivalent calls specify an unnecessary presample return series, which `garchfit` ignores.

If, for example, the original specification included an AR(2) model (that is, $R = 2$), then at least the last two NASDAQ returns are needed to initiate estimation. In this case, the last two calls to `garchfit` would produce identical results for conditional mean models with AR components up to 2nd order.

Presample Data and Transient Effects

This example shows how to:

1 Simulate a return series, `yTrue`.

2 Use the garchinfer function to infer $\{\varepsilon_t\}$ and $\{\sigma_t\}$ from the simulated return series.

First, you use automatically generated presample data to infer the approximate residuals and conditional standard deviation processes. You then use explicitly specified presample data to infer the exact residuals and conditional standard deviation processes. You then finally compare the approximate conditional standard deviation processes with the exact conditional standard deviations processes, to reveal the effect of transients in the approximate results. The effect of transients in the estimation, or inference, process parallels that in the simulations process described in "Automatically Generating Presample Data" on page 4-7.

To avoid introducing differences as a result of the optimization, this example uses garchinfer rather than garchfit. garchsim uses an ARMA model as a linear filter to transform an uncorrelated input innovations process $\{\varepsilon_t\}$ into a correlated output returns process $\{y_t\}$. garchinfer (and alternatively, garchfit) reverses this process by inferring innovations $\{\varepsilon_t\}$ and standard deviation $\{\sigma_t\}$ processes from the observations in $\{y_t\}$.

1 Specify a time series as an AR(2) conditional mean model and GARCH(1,2) conditional variance model:

```
spec = garchset('C',0,'AR',[0.5 -0.8],'K',0.0002,...
                'GARCH',0.8,'ARCH',[0.1 0.05])
spec = 

          Comment: 'Mean: ARMAX(2,0,?); Variance: GARCH(1,2)'
     Distribution: 'Gaussian'
                R: 2
                C: 0
               AR: [0.5000 -0.8000]
    VarianceModel: 'GARCH'
                P: 1
                Q: 2
                K: 2.0000e-004
            GARCH: 0.8000
             ARCH: [0.1000 0.0500]
```

Note This is an elaborate specification, typically unwarranted for a real-world financial time series, and is meant for illustrative purposes only.

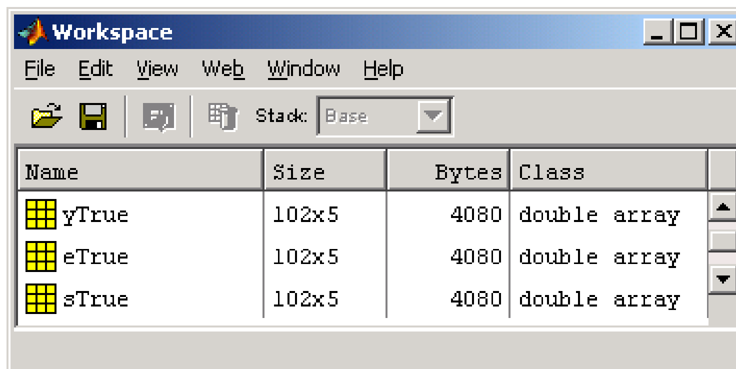
---

2 Simulate 102 observations for each of 5 realizations and reserve the first 2 rows of observations for the presample data needed by `garchinfer` in step 4. The table in “Running Simulations With User-Specified Presample Data” on page 4-13 shows that:

- The `PreInnovations` array must have at least `max(M,Q) = 2` rows.
- `PreSigmas` must have at least `P = 1` row.
- `PreSeries` must have at least `R = 2` rows.

Add the initial state = 0 as a trailing input argument:

```
randn('state',0);
rand('twister',0);
[eTrue,sTrue,yTrue] = garchsim(spec,102,5);
```



3 Call `garchinfer` without explicit presample data, using observations 3 and beyond as the observed return series input. This infers the approximate residuals and conditional standard deviations based on the default presample data inference approach:

```
[eApprox,sApprox] = garchinfer(spec,yTrue(3:end,:));
```

Workspace

File Edit View Web Window Help

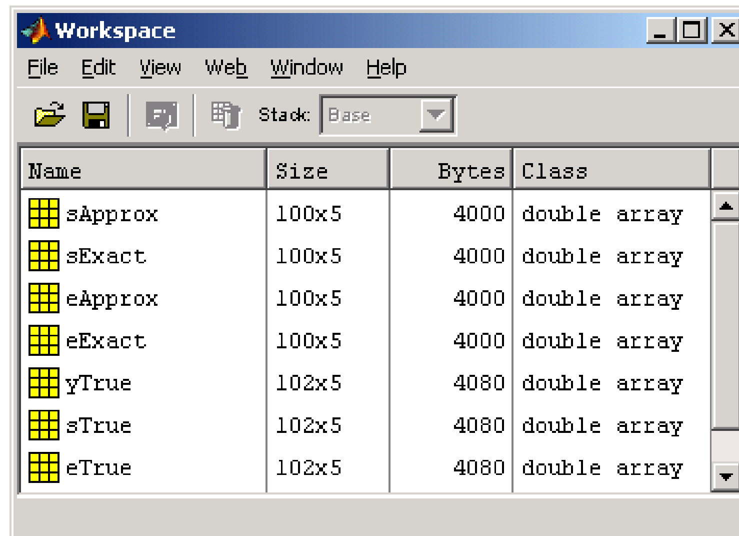
Stack: Base

Name	Size	Bytes	Class
sApprox	100x5	4000	double array
eApprox	100x5	4000	double array
yTrue	102x5	4080	double array
sTrue	102x5	4080	double array
eTrue	102x5	4080	double array

For more information, see the `garchfit` and `garchinfer` function reference pages.

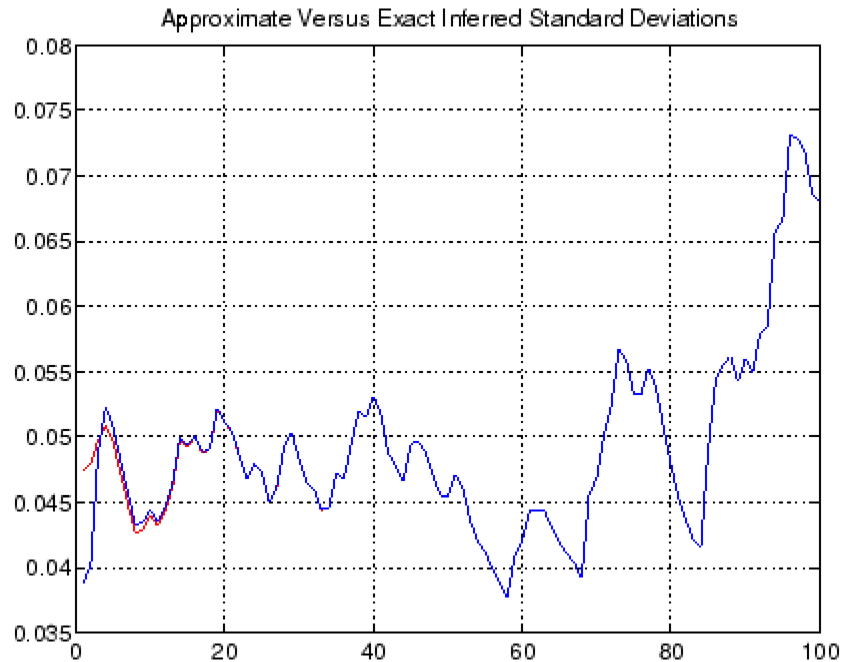
4 Call `garchinfer` again, but this time use the first two rows of the true simulated data as presample data. Use of the presample data allows you to infer the exact residuals and conditional standard deviations:

```
[eExact,sExact] = garchinfer(spec,yTrue(3:end,:),[],...
                  eTrue(1:2,:),sTrue(1:2,:),yTrue(1:2,:));
```



5 Compare the first realization of the approximate and the exact inferred conditional standard deviations reveals the distinction between automatically generated and user-specified presample data:

```
plot(sApprox(:,1),'red')
grid('on'),hold('on')
plot(sExact(:,1),'blue')
title('Approximate Versus Exact Inferred Standard Deviations')
```



The approximate and exact standard deviations are asymptotically identical. The only difference between the two curves is attributable to the transients induced by the default initial conditions. If you were to plot the first realization of the original simulated conditional standard deviations, `sTrue(3:end,1)`, on the current figure, it would lie on top of the blue curve.

Although the previous figure highlights the first realization of conditional standard deviations, the comparison holds for any realization and for the inferred residuals.

Thus, this example reveals the link between simulation and inference: you can think of `garchsim` as a correlation filter capable of processing multiple realizations simultaneously. It is the complement of `garchinfer`, which you can think of as a whitening, or inverse, filter capable of processing multiple realizations simultaneously. The `garchfit` estimation engine can process only a single realization at a time. However, the transient effects highlighted in this example are the same when applied to the estimation.

Alternative Technique for Estimating ARMA(R,M) Parameters

This example shows how you can use the GARCH Toolbox™ software as a general-purpose univariate time-series processor. It demonstrates how to estimate the parameters of ARMA(R,M) models. It uses an alternative technique and the presample inputs `PreInnovations` and `PreSeries`, and assumes a simple constant variance model.

Default Method

Estimation requires presample data to initiate the inverse filtering process. In the absence of explicit presample data, `garchfit` assigns the R required presample observations of y_t, that is, `Series`, the sample mean of `Series`. It also assigns the M required presample observations of ε_t, that is, the innovations, or residuals, their expected value of zero. This method then calculates the log-likelihood objective function value using all the available data in `Series`, and is the default GARCH Toolbox method.

Alternative Technique

Another method also sets the$M$ required presample observations of the residuals, ε_t, to zero, but uses the first R actual observations of `Series` as initial values. Thus, $\{y_1, y_2, ..., y_R\}$ are used to initiate the inverse filter, and the log-likelihood objective function value is based on the remaining observations. See Hamilton [22], page 132, or Box, Jenkins, and Reinsel [10], pages 236-237.

For example, assume that you have some hypothetical time series, `xyz`, and you want to estimate an ARMA(R,M) model with constant conditional variances. Using the alternative presample method, you would exclude the first R observations of `xyz` from the input `Series`, and reserve them for the input `PreSeries`. Specifically, you would set the input `Series = xyz(R+1:end)`, `PreInnovations = zeros(M,1)`, `PreSigmas = []`, and `PreSeries = xyz(1:R)`.

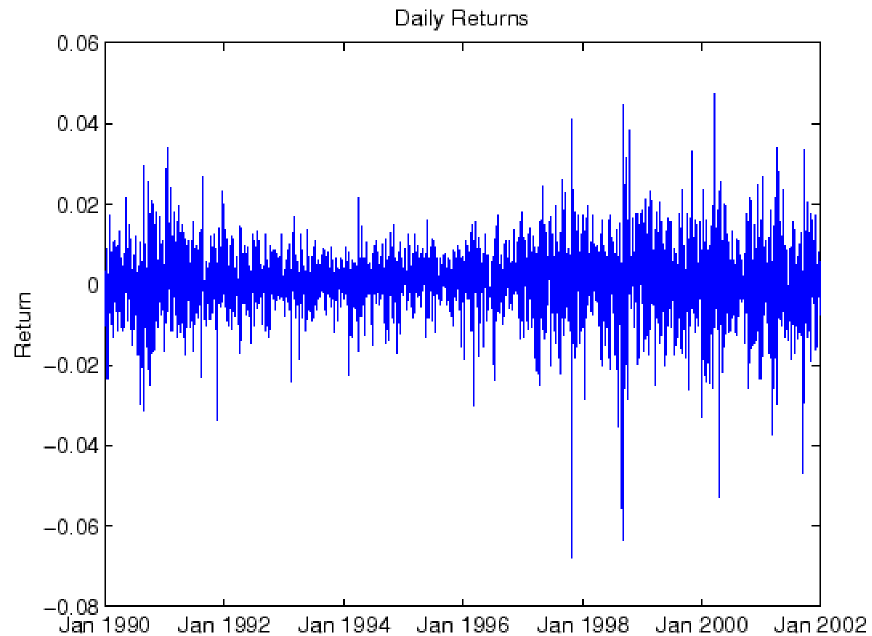
Active Lower Bound Constraint

This example illustrates an active lower bound constraint, $\kappa > 0$, for the conditional variance constant κ. This constraint is required for GARCH and GJR variance models to ensure a positive conditional variance process. It also illustrates how to identify such active constraints, and what to do about this

most commonly encountered active constraint. See "Termination Criteria and Optimization Results" on page 6-15.

1 Load the NYSE data set and convert prices to returns:

```
load garchdata
nyse = price2ret(NYSE);
plot(nyse)
axis([0 length(nyse) -0.08 0.06])
set(gca,'XTick',[1 507 1014 1518 2025 2529 3027])
set(gca,'XTickLabel',{'Jan 1990' 'Jan 1992' 'Jan 1994' ...
    'Jan 1996' 'Jan 1998' 'Jan 2000' 'Jan 2002'})
set(gca,'YTick',[-0.08:0.02:0.06])
ylabel('Return')
title('Daily Returns')
```



2 Estimate a default GARCH(1,1) model and print the estimation results. For this example, `TolCon = 1e-6`. Iterative display is disabled due to space constraints:

```
spec = garchset('Display','off','P',1,'Q',1,'TolCon',1e-6);
[coeff,errors,LLF,eFit,sFit,summary] = garchfit(spec,nyse);
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 4

                                  Standard          T
  Parameter       Value             Error       Statistic
  -----------   -----------   ------------   -----------
            C    0.00051941     0.00013701       3.7910
            K    2e-006         2.8192e-007      7.0943
     GARCH(1)    0.87166        0.0095167       91.5925
      ARCH(1)    0.10419        0.0073771       14.1238
```

3 Examination of these results reveals the estimated variance constant `K = 2e-006 = 0 + 2*TolCon = 2*TolCon`. That is, κ is equal to the theoretical lower bound plus `2*TolCon`. You can see this by printing the summary structure and looking at the constraints message field:

```
summary

summary =
          warning: 'No Warnings'
         converge: 'Function Converged to a Solution'
      constraints: 'Boundary Constraints Active: Standard
                    Errors May Be Inaccurate'
        covMatrix: [4x4 double]
       iterations: 13
    functionCalls: 115
           lambda: [1x1 struct]
```

4 Print the lower and upper bound LaGrange multipliers and examine them for nonzero values:

```
[summary.lambda.lower  summary.lambda.upper]

ans =
```

```
      1.0e+006 *

              0          0
         7.3602          0
              0          0
              0          0
```

The `garchdisp` function determines the display order of the lower and upper bound LaGrange multipliers. This result shows that the lower bound constraint κ > 0 is active.

5 Repeat the estimation with the default `TolCon = 1e-7` and verify that the constraint is no longer active:

```
spec = garchset('Display','off','P',1,'Q',1);
[coeff,errors,LLF,eFit,sFit,summary] = garchfit(spec,nyse);
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 4

                                    Standard          T
  Parameter        Value             Error        Statistic
 -----------    -----------     ------------    -----------
           C    0.00049676        0.00013137         3.7813
           K    8.9128e-007       1.5776e-007        5.6495
    GARCH(1)    0.91088           0.0069142        131.7410
     ARCH(1)    0.079942          0.0058319         13.7077
summary
summary =
          warning: 'No Warnings'
         converge: 'Function Converged to a Solution'
      constraints: 'No Boundary Constraints'
        covMatrix: [4x4 double]
       iterations: 21
    functionCalls: 208
           lambda: [1x1 struct]
[summary.lambda.lower  summary.lambda.upper]
```

```
ans =
     0     0
     0     0
     0     0
     0     0
```

Determining Convergence Status

There are two ways to determine whether an estimation achieves convergence:

- The first, easiest way is to examine the optimization details of the estimation. By default, `garchfit` displays this information in the MATLAB® Command Window.
- The second way is to request the `garchfit` optional summary output.

1 To illustrate these methods, use the `DEM2GBP` (Deutschmark/British pound foreign-exchange rate) data:

```
load garchdata
dem2gbp = price2ret(DEM2GBP);
[coeff,errors,LLF,eFit,sFit,summary] = ...
   garchfit(dem2gbp);
```

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

Number of variables: 4

Functions
 Objective:                            garchllfn
 Gradient:                             finite-differencing
 Hessian:                              finite-differencing (or Quasi-Newton)
 Nonlinear constraints:                armanlc
 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:          4

Algorithm selected
   medium-scale

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

                              max           Directional First-order
Iter F-count    f(x)   constraint  Step-size  derivative Optimality
 1     28    -7916.01    -2.01e-006 7.63e-006        857   1.42e+005
 2     36    -7959.65   -1.508e-006      0.25        389    9.8e+007
 3     45    -7963.98   -3.113e-006     0.125        131   5.29e+006
 4     52    -7965.59   -1.586e-006       0.5       55.9   4.45e+007
 5     65     -7966.9   -1.574e-006   0.00781        101   1.46e+007
 6     74    -7969.46   -2.201e-006     0.125       14.9   2.77e+007
 7     83    -7973.56   -2.663e-006     0.125       36.6   1.45e+007
 8     90    -7982.09   -1.332e-006       0.5      -6.39   5.59e+006
 9    103    -7982.13   -1.399e-006   0.00781       6.49   1.32e+006
10    111    -7982.53   -1.049e-006      0.25       12.5   1.87e+007
11    120    -7982.56   -1.186e-006     0.125       3.72    3.8e+006
12    128    -7983.69    -1.11e-006      0.25      0.184   4.91e+006
13    134    -7983.91   -7.813e-007         1      0.732   1.22e+006
14    140    -7983.98   -9.265e-007         1      0.186   1.17e+006
15    146       -7984   -8.723e-007         1     0.0427   9.52e+005
16    154       -7984   -8.775e-007      0.25     0.0152   6.33e+005
17    160       -7984    -8.75e-007         1    0.00197   6.98e+005
18    166       -7984   -8.763e-007         1   0.000931   7.38e+005
19    173       -7984   -8.759e-007       0.5   0.000469   7.37e+005
20    179       -7984   -8.761e-007         1    0.00012   7.22e+005
21    199       -7984   -8.761e-007  -6.1e-005     0.0167   7.37e+005
22    213       -7984   -8.761e-007   0.00391    0.00582   7.26e+005
Optimization terminated successfully:
Search direction less than 2*options.TolX and
maximum constraint violation is less than options.TolCon
No Active Constraints
```

The optimization details indicate successful termination.

2 Now examine the `summary` output structure:

```
summary
summary =
          warning: 'No Warnings'
         converge: 'Function Converged to a Solution'
      constraints: 'No Boundary Constraints'
        covMatrix: [4x4 double]
       iterations: 22
    functionCalls: 213
           lambda: [1x1 struct]
```

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`).

7

Forecasting the Conditional Mean and Standard Deviation of Return Series

Minimum Mean Square Error Forecasting (p. 7-2) — The outputs of the forecasting engine, `garchpred`

Generating Presample Observations (p. 7-6) — How `garchpred` generates required presample data

Asymptotic Behavior for Long-Range Forecast Horizons (p. 7-7) — Asymptotic behavior of `garchpred` outputs

Examples: Computing Forecasts (p. 7-9) — Computing forecasts of the conditional mean, a volatility forecast, and a forecast with multiple realizations

Minimum Mean Square Error Forecasting

In this section...

"About the Forecasting Engine" on page 7-2

"Conditional Standard Deviations of Future Innovations" on page 7-2

"Conditional Mean Forecasting of the Return Series" on page 7-3

"MMSE Volatility Forecasting of Returns" on page 7-3

"RMSE Associated with Conditional Mean Forecasts" on page 7-4

About the Forecasting Engine

The forecasting engine `garchpred` computes minimum mean square error (MMSE) forecasts of the conditional mean of returns $\{y_t\}$, and the conditional standard deviation of the innovations $\{\varepsilon_t\}$, in each period of a user-specified forecast horizon. To do this, `garchpred` views the conditional mean and variance models from a linear filtering perspective. It then applies iterated conditional expectations to the recursive equations, one forecast period at a time.

Each output of `garchpred` is an array. The number of rows of this array equals the user-specified forecast horizon, and its number of columns equals the number of columns (realizations, or paths) in the time-series array of asset returns, `Series`. For a general forecasting example involving multiple realizations, see "Examples: Computing Forecasts" on page 7-9.

This section discusses the four `garchpred` outputs.

Conditional Standard Deviations of Future Innovations

The first output of `garchpred`, `SigmaForecast`, is a matrix of conditional standard deviations of future innovations (residuals) on a per-period basis. This matrix represents the standard deviations derived from the MMSE forecasts associated with the recursive volatility model you defined in the GARCH specification structure.

For GARCH(P,Q) and GJR(P,Q) models, `SigmaForecast` is the square root of the MMSE conditional variance forecasts. For EGARCH(P,Q) models, `SigmaForecast` is the square root of the exponential of the MMSE forecasts of the logarithm of conditional variance.

The `garchpred` function computes the forecasts iteratively. Therefore, the first row contains the standard deviation in the first forecast period for each realization of `Series`, the second row contains the standard deviation in the second forecast period, and so on. Thus, if you specify a forecast horizon greater than one, `garchpred` also returns the per-period standard deviations of all intermediate horizons. In this case, the last row contains the standard deviation at the specified forecast horizon for each realization of `Series`.

Conditional Mean Forecasting of the Return Series

The second output of `garchpred`, `MeanForecast`, is a matrix of MMSE forecasts of the conditional mean of `Series` on a per-period basis. Again, the first row contains the forecast for each realization of `Series` in the first forecast period, the second row contains the forecast in the second forecast period, and the last row contains the forecast of `Series` at the forecast horizon.

MMSE Volatility Forecasting of Returns

The third output of `garchpred`, `SigmaTotal`, is a matrix of volatility forecasts of returns over multiperiod holding intervals. That is, the first row contains the expected standard deviation of returns for assets held for one period for each realization of `Series`, the second row contains the standard deviation of returns for assets held for two periods, and so on. Thus, the last row contains the forecast of the standard deviation of the cumulative return obtained if an asset was held for the entire forecast horizon.

`garchpred` computes the elements of `SigmaTotal` by taking the square root of

$$\operatorname{var}_t\Big[\sum_{i=1}^{s} y_{t+1}\Big] = \sum_{i=1}^{s}\Big[\Big(1+\sum_{j=1}^{s-i}\psi_j\Big)^2 E_t(\sigma_{t+1}^2)\Big] \tag{7-1}$$

where:

- S is the forecast horizon of interest (`NumPeriods`)

- ψ_j is the coefficient of the *j*th 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

$$\mathrm{var}_t[\sum_{i=1}^{s} y_{t+1}] = \sum_{i=1}^{s} E_t(\sigma_{t+1}^2)$$

. The `SigmaTotal` forecasts are correct for continuously compounded returns, and approximate for periodically compounded returns. If you model the conditional mean as a stationary invertible ARMA process, `SigmaTotal` is the same size as `SigmaForecast`.

For conditional mean models with regression components, in which you specify `X` or `XF`, `SigmaTotal` is an empty matrix, `[]`. In other words, `garchpred` computes `SigmaTotal` only if you model the conditional mean as a stationary invertible ARMA process. For more information, see Chapter 8, "Regression Components".

RMSE Associated with Conditional Mean Forecasts

The fourth output of `garchpred`, `MeanRMSE`, is a matrix of root mean square errors (RMSE) associated with the output forecast array `MeanForecast`. That is, each element of `MeanRMSE` is the conditional standard deviation of the corresponding forecast error (that is, the standard error of the forecast) in the `MeanForecast` matrix. From Baillie and Bollerslev [3], Equation 19,

$$\mathrm{var}_t(y_{t+s}) = \sum_{i=1}^{s} \psi_{s-i}^2 E_t(\sigma_{t+1}^2)$$

Using this equation, the computed MMSE forecasts of the conditional mean (`MeanForecast`), and the standard errors of the corresponding forecasts (`MeanRMSE`), you can construct approximate confidence intervals for conditional mean forecasts. The approximation becomes more accurate during periods of relatively stable volatility (see Baillie and Bollerslev [3] and Bollerslev, Engle, and Nelson [8]). As heteroscedasticity in returns disappears (that is, as the returns approach the homoscedastic, or constant variance, limit), the

approximation is exact. You can then apply the Box & Jenkins confidence bounds (see Box, Jenkins, and Reinsel [10], pages 133-145).

For conditional mean models with regression components (that is, `X` or `XF` is specified), `MeanRMSE` is an empty matrix, `[]`. In other words, `garchpred` computes `MeanRMSE` only if the conditional mean is modeled as a stationary invertible ARMA process. See Chapter 8, "Regression Components".

Generating Presample Observations

As discussed in "Minimum Mean Square Error Forecasting" on page 7-2, garchpred computes MMSE forecasts. It does this by applying iterated conditional expectations to the conditional mean and variance models one forecast period at a time. Since these models are generally recursive in nature, they often require presample data to initiate the iterative forecasting process. This initial data plays the identical role that the presample time-series inputs PreInnovations, PreSigmas, and PreSeries play in simulation and estimation. For more information, see garchsim, garchfit, and garchinfer.

The time-series array of asset returns, Series, is a required input. The garchpred function takes the initial returns needed to initiate forecasting of the conditional mean directly from the last (most recent) rows of Series.

For example, consider a conditional mean model with an AR(R) autoregressive component. In this case, garchpred takes the R observations required to initiate the forecast of each realization of Series directly from the last$R$ rows of Series.

However, garchpred obtains initial innovations and conditional standard deviations needed to initiate forecasting of the conditional variance model from the input array Series via the inverse filtering inference engine garchinfer.

For more information, see:

- "Maximum Likelihood Estimation" on page 6-2
- "Presample Observations" on page 6-12
- The garchinfer function reference page

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 noted earlier, the conditional standard deviation forecast `sigmaForecast`, which is the first `garchpred` output, approaches the unconditional standard deviation of $\{\varepsilon_t\}$.

  For GARCH(P,Q) models it is

  $$\sigma = \sqrt{\frac{\kappa}{1 - \sum_{i=1}^{P} G_i - \sum_{j=1}^{Q} Aj}}$$

  For GJR(P,Q) models, it is

  $$\sigma = \sqrt{\frac{\kappa}{1 - \sum_{i=1}^{P} G_i - \sum_{j=1}^{Q} Aj - \frac{1}{2}\sum_{j=1}^{Q} Lj}}$$

  And for EGARCH(P,Q) models, it is

  $$\sigma = \sqrt{e^{\frac{\kappa}{1-\sum_{i=1}^{P} G_i}}}$$

- GARCH effects do not affect the MMSE forecast of the conditional mean `meanForecast`, which is the second `garchpred` output. 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. It only affects the uncertainty of the mean forecast, leaving the mean forecast itself unchanged.

- The mean square error of the conditional mean `meanRMSE^2`, which is the square of the fourth `garchpred` output, approaches the unconditional variance of $\{y_t\}$.

- EGARCH(P,Q) models represent the logarithm of the conditional variance as the output of a linear filter, rather than the conditional variance process itself. Because of this, the MMSE forecasts derived from EGARCH(P,Q) models are optimal for the logarithm of the conditional variance. They are, however, generally downward-biased forecasts of the conditional variance process itself. The following output arrays are based on the conditional variance forecasts:
  - `SigmaForecast`
  - `SigmaTotal`
  - `MeanRMSE`

  Thus, these outputs generally underestimate their true expected values for conditional variance forecasts derived from EGARCH(P,Q) models. The important exception is the one-period ahead forecast, which is unbiased in all cases. For unbiased multiperiod forecasts of `SigmaForecast`, `SigmaTotal`, and `MeanRMSE`, you can perform Monte Carlo simulation using `garchsim`. For an example, see Chapter 11, "Example Workflow: Estimation, Forecasting, and Simulation".

Examples: Computing Forecasts

In this section...

"Forecasting Using garchpred" on page 7-9

"Volatility Forecasting over Multiple Periods" on page 7-12

"Forecasting with Multiple Realizations" on page 7-15

Forecasting Using `garchpred`

The section "Example: Analysis and Estimation Using the Default Model" on page 2-16 uses the default GARCH(1,1) model to model the Deutschmark/British pound foreign-exchange series. This example shows how to forecast with the `garchpred` function, using the model:

$$y_t = -6.1919e^{-005} + \varepsilon_t$$

$$\sigma_t^2 = 1.0761e^{-006} + 0.80598\sigma_{t-1}^2 + 0.15313\varepsilon_{t-1}^2$$

1 Use the following commands to restore your workspace if necessary:

```
load garchdata
dem2gbp = price2ret(DEM2GBP);
[coeff,errors,LLF,innovations,sigmas] = garchfit(dem2gbp);
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 4

                                     Standard          T
  Parameter          Value            Error        Statistic
 -----------    -----------    ------------    -----------
           C    -6.1919e-005    8.4331e-005       -0.7342
           K     1.0761e-006     1.323e-007        8.1341
    GARCH(1)     0.80598         0.016561         48.6685
     ARCH(1)     0.15313         0.013974         10.9586
```

Due to space constraints, the display output of the estimation is not included here.

2 Call `garchpred` to forecast the returns for the Deutschmark/British pound foreign-exchange series using the default model parameter estimates. Provide the specification structure `coeff` (the output of `garchfit`) and the FX return series `dem2gbp`, and the number of forecast periods as input:

Note The following example results appear in **Short E** numeric format for readability. Select **File** > **Preferences** > **Command Window** > **Text display: short e** before starting the example to duplicate this format.

Use the following command to forecast the conditional mean and standard deviation in each period of a 10-period forecast horizon:

```
[sigmaForecast,meanForecast]  = garchpred(coeff,dem2gbp,10);
[sigmaForecast,meanForecast]
ans =
  3.8340e-003  -6.1919e-005
  3.8954e-003  -6.1919e-005
  3.9535e-003  -6.1919e-005
  4.0084e-003  -6.1919e-005
  4.0603e-003  -6.1919e-005
  4.1095e-003  -6.1919e-005
  4.1562e-003  -6.1919e-005
  4.2004e-003  -6.1919e-005
  4.2424e-003  -6.1919e-005
  4.2823e-003  -6.1919e-005
```

The result consists of the MMSE forecasts of the conditional standard deviations and the conditional mean of the return series `dem2gbp` for a 10-period default horizon. They show that the default model forecast of the conditional mean is always `C=-6.1919e-05`. This is true for any forecast horizon because the expected value of any innovation, ε_t, is `0`.

The conditional standard deviation forecast (`sigmaForecast`) changes from period to period and approaches the unconditional standard deviation of $\{\varepsilon_t\}$, given by

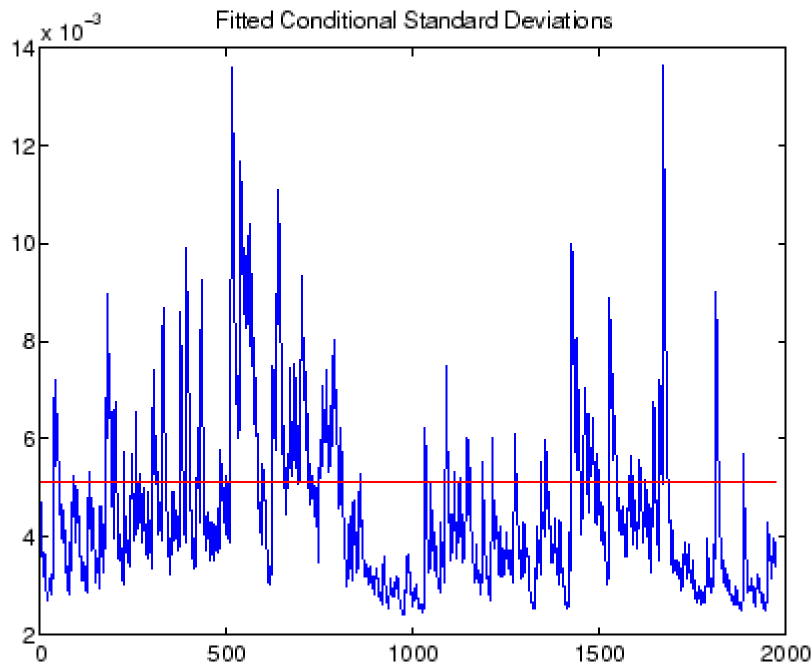
$$\sigma = \sqrt{\frac{\kappa}{1 - \sum_{i=1}^{P} G_i - \sum_{j=1}^{Q} A_j}}$$

3 Calculate the unconditional standard deviation of $\{\varepsilon_t\}$:

```
s0 = sqrt(coeff.K/(1 - sum([coeff.GARCH(:);coeff.ARCH(:)])))
s0 =
     5.1300e-003
```

4 Plot the unconditional standard deviation, `5.1300e-003`, and the conditional standard deviations, `sigmas`, derived from the fitted returns. The plot shows that the most recent values of $\{\sigma_t\}$ fall below this long-run, asymptotic value:

```
plot(sigmas), hold('on')
plot([0 size(sigmas,1)],[s0 s0],'red')
title('Fitted Conditional Standard Deviations')
hold('off')
```



Volatility Forecasting 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, you can forecast the standard deviation of the return you would obtain if you purchased shares in a mutual fund that mirrors the performance of the New York Stock Exchange Composite Index today, and sold it 10 days from now.

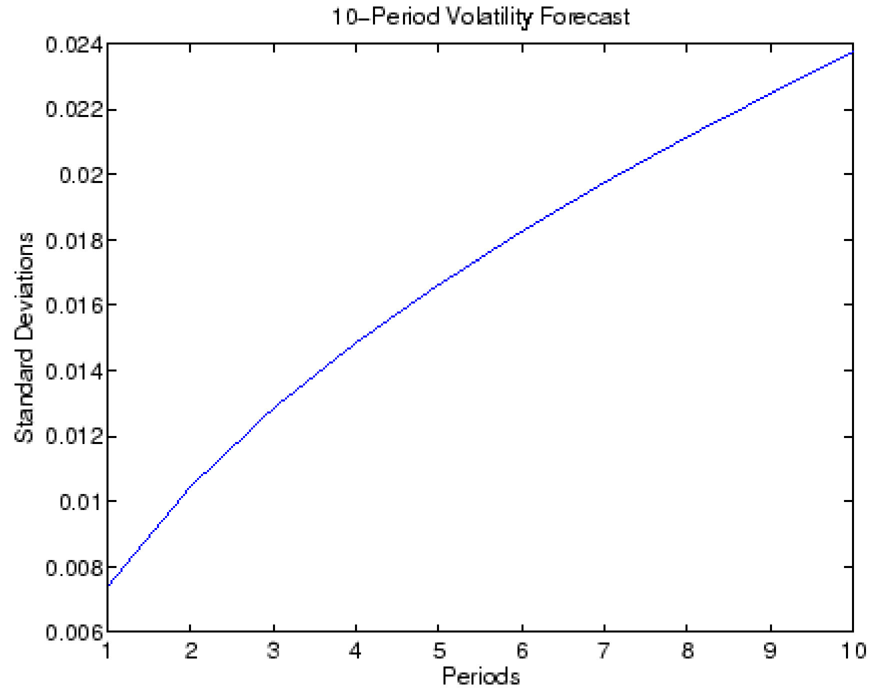
1 Use the default GARCH(1,1) model (“The Default Model” on page 2-13) to estimate the model parameters for the NYSE data set. The following text omits the display output of the estimation to save space:

```
load garchdata
nyse = price2ret(NYSE);
[coeff,errors,LLF,innovations,sigmas] = garchfit(nyse);
garchdisp(coeff,errors)
   Mean: ARMAX(0,0,0); Variance: GARCH(1,1)
   Conditional Probability Distribution: Gaussian
```

```
 Number of Model Parameters Estimated: 4
                                     Standard          T
  Parameter       Value               Error        Statistic
 -----------   -----------   ------------   -----------
           C    0.00049676     0.00013137        3.7813
           K   8.9128e-007    1.5776e-007        5.6495
    GARCH(1)       0.91088      0.0069142      131.7410
     ARCH(1)      0.079942      0.0058319       13.7077
```

2 Now, forecast and plot the standard deviation of the return you would obtain if you sold the shares after 10 days.

```
[sigmaForecast,meanForecast,sigmaTotal] = garchpred(coeff,...
                                          nyse,10);
plot(sigmaTotal)
ylabel('Standard Deviations')
xlabel('Periods')
title('10-Period Volatility Forecast')
hold('off')
```



This plot represents the standard deviation of the returns (`sigmaTotal`) expected if you held the shares for the number of periods shown on the *x*-axis. The value for the tenth period is the volatility forecast of the expected return if you purchased the shares today and held them for 10 periods. The calculation of `sigmaTotal` is strictly correct for continuously compounded returns only, and is an approximation for periodically compounded returns.

3 Convert the standard deviations `sigmaForecast` and `sigmaTotal` to variances by squaring each element. You then see an interesting relationship between the cumulative sum of `sigmaForecast.^2` and `sigmaTotal.^2`:

```
format short e
[cumsum(sigmaForecast.^2)  sigmaTotal.^2]
ans =
  5.4587e-005  5.4587e-005
  1.0956e-004  1.0956e-004
  1.6493e-004  1.6493e-004
```

```
2.2068e-004  2.2068e-004
2.7680e-004  2.7680e-004
3.3331e-004  3.3331e-004
3.9018e-004  3.9018e-004
4.4743e-004  4.4743e-004
5.0504e-004  5.0504e-004
5.6302e-004  5.6302e-004
```

Although not equivalent, this relationship in the presence of heteroscedasticity is like the square-root-of-time rule. This familiar rule converts constant variances of uncorrelated returns expressed on a per-period basis to a variance over multiple periods. This relationship between `sigmaForecast` and `sigmaTotal` holds for the default conditional mean model only (the relationship is valid for uncorrelated returns).

Forecasting with Multiple Realizations

This example illustrates how to forecast multiple realizations of an MA(1) conditional mean model with an EGARCH(1,1) conditional variance model.

1 Load the `NYSE` data set and convert prices to returns:

```
load garchdata
nyse = price2ret(NYSE);
```

2 Create a specification structure template, and estimate and display the estimation results:

```
spec = garchset('VarianceModel','EGARCH','M',1,'P',1,'Q',1,...
                'Display','off');
[coeff,errors] = garchfit(spec,nyse);
garchdisp(coeff,errors)
  Mean: ARMAX(0,1,0); Variance: EGARCH(1,1)
  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 6
                                 Standard          T
  Parameter       Value            Error       Statistic
 -----------   -----------   ------------   -----------
           C    0.00022434    0.00014038        1.5981
       MA(1)    0.10677       0.018795          5.6806
           K   -0.25399       0.031452         -8.0755
```

```
     GARCH(1)     0.97329        0.003231       301.2365
      ARCH(1)     0.14514        0.011845        12.2533
  Leverage(1)    -0.10359        0.0081483      -12.7128
```

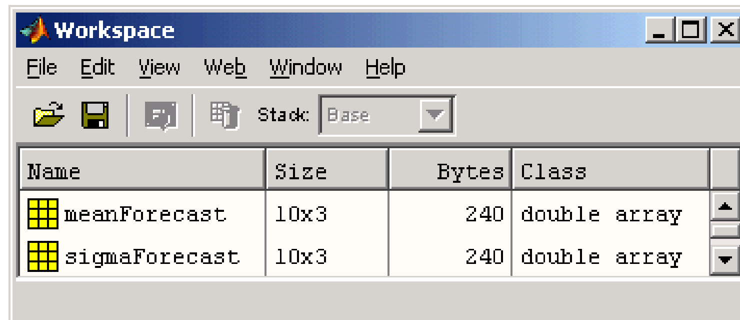
3 Based on the estimation results, do the following:

a Simulate 1000 observations for each of three independent realizations.

b Forecast the conditional standard deviations and returns for a 10-period forecast horizon.

```
randn('state',0);
rand('twister',0);
[innovations,sigmas,series] = garchsim(coeff,1000,3);
[sigmaForecast,meanForecast]= garchpred(coeff,series,10);
```

The sigmaForecast and meanForecast outputs are 10-by-3 arrays. Both arrays have the same number of rows as the specified number of periods. The first row contains the standard deviations and mean forecasts for the first period, and the last row contains these values for the most recent period. Both arrays have the same number of columns as there are realizations, that is, columns, in the simulated return series, series.



8

Regression Components

Introduction (p. 8-2)	Introduces the concept of a regression component in the conditional mean model
Example: Incorporating a Regression Model into an Estimation (p. 8-3)	How to use the asymptotic equivalence of autoregressive models and linear regression models to perform an estimation when the conditional mean model includes a regression component
Simulation and Inference Using a Regression Component (p. 8-8)	Syntax for including a matrix of explanatory data (a regression matrix) in calls to `garchsim` and `garchinfer`.
Forecasting Using a Regression Component (p. 8-9)	Explains the need for both explanatory and forecast explanatory data when you incorporate a regression component in a forecast
Ordinary Least Squares Regression (p. 8-11)	Example of an ordinary least squares regression
Regression in a Monte Carlo Framework (p. 8-13)	Considers Monte Carlo simulation that includes a regression component

Introduction

The GARCH Toolbox™ software allows conditional mean models with regression components, that is, of general ARMAX(R,M,Nx) form.

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

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

Conditional mean models with a regression component introduce additional complexity, because GARCH Toolbox functions have 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 GARCH Toolbox primary functions (that is, `garchfit`, `garchinfer`, `garchpred`, and `garchsim`) accept an optional regression matrix, `X`, that represents X in the equation shown here. You must do the following:

- Ensure that the regression matrix you provide is valid.
- Collect and format the past history of explanatory data you include in `X`.
- For forecasting, forecast `X` into the future to form `XF`.

Example: Incorporating a Regression Model into an Estimation

In this section...
“Fitting a Model to a Simulated Return Series” on page 8-3
“Fitting a Regression Model to the Same Return Series” on page 8-5

Fitting a Model to a Simulated Return Series

This section uses an AR(R)/GARCH(P,Q) model to fit a simulated return series to the defined model.

1 **Define an AR(2)/GARCH(1,1) model.** Start by creating a specification structure for an AR(2)/GARCH(1,1) composite model. Set the `'Display'` parameter `'off'` to suppress the optimization details that `garchfit` normally displays.

```
spec = garchset('AR',[0.5 -0.8],'C',0,'Regress',[0.5 -0.8],...
                'GARCH',0.7,'ARCH',0.1,'K',0.005,...
                'Display','off')
spec =
          Comment: 'Mean: ARMAX(2,0,?); Variance: GARCH(1,1)'
     Distribution: 'Gaussian'
                R: 2
                C: 0
               AR: [0.5000 -0.8000]
          Regress: [0.5000 -0.8000]
    VarianceModel: 'GARCH'
                P: 1
                Q: 1
                K: 0.0050
            GARCH: 0.7000
             ARCH: 0.1000
          Display: 'off'
```

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.

2 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:

a Estimate the parameters of the simulated return series.

b Compare the parameter estimates to the original coefficients in `spec`.

```
randn('state',0);
rand('twister',0);
[e,s,y] = garchsim(spec,2000,1);
[coeff,errors] = garchfit(spec,y);
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 6

                                    Standard          T
  Parameter        Value              Error        Statistic
 -----------    -----------     ------------    -----------
           C     -0.00044755     0.0034623         -0.1293
       AR(1)     0.50257         0.01392           36.1049
       AR(2)     -0.8002         0.013981         -57.2344
           K     0.0050532       0.001971           2.5637
    GARCH(1)     0.70954         0.095319           7.4439
     ARCH(1)     0.083296        0.022665           3.6752
```

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

Because you specified no explanatory regression matrix as input to `garchsim` and `garchfit`, these functions ignore the regression coefficients

(`Regress`). The `garchdisp` output shows a `0` for the order of the regression component.

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`. The return series `y` was simulated in "Fitting a Model to a Simulated Return Series" on page 8-3.

1 **Remove the 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)'
     Distribution: 'Gaussian'
                C: 0
          Regress: [0.5000 -0.8000]
    VarianceModel: 'GARCH'
                P: 1
                Q: 1
                K: 0.0050
            GARCH: 0.7000
             ARCH: 0.1000
          Display: 'off'
```

2 **Create the regression matrix.** Create a regression matrix of first- and second-order lags using the simulated returns vector `y` from "Fitting a Model to a Simulated Return Series" on page 8-3 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
```

3 **Examine the regression matrix.** A `NaN` (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 (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.

4 **Repair the regression matrix.** You can do one of two things in order 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.

---

5 **Fit the regression model.** Now 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)
  Mean: ARMAX(0,0,2); Variance: GARCH(1,1)
  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 6
                                    Standard          T
  Parameter        Value             Error        Statistic
 -----------    -----------    ------------    -----------
          C     -0.00044754     0.0034628         -0.1292
  Regress(1)     0.50257        0.01392           36.1048
  Regress(2)    -0.8002         0.013981         -57.2346
          K      0.0050526      0.0019708          2.5637
    GARCH(1)     0.70957        0.095311           7.4447
     ARCH(1)     0.083292       0.022663           3.6752
```

These estimation results are like those shown for the AR model in the section “Fitting a Model to a Simulated Return Series” on page 8-3. This similarity illustrates the asymptotic equivalence of autoregressive models and linear regression models.

This part of the example illustrates the extra steps involved in formatting the explanatory matrix. It 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 like including one with `garchfit`. (See "Example: Incorporating a Regression Model into an Estimation" on page 8-3.)

For example, the following command simulates a single realization of 2000 observations of the innovations, conditional standard deviations, and returns:

```
randn('state',0);
rand('twister',0);
[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

In this section...

"Using Forecasted Explanatory Data" on page 8-9
"Generating Forecasted Explanatory Data" on page 8-10

Using Forecasted Explanatory Data

To forecast the conditional mean of a return series `y` in each period of a 10-period forecast horizon, call `garchpred` with the following syntax:

```
NumPeriods = 10;
[sigmaForecast,meanForecast] = ...
      garchpred(spec,y,NumPeriods,X,XF);
```

where `X` is the same regression matrix shown in "Fitting a Regression Model to the Same Return Series" on page 8-5, and `XF` is a regression matrix of forecasted explanatory data. In fact, `XF` represents a projection into the future of the explanatory data in `X`. This command 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 must equal or exceed the forecast horizon, `NumPeriods`. When the number of forecasts in `XF` exceeds the forecast horizon, `garchpred` uses only the first `NumPeriods` forecasts. If `XF` is empty (`[]`) or missing, the conditional mean forecast, `meanForecast`, has no regression component.

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

Forecasting Only the Conditional Standard Deviation

To forecast the conditional standard deviation (that is, `sigmaForecast`), `XF` is unnecessary, and `garchpred` ignores it if it is present. This is true even if you included the matrix `X` in the simulation and/or estimation process.

For example, you could use the following syntax to forecast only the conditional standard deviation of the innovations $\{\varepsilon_t\}$ over a 10-period forecast horizon:

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

Forecasting the Conditional Mean

To forecast the conditional mean (that is, `meanForecast`), specify both `X` and `XF`. For example, to forecast the conditional mean of the return series `y` over a 10-period forecast horizon:

```
[sigmaForecast,meanForecast] = garchpred(spec,y,10,X,XF);
```

Generating 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 that corresponds to a single explanatory variable. You must find a way to generate the forecast of `X` (that is, `XF`).

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

Ordinary Least Squares Regression

This example illustrates an ordinary least squares regression, by simulating a return series that scales the daily return values of the New York Stock Exchange Composite Index. It also provides an example of a constant conditional variance model.

1 Load the NYSE data set and convert the price series to a return series:

```
load garchdata
nyse = price2ret(NYSE);
```

2 Create a specification structure. Set the Display flag to 'off' to suppress the optimization details that garchfit usually displays:

```
spec = garchset('P',0,'Q',0,'C',0,...
                'Regress',1.2,...
                'K',0.00015,...
                'Display','off')
spec =
          Comment: 'Mean: ARMAX(0,0,?); Variance: GARCH(0,0)'
     Distribution: 'Gaussian'
                C: 0
          Regress: 1.2000
    VarianceModel: 'GARCH'
                K: 1.5000e-004
          Display: 'off'
```

3 Simulate a single realization of 2000 observations, fit the model, and examine the results:

```
randn('state',0);
rand('twister',0);
[e,s,y]        = garchsim(spec,2000,1,[],nyse);
[coeff,errors] = garchfit(spec,y,nyse);
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
```

```
Number of Model Parameters Estimated: 3

                              Standard          T
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
          C    4.9091e-006    0.00027114        0.0181
Regress(1)    1.2251         0.028909         42.3786
          K    0.00014662     4.6945e-006      31.2334
```

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 appears purely for illustrative purposes. Although you can use the GARCH Toolbox™ software to perform OLS, to do so is computationally inefficient and not recommended.

Regression in a Monte Carlo Framework

In the general case, these functions process multiple realizations (that is, sample paths) of univariate time series:

- garchsim
- garchinfer
- garchpred

The outputs of garchsim and the observed return series input to garchpred and garchinfer can be time-series matrices in which each column represents an independent realization. garchfit is different, because the input observed return series of interest must be a vector (that is, 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 $\{y_t\}$:

```
NumSamples = 100;
NumPaths   = 10;
randn('state',0);
rand('twister',0);
[e,s,y] = garchsim(spec,NumSamples,NumPaths,[],X);
```

In a true Monte Carlo simulation of this 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 a single-column output.

Univariate Unit Root Tests

Introduction (p. 9-2)	Augmented Dickey-Fuller and Phillips-Perron univariate unit root tests.
Dickey-Fuller Tests (p. 9-4)	Three augmented Dickey-Fuller unit root tests: `dfARTest`, `dfARDTest`, and `dfTSTest`.
Phillips-Perron Tests (p. 9-6)	Three Phillips-Perron unit root tests: `ppARTest`, `ppARDTest`, and `ppTSTest`.
How to Test for Unit Roots: Inputs and Outputs (p. 9-8)	How to use the common interface to the Dickey-Fuller and Phillips-Perron functions to test for unit roots
Interpretation of Results (p. 9-11)	Pitfalls in interpreting unit root tests
Examples: Unit Root Tests (p. 9-13)	Conducts two tests: one with a trend stationary component, and a second with a drift component

Introduction

In this section...

"Critical Values" on page 9-2

"Serial Dependence" on page 9-2

Critical Values

The GARCH Toolbox™ software supports several members of the Phillips-Perron and augmented Dickey-Fuller classes of univariate unit root tests. The test statistics for these tests are straightforward to evaluate by ordinary least-squares regression, but many of the most common parametric cases follow nonstandard distributions. Therefore, the test statistics need to be compared to critical values derived from Monte Carlo simulations.

The GARCH Toolbox software derives critical values from the simulations for various combinations of sample size and significance level. The significance level sets the probability of a Type I error of incorrectly rejecting the null hypothesis of the underlying process when it is true. Specifically, five million Monte Carlo trials of a given sample size are generated using independent, identically distributed standard Gaussian disturbances. For each sample size, tabulated critical values are the quantiles associated with given cumulative probabilities (significance levels) of the simulated test statistic.

The test suite supports sample sizes as small as 10 and significance levels ranging from 0.001 to 0.999. For small samples, the critical values are exact only for Gaussian residuals. As the sample size becomes larger, critical values are also valid for non-Gaussian residuals. All univariate unit root tests are conventional single-tailed tests.

Serial Dependence

Although augmented Dickey-Fuller and Phillips-Perron tests both attempt to compensate for serial dependence in the residuals process, they do so in different ways. For a given parametric specification of the null hypothesis, Phillips-Perron tests retain the same OLS (ordinary least squares) regression model, but they adjust the test statistics to account for serially dependent residuals. The augmented Dickey-Fuller tests, by contrast, add lagged

changes of the observed time series as explanatory variables in the OLS regression model. Hamilton [22] and Greene [19] contain more discussion of these tests.

Dickey-Fuller Tests

In this section...

"Definitions of Operators" on page 9-4
"dfARTest" on page 9-4
"dfARDTest" on page 9-4
"dfTSTest" on page 9-5

Definitions of Operators

The GARCH Toolbox™ software supports three augmented Dickey-Fuller unit root hypothesis tests. In the following equations, define y_t and ε_t as the univariate time series of observed data and model residuals, respectively. Also, define the first difference operator Δ such that $\Delta y_t = y_t - y_{t-1}$.

dfARTest

The first form of the augmented Dickey-Fuller unit root test assumes that a zero drift unit root process underlies the observed time series y_t. Specifically, under the null hypothesis, the true underlying process is a zero drift ARIMA(P,1,0) model

$$y_t = y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots + \zeta_p \Delta y_{t-p} + \varepsilon_t$$

which is equivalent to an integrated AR(P+1) model.

As an alternative, the estimated OLS regression model is

$$y_t = \phi y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots + \zeta_p \Delta y_{t-p} + \varepsilon_t$$

for some AR(1) coefficient $\varphi < 1$.

dfARDTest

The second form of the augmented Dickey-Fuller unit root test also assumes that a zero drift unit root process underlies the observed time series y_t.

Specifically, under the null hypothesis, the true underlying process is a zero drift ARIMA(P,1,0) model

$$y_t = y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots + \zeta_p \Delta y_{t-p} + \varepsilon_t$$

which is equivalent to an integrated AR(P+1) model.

In this case, the alternative estimated OLS regression model is

$$y_t = C + \phi y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots + \zeta_p \Delta y_{t-p} + \varepsilon_t$$

for some constant C and AR(1) coefficient $\varphi < 1$.

dfTSTest

The third form of the augmented Dickey-Fuller unit root test assumes that a unit root process with arbitrary drift underlies the observed time series y_t. Specifically, under the null hypothesis, the true process underlying the observed time series y_t is an ARIMA(P,1,0) model with drift

$$y_t = C + y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots + \zeta_p \Delta y_{t-p} + \varepsilon_t$$

which is equivalent to an integrated AR(P+1) model.

As an alternative, the estimated OLS regression model includes a time trend,

$$y_t = C + \phi y_{t-1} + \delta t + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots + \zeta_p \Delta y_{t-p} + \varepsilon_t$$

for some constant C, AR(1) coefficient $\varphi < 1$, and time trend stationary coefficient δ.

Phillips-Perron Tests

In this section...

"Definitions of Operators" on page 9-6
"ppARTest" on page 9-6
"ppARDTest" on page 9-6
"ppTSTest" on page 9-7

Definitions of Operators

The GARCH Toolbox™ software supports three Phillips-Perron unit root hypothesis tests. In the following equations, define y_t and ε_t as the univariate time series of observed data and model residuals, respectively.

ppARTest

The first form of the Phillips-Perron unit root test assumes that a zero drift unit root process underlies the observed time series y_t. Under the null hypothesis, the assumed underlying process is

$$y_t = y_{t-1} + \varepsilon_t$$

As an alternative, the estimated OLS regression model is

$$y_t = \phi y_{t-1} + \varepsilon_t$$

for some AR(1) coefficient $\varphi < 1$.

ppARDTest

The second form of the Phillips-Perron unit root test also assumes that a zero drift unit root process underlies the observed time series y_t. Under the null hypothesis, the assumed underlying process is

$$y_t = y_{t-1} + \varepsilon_t$$

As an alternative, the estimated OLS regression model is

$$y_t = C + \phi y_{t-1} + \varepsilon_t$$

for some constant C and AR(1) coefficient $\varphi < 1$.

ppTSTest

The third form of the Phillips-Perron unit root test assumes that a unit root process with arbitrary drift underlies the observed time series y_t. Under the null hypothesis, the assumed underlying process is

$$y_t = C + y_{t-1} + \varepsilon_t$$

for an arbitrary constant C. As an alternative, the estimated OLS regression model is

$$y_t = C + \phi y_{t-1} + \delta t + \varepsilon_t$$

for some constant C, AR(1) coefficient $\varphi < 1$, and time trend stationary coefficient δ.

How to Test for Unit Roots: Inputs and Outputs

In this section...

"About the Common Interface" on page 9-8
"Lags" on page 9-8
"Significance Level" on page 9-9
"TestType" on page 9-9
"Outputs" on page 9-10

About the Common Interface

All of the Dickey-Fuller and Phillips-Perron functions share a common interface. In addition to a univariate time series y_t to be tested, all functions accept the following arguments:

- An integer input vector `Lags`.
- A vector `Alpha`, to set significance levels.
- A character string `TestType`, to select the form of the test.

The output vectors are `H`, `PValue`, `TestStat`, and `CriticalValue`.

See "Examples: Unit Root Tests" on page 9-13 for more information about the syntax required for both input and output parameters. The examples also illustrate how to interpret tests results.

Lags

The input vector `Lags` always serves as a correction for serial correlation of residuals. The precise meaning of the vector, however, differs between the Dickey-Fuller and the Phillips-Perron tests:

- In Dickey-Fuller tests, `Lags` indicates the number of lagged changes or first differences in y_t that are included in the OLS regression model. It is represented as p in the Dickey-Fuller equations.

- In Phillips-Perron tests, `Lags` indicates the number of lagged autocovariance terms included in the Newey-West estimation of the asymptotic variance of the sample mean of residuals.

In all cases, setting `Lags = 0` applies no correction for serial correlation, and the Dickey-Fuller and Phillips-Perron tests produce identical results.

Significance Level

The significance level `Alpha` always denotes the same probability, or set of probabilities, for all six unit root tests. The significance level is the probability, in the appropriate tail of the distribution, of rejecting the null hypothesis when it is in fact true and should be accepted. See “Critical Values” on page 9-2.

TestType

The input `TestType` specifies the basic form of the test used to construct the test statistic. The three test types are *AR*, *t*, and *F*. All three tests are conventional, single-tailed tests.

AR and t Tests

Suppose you conduct a Dickey-Fuller or Phillips-Perron test where you specify no correction for serial dependence (`Lags = 0`). Two possibilities exist:

- Set `TestType = 'AR'` to select a unit root test based on the AR(1) regression coefficient φ, without the need to calculate the standard error. In this case, the test statistic χ based on T observations of y_t is $\chi = T(\varphi - 1)$.
- Set `TestType = 't'` to select a unit root test based on the studentized *t* test. In this case, the test statistic χ based on the AR(1) regression coefficient φ and its standard error σ_φ is $\chi = (\varphi - 1)/\sigma_\varphi$.

When you specify a correction for serial dependence (`Lags > 0`), the test function adjusts the computation of χ. See Hamilton (1994) for details.

Both the *AR* and the *t* test are lower-tailed tests. The null hypothesis is rejected if the test statistic is less than the critical value.

F Tests

Two Dickey-Fuller tests, `dfARDTest` and `dfTSTest`, let you specify joint OLS *F* tests. For `dfARDTest`, the *F* test is of a unit root ($\Phi = 1$) with zero drift (`C=0`). For `dfTSTest`, the *F* test is of a unit root ($\Phi = 1$) with a zero trend stationary coefficient ($\delta = 1$). In both cases, the joint *F* test is an upper-tailed test. Reject the null hypothesis if the test statistic is greater than the critical value.

Outputs

The six unit root tests return the same set of output arguments. The first output is a vector of logical indicators, `H`. `H = 0` indicates acceptance of the null hypothesis. `H = 1` indicates rejection of the null hypothesis. Each element of `H` corresponds to a particular lag of `Lags` and significance level of `Alpha`. Each element of `H` also corresponds to the following:

- An output vector of *p*-values called `PValue`.
- A vector of test statistics called `TestStat`.
- A vector of critical values called `CriticalValue`.

Interpretation of Results

Analysts often associate rejection of the null unit root hypothesis with the assertion of a stationary AR(1) model. They assume that acceptance of the alternative hypothesis implies that the time series y_t is stationary. This assumption is correct in most — but not all — practical applications. In fact, there are many reasons why you should interpret unit root test results with care.

For instance, an AR(1) model is stationary if and only if the magnitude of the AR coefficient is strictly less than 1 (that is, $|\varphi| < 1$). Assume, for example, that the AR coefficient estimated by OLS is $\Phi = -2$. A test statistic based on this coefficient is well under the applicable critical value. You correctly reject the null hypothesis. Yet the time series is nonstationary!

Another pitfall is to confuse unit root tests with random walk tests. For a unit root model to be a random walk, the residuals are generally assumed to be independent and identically distributed Gaussian random variables. Other forms of random walk exist, but all require the residuals to be at least uncorrelated. Since unit root tests are often designed to compensate for serial correlation, unit root processes are more general than random walks. Put another way, the unit root null hypothesis includes a random walk.

Given a stationary process of finite sample size, a unit root process exists that describes it arbitrarily well. Thus, you should use a unit root test to formulate a well-performing, simple representation of an observed time series. Do not use the test only to determine whether the true underlying process actually contains a unit root.

Although this section includes general comments that apply to any unit root test, there are subtleties that pertain to individual tests in the augmented Dickey-Fuller suite. Specifically, in certain circumstances the results of Dickey-Fuller tests can be particularly sensitive to the form of the test statistic.

Augmented Dickey-Fuller tests compensate for serial dependence by adding lagged changes of the observed time series as explanatory variables in the OLS regression model. The null hypothesis assumes that the polynomial associated with the coefficients of lagged changes is stationary. That is, the null hypothesis posits that the specified time series is nonstationary in levels,

but stationary in first differences. If a nonstationary polynomial of lagged changes is found, a warning message appears, indicating that the test results may be unreliable.

In fact, in these situations, the test statistics, and therefore the acceptance/rejection decision, might be entirely inconsistent between the two forms of the test as indicated by the `TestType` input. The t form of the test may indicate a very strong rejection of the unit root null hypothesis, while the AR form may indicate a very strong acceptance. In these situations, the t form of the test is likely more reliable, although the maintained assumptions underlying each type of test are unlikely to be satisfied.

Situations in which a nonstationary polynomial is found may require special attention, and subjective judgment regarding the nature of the observed data. For example, a nonstationary polynomial may indicate that the data is nonstationary in both levels and first differences. In this case, you can appropriately reject the unit root null. In many cases it is helpful to preprocess the data, perhaps differencing or taking logarithms, to better condition the time series.

Examples: Unit Root Tests

In this section...

"About These Examples" on page 9-13

"Testing GDP by OLS Regression with a Stationary Component" on page 9-14

"Testing T-Bill Rate by OLS Regression with a Drift Component" on page 9-17

About These Examples

The following examples make use of two economic time series from the U.S. Federal Reserve Economic Data (FRED) Web site, maintained by the Federal Reserve Bank of St. Louis: `http://research.stlouisfed.org/fred`.

The first example is a quarterly time series of seasonally adjusted, annualized, real Gross Domestic Product (GDP) of the United States from January 1, 1947 to April 1, 2005. It is quoted in billions of year 2000 U.S. dollars, for a total of 234 quarterly observations (Series GDPC96).

The second is a monthly time series of the three-month U.S. Treasury Bill secondary market rate from January 1, 1947 to September 1, 2005. It is quoted in percent on an annualized discount rate basis, for a total of 705 monthly observations (Series TB3MS).

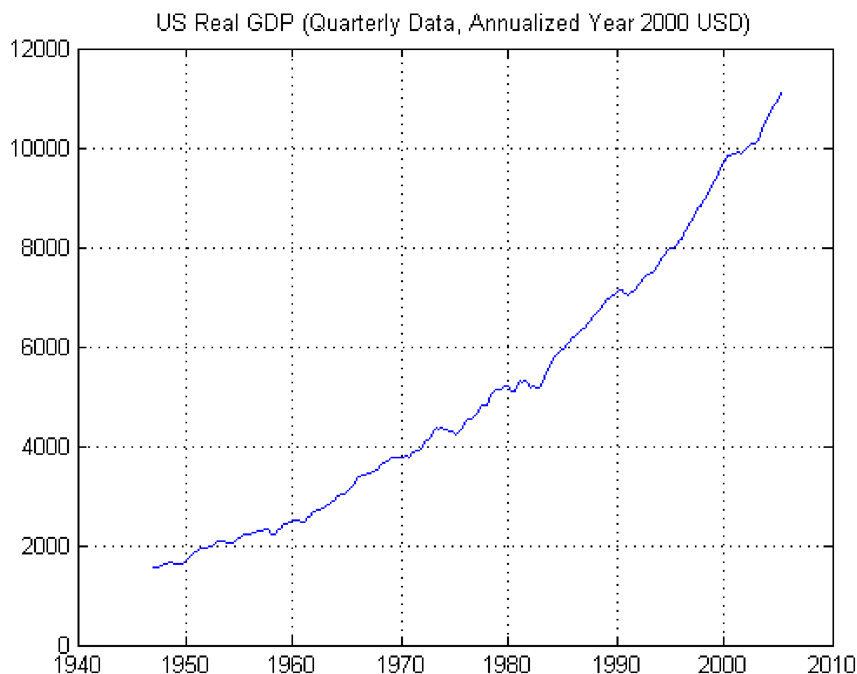
To prepare the examples, load `unitRootData`, the file that stores the observed GDP and T-Bill time series and the associated serial dates:

```
load unitRootData
whos
  Name              Size                        Bytes  Class

  GDP             234x1                          1872  double array
  GDPDates        234x1                          1872  double array
  TBillDates      705x1                          5640  double array
  TBillRates      705x1                          5640  double array

Grand total is 1878 elements using 15024 bytes
```

Testing GDP by OLS Regression with a Stationary Component

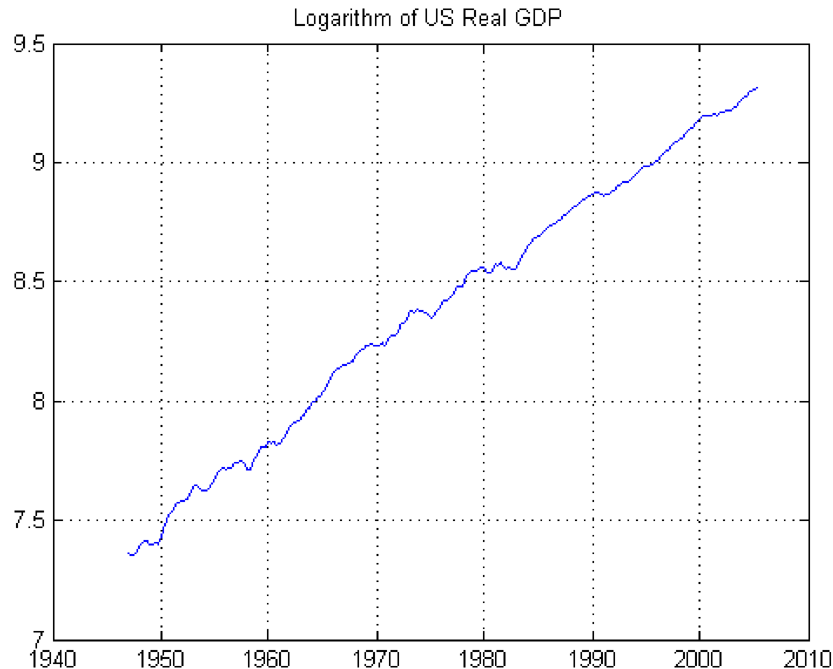
1 To launch the first example, plot GDP against time:

```
plot(GDPDates, GDP), datetick('x'), grid('on')
title('US Real GDP (Quarterly Data, Annualized Year 2000 USD)')
```



2 This plot suggests exponential growth in the time series for real GDP. Therefore, take the logarithm of GDP data to obtain a linear time trend in the plot:

```
y = log(GDP);
plot(GDPDates, y), datetick('x'), grid('on')
title('Logarithm of US Real GDP')
```



3 Base your unit root test on an OLS regression model that includes a trend stationary component. Compare results of the Dickey-Fuller and Phillips-Perron trend stationary t tests, using `dfTSTest` and `ppTSTest` at several common lags and at the 5% significance level:

```
[h, pValue, tStat, cValue] = dfTSTest(y, [0:4], ...
[0.05 0.05 0.05 0.05 0.05], 't');
[H, PValue, TStat, CValue] = ppTSTest(y, [0:4], 0.05); ...
```

Note the input argument lists in this example. The inputs to the Dickey-Fuller test explicitly specify a 5% significance level and a studentized t test for all tests. In contrast, the call to the Phillips-Perron test specifies a scalar 5% significance level. This is scalar-expanded to match the length of the `Lags` input. In addition, when you do not specify a `TestType`, the syntax for the Phillips-Perron test accepts the default t test.

All elements of the logical indicator variables, `h` and `H`, are logical zero. This indicates that there is no significant statistical evidence to reject the null hypothesis of a unit root ($\Phi = 1$):

```
[h ; H]
ans =
     0     0     0     0     0
     0     0     0     0     0
```

4 Furthermore, compare the *p*-values, OLS test statistics, and critical values of the Phillips-Perron test in the first line and the Dickey-Fuller test in the second line:

```
[pValue ; PValue]
ans =
    0.6058    0.1841    0.0871    0.1964    0.3372
    0.6058    0.4755    0.3923    0.3555    0.3484

[tStat ; TStat]
ans =
   -1.9708   -2.8428   -3.2012   -2.8079   -2.5181
   -1.9708   -2.2364   -2.4059   -2.4808   -2.4952

[cValue ; CValue]
ans =
   -3.4315   -3.4315   -3.4316   -3.4317   -3.4318
   -3.4315   -3.4315   -3.4315   -3.4315   -3.4315
```

The first element of each Phillips-Perron output vector matches the first element of the Dickey-Fuller output vector. This confirms that the two tests are identical when `Lags = 0`, that is, when you make no correction for dependence.

5 Similarly, you can compare *AR* tests at unique combinations of lags and significance levels:

```
[h, pValue, tStat, cValue] = dfTSTest(y, [0 1 2]', ...
  [0.01 0.025 0.05], 'AR');
[H, PValue, TStat, CValue] = ppTSTest(y, [1 2 3]',...
  [0.01 0.05 0.075], 'AR');
[h  H]
ans =
     0     0
     0     0
```

```
     0     0

[pValue  PValue]
ans =
    0.6683    0.5199
    0.1721    0.4156
    0.0717    0.3671

[tStat  TStat]
ans =
   -7.2381   -9.4831
  -15.1154  -11.0620
  -19.4593  -11.7962

[cValue  CValue]
ans =
  -28.3742  -28.3742
  -24.3272  -21.1594
  -21.1554  -19.2414
```

6 Now examine the Dickey-Fuller joint F test of a unit root ($\Phi = 1$) with zero trend stationary coefficient ($\delta = 0$) under the same conditions:

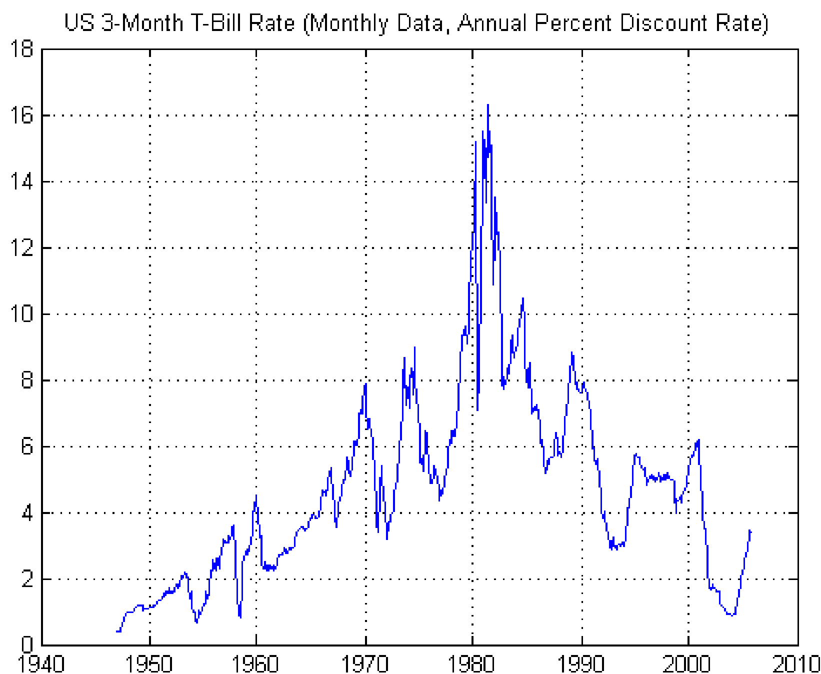
```
[h, pValue, tStat, cValue] = dfTSTest(y, ...
   [0 1 2]', [0.01 0.025 0.05], 'F');
[h, pValue, tStat, cValue]
ans =
         0    0.6207    2.5622    8.4814
         0    0.1891    4.4635    7.2834
         0    0.0884    5.5669    6.3548
```

In each of these comparisons, the row-and-column orientation of the `Lags` input vector determines the row-and-column orientation of the output vectors.

Testing T-Bill Rate by OLS Regression with a Drift Component

1 To start the second example, plot the three-month T-Bill rate against time:

```
plot(TBillDates, TBillRates), datetick('x'), grid('on')
title('US 3-Month T-Bill Rate (Monthly Data, ...
  Annual Percent Discount Rate)')
```



2 The interest rate for Treasury Bills does not exhibit a time trend. However, the plot of three-month T-Bill data suggests that the OLS regression model should include an additive constant to account for drift. To incorporate drift in your model, use `ppARDTest` and `dfARDTest`. This example compares the results of these two tests:

```
[h, pValue, tStat, cValue] = dfARDTest(TBillRates, [0:4]', ...
0.05, 't');
[H, PValue, TStat, CValue] = ppARDTest(TBillRates, [0:4]');
```

3 The following call to the Phillips-Perron test uses the default significance level for input `Alpha`, 0.05, and the default `TestType`, *t*.

Now compare the results:

```
[h, H, pValue, PValue, tStat, TStat, cValue, CValue]
ans =
0 0    0.2133    0.2133   -2.1889   -2.1889   -2.8667   -2.8667
1.0000 0    0.0428    0.1280   -2.9286   -2.4531   -2.8667   -2.8667
0 0    0.1313    0.1166   -2.4409   -2.4995   -2.8667   -2.8667
0 0    0.1191    0.1182   -2.4887   -2.4925   -2.8667   -2.8667
0 0    0.1235    0.1204   -2.4700   -2.4831   -2.8667   -2.8667
```

4 These results indicate that the null hypothesis of a unit root is rejected in the second Dickey-Fuller case at the 5% significance level, where `H = 1` at `Lags = 1`. If, however, you test a first-order correction at a significance level smaller than the reported p-value of 0.0428, such as 0.03, the null hypothesis is now accepted:

```
[h,pValue,tStat,cValue] = dfARDTest(TBillRates,1,0.03,'t');
[h,pValue,tStat,cValue]
ans =
         0    0.0428   -2.9286   -3.0633
```

5 Lastly, examine the Dickey-Fuller joint *F* test of a unit root ($\Phi = 1$) with zero drift ($C = 0$):

```
[h,pValue,tStat,cValue] = dfARDTest(TBillRates,[0 2 4]', ...
[0.01 0.02 0.1]);
[h,pValue,tStat,cValue]
ans =
         0    0.2133   -2.1889   -3.4405
         0    0.1313   -2.4409   -3.2078
         0    0.1235   -2.4700   -2.5696
```

10

Model Selection and Analysis

Using The Autocorrelation and Partial Autocorrelation Functions (p. 10-2)	How to use the autocorrelation and partial autocorrelation functions for model selection and assessment
Likelihood Ratio Tests (p. 10-3)	Uses likelihood ratio tests to determine whether evidence exists to support the use of a specific GARCH model
Akaike and Bayesian Information Criteria (p. 10-6)	Uses Akaike (AIC) and Bayesian (BIC) information criteria to compare alternative models
Equality Constraints and Parameter Significance (p. 10-9)	Sets and constrains model parameters to assess their significance
Equality Constraints and Initial Parameter Estimates (p. 10-14)	Shows the need for a complete model specification when you specify equality constraints
Examples: Simplicity and Parsimony (p. 10-17)	Why you should use the smallest, simplest model that adequately describes your data

Using The Autocorrelation and Partial Autocorrelation Functions

See “Example: Analysis and Estimation Using the Default Model” on page 2-16 for information about using the autocorrelation and partial autocorrelation functions as qualitative guides in the process of model selection and assessment. This example also introduces the following functions:

- The Ljung-Box-Pierce Q-test
- Engle’s ARCH test functions

Likelihood Ratio Tests

Testing Support for a GARCH(2,1) Model

"Example: Analysis and Estimation Using the Default Model" on page 2-16 shows that the default GARCH(1,1) model explains most of the variability of the daily returns observations of the Deutschemark/British Pound foreign-exchange rate. This example uses the `lratiotest` function to determine whether evidence exists to support the use of a GARCH(2,1) model. The example first fits the Deutschmark/British Pound foreign-exchange rate return series to the default GARCH(1,1) model. It then fits the same series using the following, more elaborate, GARCH(2,1) model:

$$y_t = C + \varepsilon_t$$

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

1 If the Deutschmark/British Pound foreign-exchange rate data is not in your workspace, restore it:

```
load garchdata
dem2gbp = price2ret(DEM2GBP);
```

2 Estimate the GARCH(1,1) model:

a Create a GARCH(1,1) default model with `Display` set to `'off'`:

```
spec11 = garchset('P',1,'Q',1,'Display','off');
```

b Estimate the model, including the maximized log-likelihood function value, and display the results:

```
[coeff11,errors11,LLF11] = garchfit(spec11,dem2gbp);
garchdisp(coeff11,errors11)
Mean: ARMAX(0,0,0); Variance: GARCH(1,1)

Conditional Probability Distribution: Gaussian
Number of Model Parameters Estimated: 4

                              Standard         T
```

```
  Parameter       Value           Error         Statistic
 ----------   -----------   ------------   -----------
          C    -6.1919e-005   8.4331e-005     -0.7342
          K    1.0761e-006    1.323e-007       8.1341
   GARCH(1)    0.80598        0.016561        48.6685
    ARCH(1)    0.15313        0.013974        10.9586
```

3 Estimate the GARCH(2,1) model:

a Create a GARCH(2,1) specification structure with `Display` set to `'off'`:

```
spec21 = garchset('P',2,'Q',1,'Display','off');
```

b Then estimate the GARCH(2,1) model and display the results. Again, calculate the maximized log-likelihood function value.

```
[coeff21,errors21,LLF21] = garchfit(spec21,dem2gbp);
garchdisp(coeff21,errors21)
Mean: ARMAX(0,0,0); Variance: GARCH(2,1)

Conditional Probability Distribution: Gaussian
Number of Model Parameters Estimated: 5

                                  Standard          T
  Parameter       Value            Error        Statistic
-----------   -----------   ------------   -----------
          C    -5.0071e-005   8.4756e-005     -0.5908
          K    1.1196e-006    1.5358e-007      7.2904
   GARCH(1)    0.49404        0.11249          4.3918
   GARCH(2)    0.2938         0.10295          2.8537
    ARCH(1)    0.16805        0.016589        10.1305
```

4 Perform the Likelihood Ratio Test.

Of the two models, GARCH(1,1) and GARCH(2,1), that are 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 [22], pages 142-144).

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

The LRT statistic is asymptotically chi-square distributed with degrees of freedom equal to the number of restrictions imposed.

a Since the GARCH(1,1) model imposes one restriction, specify one degree 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 =
         1.0000    0.0218    5.2624    3.8415
```

`H = 1` indicates that there is enough statistical evidence in support of the GARCH(2,1) model.

b Alternatively, at the `0.02` significance level:

```
[H,pValue,Stat,CriticalValue] = lratiotest(LLF21,LLF11,1,0.02);
[H,pValue,Stat,CriticalValue]
ans =
         0    0.0218    5.2624    5.4119
```

`H = 0` indicates that there is enough statistical evidence in support of the GARCH(2,1) model.

Akaike and Bayesian Information Criteria

You can use Akaike (AIC) and Bayesian (BIC) information criteria to compare alternative models. Information criteria penalize models with additional parameters. Therefore, the AIC and BIC model order selection criteria are based on parsimony (see Box, Jenkins, and Reinsel [10], pages 200-201).

The following example uses the default GARCH(1,1) and GARCH(2,1) models developed in “Likelihood Ratio Tests” on page 10-3.

1 Count the estimated parameters.

   Provide the number of parameters estimated in the model for both AIC and BIC. For the relatively simple models used here, you can just count the number of parameters. The GARCH(2,1) model estimated five parameters:

   - C
   - κ
   - G_1
   - G_2
   - A_1

   The GARCH(1,1) model estimated four parameters:

   - C
   - κ
   - G_1
   - A_1

Tip To count the estimated parameters in more elaborate models, use the function `garchcount`. `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
```

2 Compute the AIC and BIC criteria.

a To see the results more precisely, set the numeric format to `long`:

```
format long
```

Tip You can also set the numeric format by selecting **File > Preferences > Command Window > Text display** from the MATLAB® toolbar.

b Use the `aicbic` function to compute the AIC and BIC statistics for the GARCH(2,1) model and the GARCH(1,1) model, and specify the number of observations in the return series:

```
[AIC,BIC] = aicbic(LLF21,n21,1974);
[AIC BIC]
ans =
  1.0e+004 * -1.59632585502853  -1.59353194641854
[AIC,BIC] = aicbic(LLF11,n11,1974);
[AIC BIC]
ans =
  1.0e+004 * -1.59599961321328  -1.59376448632528
```

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. BIC imposes a greater penalty for additional parameters than does AIC. Thus, BIC always provides a given model with a number of parameters no greater than that chosen by AIC.

Equality Constraints and Parameter Significance

In this section...

"Specification Structure Fix Fields" on page 10-9

"Comparing the GARCH (1, 1) Estimation Results with the GARCH (2,1) Model Fit to the NASDAQ Returns" on page 10-11

Specification Structure Fix Fields

Each of these coefficient fields in the specification structure:

- `C`
- `AR`
- `MA`
- `Regress`
- `K`
- `GARCH`
- `ARCH`
- `Leverage`
- `DoF`

has a corresponding logical field that lets you hold any individual parameter fixed. These fix fields are:

- `FixC`
- `FixAR`
- `FixMA`
- `FixRegress`
- `FixK`
- `FixGARCH`
- `FixARCH`

- FixLeverage
- FixDoF

This example fits the nasdaq return series to the default GARCH(1,1) model.

1 If the nasdaq data is not already in your workspace, restore it:

```
load garchdata
nasdaq = price2ret(NASDAQ);
spec11 = garchset('P',1,'Q',1,'Display','off');
[coeff11,errors11,LLF11] = garchfit(spec11,nasdaq);
garchdisp(coeff11,errors11)

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

Conditional Probability Distribution: Gaussian
Number of Model Parameters Estimated: 4

                                    Standard          T
  Parameter        Value             Error        Statistic
 -----------    -----------     ------------    -----------
          C     0.00085852       0.00018353         4.6778
          K     2.2595e-006      3.3806e-007        6.6836
    GARCH(1)    0.87513          0.0089892         97.3531
     ARCH(1)    0.11635          0.0085331         13.6348
```

2 Since the estimated model has no equality constraints, all the fixed fields are implicitly empty; for example:

```
garchget(coeff11,'FixGARCH')
ans =
     []
```

When not empty ([]), each fix field is the same size as the corresponding coefficient field.

A 0 in a particular element of a fix field indicates that the corresponding value in its companion field is an initial parameter guess. garchfit refines this guess during the estimation process.

A 1 in a particular element of a fix field indicates that `garchfit` holds the corresponding element of its value field fixed during the estimation process (that is, an equality constraint).

Note To remove the constant C from the conditional mean model, that is, to fix `C = 0` without providing initial parameter estimates for the remaining parameters, set `C = NaN` (Not-a-Number). In this case, the value of `FixC` has no effect.

Comparing the GARCH (1, 1) Estimation Results with the GARCH (2,1) Model Fit to the NASDAQ Returns

This example compares the estimation results for the default GARCH(1,1) model with those obtained from fitting a GARCH(2,1) model to the NASDAQ returns. (See “Example Financial Time-Series Data Sets” on page 1-12.)

1 Restore your workspace as needed:

```
load garchdata
nasdaq = price2ret(NASDAQ);
```

2 Estimate the model parameters and display the results:

```
spec21 = garchset('P',2,'Q',1,'Display','off');
[coeff21,errors21,LLF21] = garchfit(spec21,nasdaq);
garchdisp(coeff21,errors21)

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

Conditional Probability Distribution: Gaussian
Number of Model Parameters Estimated: 5

                                    Standard          T
  Parameter        Value             Error        Statistic
 -----------    -----------    ------------    -----------
          C     0.00086237      0.00018378        4.6925
          K     2.3016e-006     4.7519e-007       4.8436
   GARCH(1)     0.83571         0.18533           4.5092
   GARCH(2)     0.036149        0.16562           0.2183
```

```
ARCH(1)     0.1195          0.020346          5.8734
```

The `T Statistic` column is the parameter value divided by the standard error, and is normally distributed for large samples. `T-statistic` measures the number of standard deviations the parameter estimate is away from zero. As a general rule, a `T-statistic` greater than 2 in magnitude corresponds to approximately a 95 percent confidence interval. The T-statistics shown here imply that the `GARCH(2)` parameter adds little if any explanatory power to the model.

3 Assess significance of the GARCH(2) parameter.

a Constrain the `GARCH(2)` parameter at `0`:

```
specG2 = garchset(coeff21,'GARCH',[0.8 0],'FixGARCH',[0 1]);
```

Using the `specG2` structure, `garchfit` holds `GARCH(2)` fixed at `0`, and refines `GARCH(1)` from an initial value of `0.8` during the estimation process. In other words, the `specG2` specification structure tests the composite model

$$y_t = C + \varepsilon_t$$

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

$$\sigma_t^2 = \kappa + G_1\sigma_{t-1}^2 + 0\sigma_{t-2}^2 + A_1\varepsilon_{t-1}^2$$

which is mathematically equivalent to the default GARCH(1,1) model.

b Estimate the model subject to the equality constraint, and display the results:

```
[coeffG2,errorsG2,LLFG2] = garchfit(specG2,nasdaq);
garchdisp(coeffG2,errorsG2)

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

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 4
```

```
                                  Standard          T
  Parameter        Value           Error        Statistic
 -----------    -----------    ------------    -----------
          C      0.00085827     0.00018353        4.6766
          K      2.2574e-006    3.3785e-007       6.6818
   GARCH(1)      0.87518        0.0089856        97.3979
   GARCH(2)      0              Fixed              Fixed
    ARCH(1)      0.11631        0.0085298        13.6357
```

The `Standard Error` and `T-statistic` columns for the second GARCH parameter indicate that `garchfit` holds the `GARCH(2)` 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. The results are virtually identical to those obtained from the GARCH(1,1) model.

c Apply the likelihood ratio test:

```
[H,pValue,Stat,CriticalValue] = lratiotest(LLF21,LLFG2,...
                                                1,0.05);
[H pValue Stat CriticalValue]
ans =
    0    0.7835   0.0755    3.8415
```

This is the expected result. Because the two models are virtually identical, the results support acceptance of the simpler restricted model. This is essentially just the default GARCH(1,1) model.

Equality Constraints and Initial Parameter Estimates

In this section...

"About this Example" on page 10-14
"Complete Model Specification" on page 10-14
"Empty Fix Fields" on page 10-15
"Limiting Use of Equality Constraints" on page 10-16

About this Example

This example highlights important points regarding equality constraints and initial parameter estimates in the GARCH Toolbox™ software.

For information about using the specification structure fix fields to set equality constraints, see "Specification Structure Fix Fields" on page 10-9.

Complete Model Specification

To set equality constraints during estimation, you must provide a complete model specification, that is, the specification must include initial parameter estimates consistent with the model orders. The only flexibility in this regard is that you can specify the model for either the conditional mean or the conditional variance, without specifying the other.

The following example attempts 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,'C',0,'AR',0.5,'FixAR',1,...
               'P',1,'Q',1,'K',0.0005,'GARCH',0.8,...
               'ARCH',0.1,'FixGARCH',1)
spec =

        Comment: 'Mean: ARMAX(1,1,?); Variance: GARCH(1,1)'
   Distribution: 'Gaussian'
              R: 1
              M: 1
```

```
                C: 0
               AR: 0.5000
               MA: []
    VarianceModel: 'GARCH'
                P: 1
                Q: 1
                K: 5.0000e-004
            GARCH: 0.8000
             ARCH: 0.1000
            FixAR: 1
         FixGARCH: 1
```

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 does the following:

- Ignores the C, AR, and FixAR fields.
- Computes initial parameter estimates.
- Overwrites existing parameters in the incomplete conditional mean specification.
- Estimates all conditional mean parameters (that is, C, AR, and MA).
- 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 empty fix field, ([]), is equivalent to a vector of zeros of compatible length. 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)
```

Note To remove the constant `C` from the conditional mean model, use `garchset` to set `C` to `NaN`. This fixes `C = 0` without providing initial parameter estimates for the remaining parameters. In this case, the value of `FixC` is ignored.

Limiting Use of Equality Constraints

Although the ability to set equality constraints is useful, equality constraints complicate the estimation process. For this reason, you should avoid setting several equality constraints simultaneously. For example, to really estimate a GARCH(1,1) model, specify a GARCH(1,1) model instead of a more elaborate model with numerous constraints.

Examples: Simplicity and Parsimony

As a general rule, you should specify the smallest, simplest 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 AIC and BIC discussed in the section Chapter 10, “Model Selection and Analysis”, penalize models for their complexity.

Diagnostic tools such as the autocorrelation function (ACF) and partial autocorrelation function (PACF), are recommended for guiding model selection. For example, the section “Example: Analysis and Estimation Using the Default Model” on page 2-16 examines the ACF and PACF of the Deutschmark/British Pound foreign-exchange rate (see “Example Financial Time-Series Data Sets” on page 1-12). The results support the use of a simple constant for the conditional mean model as adequate to describe the data.

The following example illustrates a complicated model specification. It simulates a return series as a pure GARCH(1,1) innovations process (that is, the default model). It then attempts to overfit an ARMA(1,1)/GARCH(1,1) composite model to the data.

1. Create a specification structure for the innovations process and simulate the returns:

```
spec = garchset('C',0,'K',0.00005,'GARCH',0.85,'ARCH',0.1,...
                'Display','off');
randn('state',0);
rand('twister',0);
[e,s,y] = garchsim(spec,5000,1);
```

2. Fit the default model to the known GARCH(1,1) innovations process and display the estimation results:

```
[coeff,errors] = garchfit(spec,y);
garchdisp(coeff,errors)

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

  Conditional Probability Distribution: Gaussian
```

```
  Number of Model Parameters Estimated: 4

                                      Standard          T
  Parameter        Value               Error        Statistic
 -----------    -----------    ------------    -----------
           C    -5.8129e-005    0.0004096        -0.1419
           K     4.6408e-005    8.3396e-006       5.5648
    GARCH(1)     0.85994        0.014612         58.8515
     ARCH(1)     0.095354       0.0097535         9.7765
```

These estimation results indicate that the model that best fits the observed data is approximately

$$y_t = -5.8129e^{-005} + \varepsilon_t$$

$$\sigma_t^2 = 4.6408e^{-005} + 0.85994\sigma_{t-1}^2 + 0.95354\varepsilon_{t-1}^2$$

3 Continue by fitting the known GARCH(1,1) innovations process to an ARMA(1,1) mean model, and display the estimation results:

```
spec11   = garchset(spec,'R',1,'M',1);
[coeff11,errors11]     = garchfit(spec11,y);
garchdisp(coeff11,errors11)

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

  Conditional Probability Distribution: Gaussian
  Number of Model Parameters Estimated: 6

                                      Standard          T
  Parameter        Value               Error        Statistic
 -----------    -----------    ------------    -----------
           C    -7.1366e-005    0.00052468       -0.1360
       AR(1)    -0.24509        0.32706          -0.7494
       MA(1)     0.28515        0.32362           0.8811
           K     4.6868e-005    8.4098e-006       5.5731
    GARCH(1)     0.85917        0.014733         58.3160
     ARCH(1)     0.095584       0.0097975         9.7560
```

4 Examine the results. Close examination of the conditional mean equation reveals that the AR(1) and MA(1) parameters are similar. In fact, when rewriting the mean equation in backshift (that is, lag) operator notation, where $By_t = y_{t-1}$,

$$(1+0.24509B)y_t = -7.1366e^{-005} + (1+0.28515B)\varepsilon_t$$

the autoregressive and moving-average polynomials come close to canceling each other (see Box, Jenkins, and Reinsel [10], pages 263-267). This is an example of *parameter redundancy*, or *pole-zero cancellation*. This supports the use of the simple default model. In fact, the more elaborate ARMA(1,1) model only complicates the analysis by requiring the estimation of two additional parameters.

Example Workflow: Estimation, Forecasting, and Simulation

Estimating the Model (p. 11-3) — Fits ARMA(1,1) and GJR(1,1) models to the conditional mean and variance processes, respectively, of the NASDAQ return series, assuming conditionally t-distributed residuals

Forecasting (p. 11-5) — Uses the estimated model from the first part of the example to forecast the conditional standard deviations of residuals, the returns, the standard deviations of multi-period cumulative returns, and the standard errors of the forecast of returns over multiple periods

Forecasting Using Monte Carlo Simulation (p. 11-7)	Uses the estimated model from the first part of the example and vector-format presample data to perform dependent-path Monte Carlo simulation of multiple realizations
Comparing Forecasts with Simulation Results (p. 11-9)	Illustrates the relationship between forecasting and dependent-path Monte Carlo simulation by comparing and contrasting the forecasts with their counterparts derived from the Monte Carlo simulation

Estimating the Model

The first part of the example fits the NASDAQ daily returns to an ARMA(1,1)/GJR(1,1) model with conditionally t-distributed residuals. (See "Example Financial Time-Series Data Sets" on page 1-12 for more information about the NASDAQ Composite Index data set.)

1 Load the `nasdaq` data set and convert daily closing prices to daily returns:

```
load garchdata
nasdaq = price2ret(NASDAQ);
```

2 Create a specification structure for an ARMA(1,1)/GJR(1,1) model with conditionally t-distributed residuals:

```
spec = garchset('VarianceModel','GJR',...
                'R',1,'M',1,'P',1,'Q',1);
spec = garchset(spec,'Display','off','Distribution','T');
```

Note This example is for illustration purposes only. Such an elaborate ARMA(1,1) model is typically not needed, and should only be used after you have performed a sound pre-estimation analysis.

---

3 Estimate the parameters of the mean and conditional variance models via `garchfit`. Make sure that the example returns the estimated residuals and conditional standard deviations inferred from the optimization process, so that they can be used as presample data:

```
[coeff,errors,LLF,eFit,sFit] = garchfit(spec,nasdaq);
```

Alternatively, you could replace this call to `garchfit` with the following successive calls to `garchfit` and `garchinfer`. This is because the estimated residuals and conditional standard deviations are also available from the inference function `garchinfer`:

```
[coeff,errors] = garchfit(spec,nasdaq);
[eFit,sFit]    = garchinfer(coeff,nasdaq);
```

Either approach produces the same estimation results:

```
garchdisp(coeff,errors)

   Mean: ARMAX(1,1,0); Variance: GJR(1,1)

  Conditional Probability Distribution: T
  Number of Model Parameters Estimated: 8

                                 Standard          T
  Parameter       Value            Error        Statistic
 -----------   -----------   ------------   -----------
           C    0.00099709     0.00023381        4.2646
       AR(1)    -0.10719       0.11571          -0.9264
       MA(1)    0.26272        0.11208           2.3441
           K    1.4684e-006    3.8716e-007       3.7927
    GARCH(1)    0.89993        0.011223         80.1855
     ARCH(1)    0.048844       0.013619          3.5863
 Leverage(1)    0.086624       0.016922          5.1189
         DoF    7.8274         0.9301            8.4157
```

Forecasting

The second part of the example uses the model from "Estimating the Model" on page 11-3 to compute forecasts for the nasdaq return series 30 days into the future.

1 Set the forecast horizon to 30 days (one month):

```
horizon = 30;  % Define the forecast horizon
```

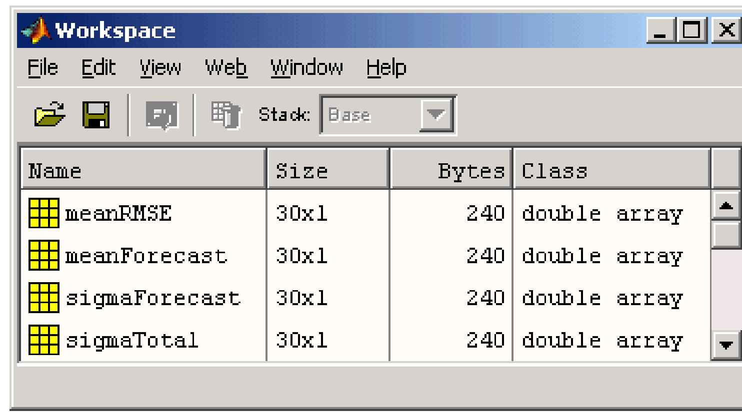
2 Call the forecasting engine, garchpred, with the estimated model parameters, coeff, the nasdaq returns, and the forecast horizon:

```
[sigmaForecast,meanForecast,sigmaTotal,meanRMSE] = ...
      garchpred(coeff,nasdaq,horizon);
```

This call to garchpred returns the following parameters:

- Forecasts of conditional standard deviations of the residuals (sigmaForecast)
- Forecasts of the nasdaq returns (meanForecast)
- Forecasts of the standard deviations of the cumulative holding period nasdaq returns (sigmaTotal)
- Standard errors associated with forecasts of nasdaq returns (meanRMSE)

Because the return series nasdaq is a vector, all garchpred outputs are vectors. Because garchpred uses iterated conditional expectations to successively update forecasts, all garchpred outputs have 30 rows. The first row stores the 1-period-ahead forecasts, the second row stores the 2-period-ahead forecasts, and so on. Thus, the last row stores the forecasts at the 30-day horizon.


Workspace
File Edit View Web Window Help
Stack: Base
Name Size Bytes Class
meanRMSE 30x1 240 double array
meanForecast 30x1 240 double array
sigmaForecast 30x1 240 double array
sigmaTotal 30x1 240 double array

Forecasting Using Monte Carlo Simulation

The third part of the example uses the same estimated model, `coeff`, as in "Forecasting" on page 11-5. This part of the example simulates 20000 realizations for the same 30-day period.

The example explicitly specifies the needed presample data:

- It uses the inferred residuals (`eFit`) and standard deviations (`sFit`) from "Estimating the Model" on page 11-3 as the presample inputs `PreInnovations` and `PreSigmas`, respectively.
- It uses the `nasdaq` return series as the presample input `PreSeries`.

Because all inputs are vectors, `garchsim` applies the same vector to each column of the corresponding outputs, `Innovations`, `Sigmas`, and `Series`. In this context, called *dependent-path simulation*, all simulated sample paths share a common conditioning set and evolve from the same set of initial conditions. This enables Monte Carlo simulation of forecasts and forecast error distributions.

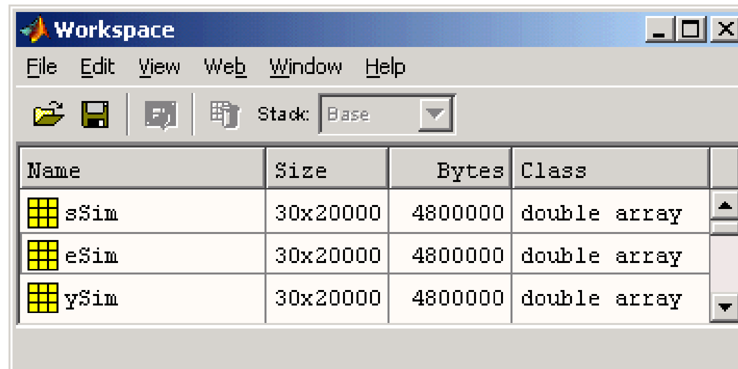
Specify `PreInnovations`, `PreSigmas`, and `PreSeries` as matrices, where each column is a realization, or as single-column vectors. In either case, they must have a sufficient number of rows to initiate the simulation (see "Running Simulations With User-Specified Presample Data" on page 4-13).

For this application of Monte Carlo simulation, the example generates a relatively large number of realizations, or sample paths, so that it can aggregate across realizations. Because each realization corresponds to a column in the `garchsim` time-series output arrays, the output arrays are large, with many columns.

1. Simulate 20000 paths (columns):

```
nPaths = 20000;  % Define the number of realizations.
randn('state',0);
rand('twister',0);
[eSim,sSim,ySim] = garchsim(coeff,horizon,nPaths,...
[],[],[], eFit,sFit,nasdaq);
```

Each time-series output that `garchsim` returns is an array of size `horizon`-by-`nPaths`, or 30-by-20000. Although more realizations (for example, 100000) provide more accurate simulation results, you may want to decrease the number of paths (for example, to 10000) to avoid memory limitations.



2 Because `garchsim` needs only the last, or most recent, observation of each, the following command produces identical results:

```
randn('state',0);
rand('twister',0);
[eSim,sSim,ySim] = garchsim(coeff,horizon,nPaths, ...
   [],[],[], eFit(end),sFit(end),nasdaq(end));
```

Comparing Forecasts with Simulation Results

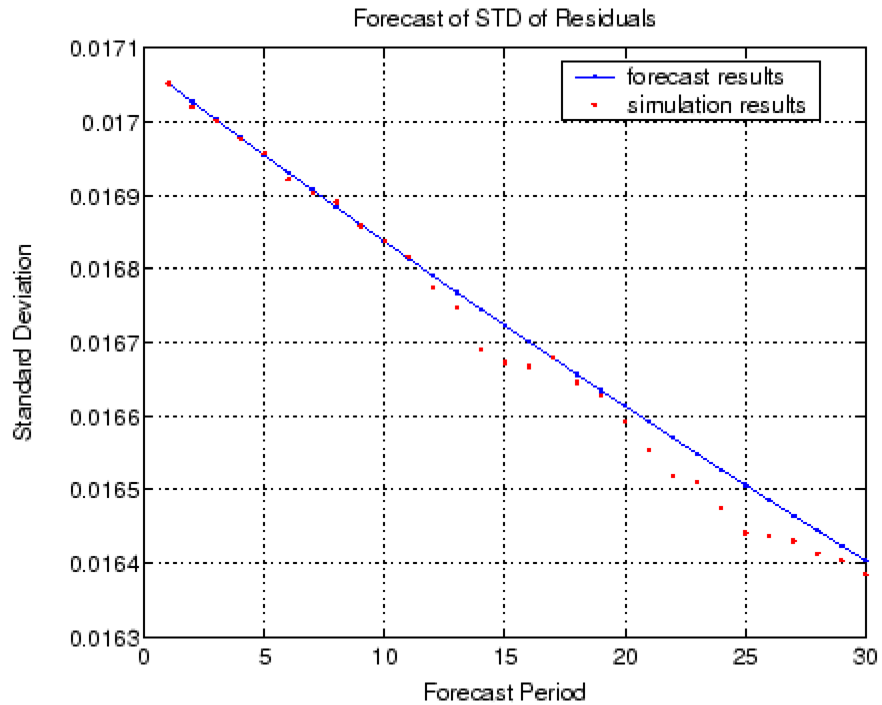
The fourth part of this example graphically compares the forecasts from "Forecasting" on page 11-5 with their counterparts derived from "Forecasting Using Monte Carlo Simulation" on page 11-7. The first four figures compare directly each of the garchpred outputs, in turn, with the corresponding statistical result obtained from simulation. The last two figures illustrate histograms from which you can compute approximate probability density functions and empirical confidence bounds.

Note For an EGARCH model, multi-period MMSE forecasts are generally downward-biased and underestimate their true expected values for conditional variance forecasts. This is not true for one-period-ahead forecasts, which are unbiased in all cases. For unbiased multi-period forecasts of sigmaForecast, sigmaTotal, and meanRMSE, you can perform Monte Carlo simulation using garchsim. For more information, see "Asymptotic Behavior for Long-Range Forecast Horizons" on page 7-7.

---

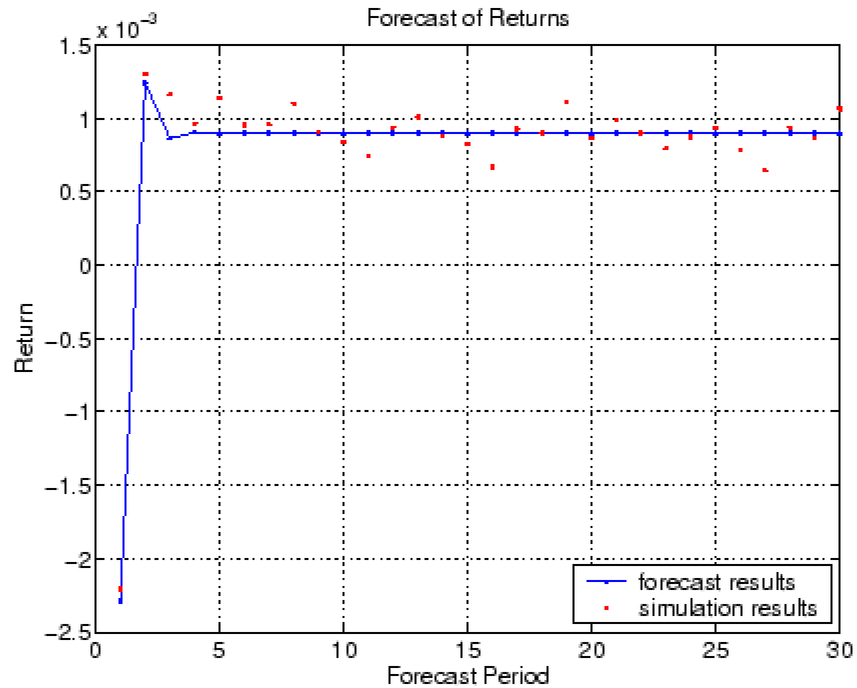
1 Compare the first garchpred output, sigmaForecast (the conditional standard deviations of future innovations), with its counterpart derived from the Monte Carlo simulation:

```
figure
plot(sigmaForecast,'.-b')
hold('on')
grid('on')
plot(sqrt(mean(sSim.^2,2)),'.r')
title('Forecast of STD of Residuals')
legend('forecast results','simulation results')
xlabel('Forecast Period')
ylabel('Standard Deviation')
```



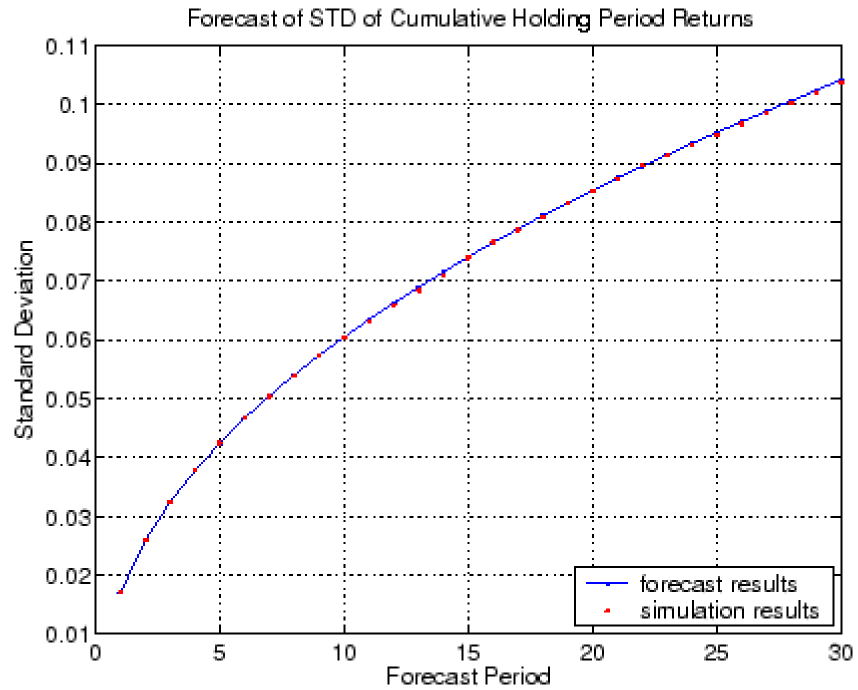
2 Compare the second `garchpred` output, `meanForecast`(the MMSE forecasts of the conditional mean of the `nasdaq` return series), with its counterpart derived from the Monte Carlo simulation:

```
figure(2)
plot(meanForecast,'.-b')
hold('on')
grid('on')
plot(mean(ySim,2),'.r')
title('Forecast of Returns')
legend('forecast results','simulation results',4)
xlabel('Forecast Period')
ylabel('Return')
```



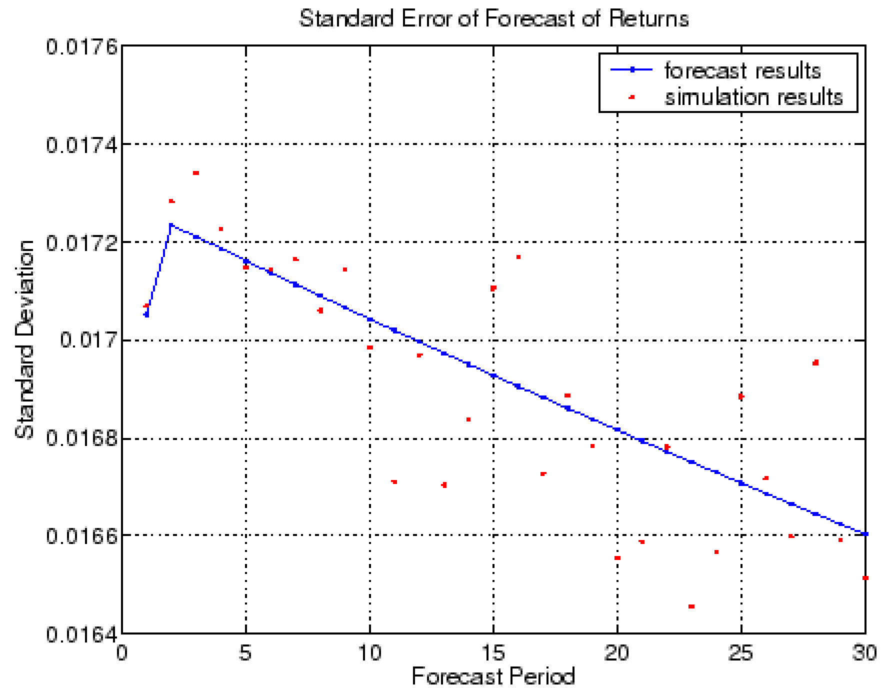
3 Compare the third `garchpred` output, `sigmaTotal`, that is, cumulative holding period returns, with its counterpart derived from the Monte Carlo simulation:

```
holdingPeriodReturns = log(ret2price(ySim,1));
figure(3)
plot(sigmaTotal,'.-b')
hold('on')
grid('on')
plot(std(holdingPeriodReturns(2:end,:)'),'.r')
title('Forecast of STD of Cumulative Holding Period Returns')
legend('forecast results','simulation results',4)
xlabel('Forecast Period')
ylabel('Standard Deviation')
```



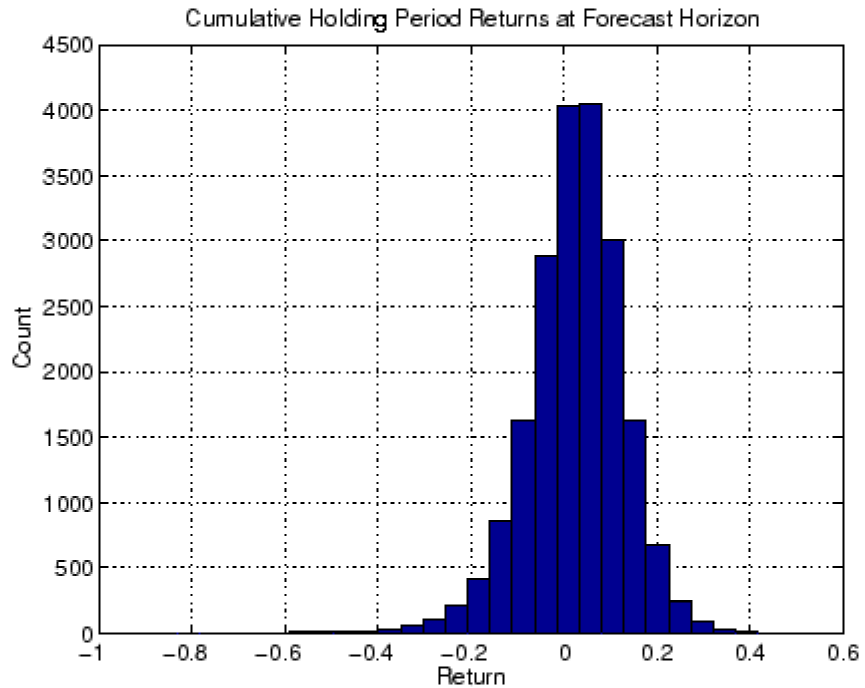
4 Compare the fourth `garchpred` output, `meanRMSE`, that is the root mean square errors (RMSE) of the forecasted returns, with its counterpart derived from the Monte Carlo simulation:

```
figure(4)
plot(meanRMSE,'.-b')
hold('on')
grid('on')
plot(std(ySim'),'.r')
title('Standard Error of Forecast of Returns')
legend('forecast results','simulation results')
xlabel('Forecast Period')
ylabel('Standard Deviation')
```



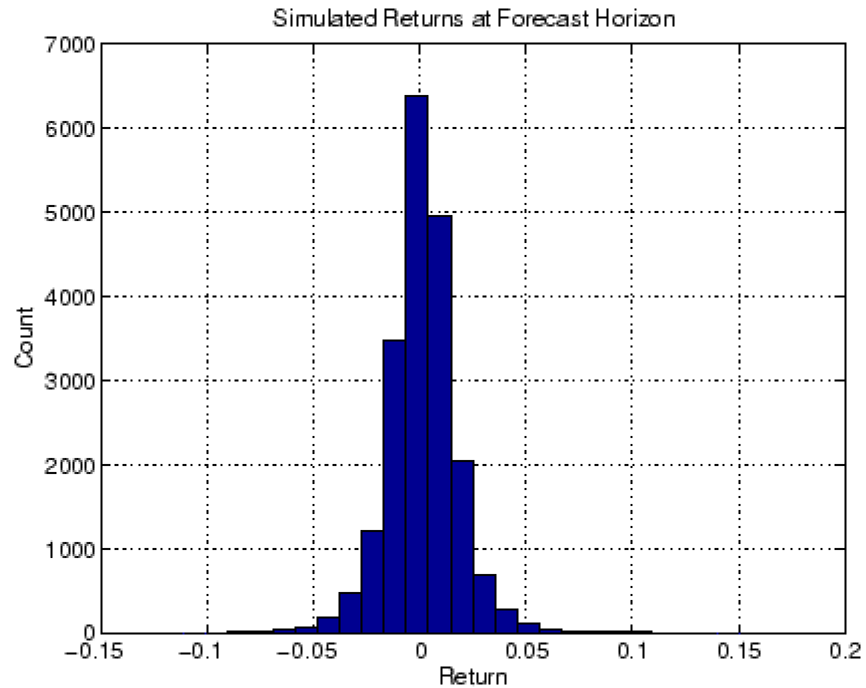
5 Use a histogram to illustrate the distribution of the cumulative holding period return obtained if an asset was held for the full 30-day forecast horizon. That is, plot the return obtained by purchasing a mutual fund that mirrors the performance of the NASDAQ Composite Index today, and sold after 30 days. This histogram is directly related to the final red dot in step 3:

```
figure(5)
hist(holdingPeriodReturns(end,:),30)
grid('on')
title('Cumulative Holding Period Returns at Forecast Horizon')
xlabel('Return')
ylabel('Count')
```



6 Use a histogram to illustrate the distribution of the single-period return at the forecast horizon, that is, the return of the same mutual fund, the 30th day from now. This histogram is directly related to the final red dots in steps 2 and 4:

```
figure(6)
hist(ySim(end,:),30)
grid('on')
title('Simulated Returns at Forecast Horizon')
xlabel('Return')
ylabel('Count')
```



Note Detailed analyses of such elaborate conditional mean and variance models are not usually required to describe typical financial time series. Furthermore, such a graphical analysis may not necessarily make sense for a given model. This example is intended to highlight the range of possibilities and provide a deeper understanding of the interaction between the simulation, forecasting, and estimation engines. For more information, see `garchsim`, `garchpred`, and `garchfit`.

12

Function Reference

Data Preprocessing (p. 12-2)	Filters, smoothers, and transformations
GARCH Specification Structure (p. 12-2)	Manipulate GARCH specification structures
GARCH Modeling (p. 12-3)	Model estimation, forecasting, and Monte Carlo simulation
General Utilities (p. 12-3)	Utilities for general GARCH modeling tasks
Graphics (p. 12-4)	Time series visualization
Statistics and Tests (p. 12-4)	Compute statistics and perform tests

Data Preprocessing

`hpfilter`	Run Hodrick-Prescott filter

GARCH Specification Structure

`garchget`	Get value of GARCH specification structure parameter
`garchset`	Create or modify GARCH specification structure

GARCH Modeling

`garchfit`	Estimate univariate GARCH process parameters
`garchpred`	Perform univariate GARCH process forecasting
`garchsim`	Perform univariate GARCH process simulation

General Utilities

`garchar`	Convert finite-order ARMA models to infinite-order autoregressive (AR) models
`garchcount`	Count number of GARCH estimation coefficients
`garchdisp`	Display GARCH process estimation results
`garchinfer`	Infer GARCH innovation processes from return series
`garchma`	Convert finite-order ARMA models to infinite-order moving average (MA) models
`lagmatrix`	Create lagged time-series matrix
`price2ret`	Convert price series to return series
`ret2price`	Convert return series to price series

Graphics

`garchplot`	Plot matched univariate innovations, volatility, and return series

Statistics and Tests

`aicbic`	Calculate Akaike (AIC) and Bayesian (BIC) information criteria for model order selection
`archtest`	Run Engle's hypothesis test to detect presence of ARCH/GARCH effects
`autocorr`	Plot or return computed sample autocorrelation function
`crosscorr`	Plot or return computed sample cross-correlation function
`dfARDTest`	Run augmented Dickey-Fuller unit root test based on AR model with drift
`dfARTest`	Run augmented Dickey-Fuller unit root test based on zero drift AR model
`dfTSTest`	Run augmented Dickey-Fuller unit root test based on trend stationary AR model
`lbqtest`	Run Ljung-Box Q-statistic lack-of-fit hypothesis test
`lratiotest`	Run Likelihood ratio hypothesis test
`parcorr`	Plot or return computed sample partial autocorrelation function
`ppARDTest`	Run Phillips-Perron unit root test based on AR(1) model with drift

`ppARTest`	Run Phillips-Perron unit root test based on zero drift AR(1) model
`ppTSTest`	Run Phillips-Perron unit root test based on trend stationary AR(1) model

Functions — Alphabetical List

Purpose

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

Syntax

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

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(LLF,NumParams)` computes only the Akaike (AIC) information criteria.
- `[AIC,BIC] = aicbic(LLF,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.

Input Arguments

`LLF`	Vector of optimized log-likelihood objective function (LLF) values associated with parameter estimates of the models to be tested. `aicbic` assumes that 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 `LLF`. `NumParams` can be a scalar applied to all values in `LLF`, or a vector the same length as `LLF`. 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 `LLF`. `NumObs` can be a scalar applied to all values in `LLF`, or a vector the same length as `LLF`.

It is required to compute BIC. All elements of `NumObs` must be positive integers.

Output Arguments

`AIC`	Vector of AIC statistics associated with each `LLF` objective function value. The AIC statistic is defined as `AIC` = (-2*`LLF`) + (2*`NumParams`)
`BIC`	Vector of BIC statistics associated with each `LLF` objective function value. The BIC statistic is defined as `BIC` = = (-2*`LLF`) + (`NumParams` * log(`NumObs`))

Examples

See "Akaike and Bayesian Information Criteria" on page 10-6.

See Also

`garchdisp`, `garchfit`, `garchinfer`

References

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

Purpose Run Engle's hypothesis test to detect presence of ARCH/GARCH effects

Syntax

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

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; that is, that no ARCH effects exist.

Given sample residuals obtained from a curve fit (for example, a regression model), `archtest` tests for the presence of Mth order ARCH effects. It does so 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.
- 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.

Input Arguments

`Residuals`	Time-series column vector of sample residuals obtained from a curve fit, which `archtest` examines for the presence of ARCH effects. The last row contains the most recent observation.
`Lags`	Vector of positive integers indicating the lags of the squared sample residuals included in the ARCH test statistic. If specified, each lag should be less than the

length of `Residuals`. If `Lags = []` or is unspecified, the default is `1` lag (that is, first-order ARCH).

`Alpha`	Significance levels 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 unspecified, the default is `0.05`. For all elements, α of `Alpha`, $0 < \alpha < 1$.

Output Arguments

`H`	Boolean decision vector. `0` indicates acceptance of the null hypothesis that no ARCH effects exist; that is, there is homoscedasticity 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`.

Examples

Example 1

Create a time-series column vector of 100 (synthetic) residuals, then test for the first, second, and fourth 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
```

Example 2

See "Example: Analysis and Estimation Using the Default Model" on page 2-16 for another example.

See Also

`lbqtest`

References

Box, G.E.P., G.M. Jenkins, and 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, 1982, pp. 987-1007.

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

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

Purpose

Plot or return computed sample autocorrelation function

Syntax

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

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.

Input Arguments

`Series`	Column vector of observations of a univariate time series for which `autocorr` computes or plots the sample autocorrelation function (ACF). The last row of `Series` contains the most recent observation of the time series.
`nLags`	Positive scalar integer indicating the number of lags of the ACF to compute. If `nLags = []` or is unspecified, the default is to compute the ACF at lags 0, 1, 2, ..., T, where T = `min([20,length(Series)-1])`.
`M`	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 greater than `M`. If `M = []` or is unspecified, 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 $\frac{1}{\sqrt{N}}$. `M` must be less than `nLags`.
`nSTDs`	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 $\pm(\frac{nSTDs}{\sqrt{N}})$. If `nSTDs = []` or is unspecified, the default is `2` (that is, approximate 95 percent confidence interval).

Output Arguments

`ACF`	Sample autocorrelation 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. `autocorr` computes `Bounds` only for lags greater than `M`.

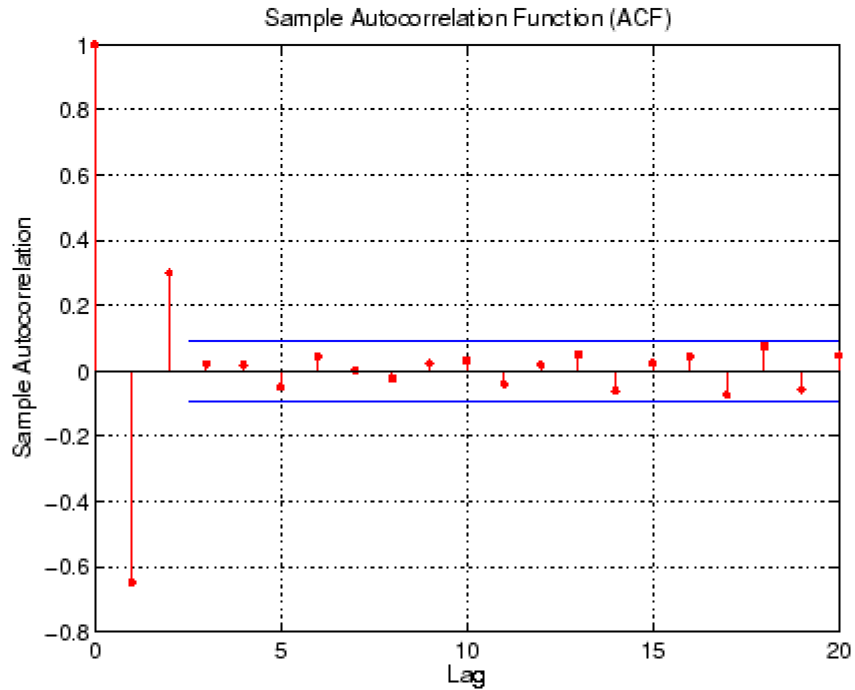
Examples

Example 1

Create an MA(2) time series from a column vector of 1000 Gaussian deviates. Then, assess whether the ACF is effectively zero for lags greater than 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.
```

```
% Compute the ACF with 95 percent confidence.
[ACF, Lags, Bounds] = autocorr(y, [], 2);
[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

Bounds =
    0.0899
   -0.0899

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



Example 2

Although various estimates of the sample autocorrelation function exist, the form adopted here follows that of Box, Jenkins, and Reinsel, specifically:

$$r_k = \frac{c_k}{c_0} \qquad \textbf{(13-1)}$$

$$c_k = \frac{1}{N}\sum_{t=1}^{N-k}(z_t - \bar{z})(z_{(t+k)} - \bar{z}) \qquad k = 0,1,2,...K \qquad \textbf{(13-2)}$$

The `autocorr` function computes the sample ACF by removing the sample mean of the input Series, then normalizing the sequence such

that the ACF at lag zero is unity. In certain applications, it is useful to rescale the resulting normalized ACF by the sample variance. In this case, the correct scale factor to use is `var(Series,1)`.

The following commands simulate 1000 standard Gaussian random numbers, then compares the first 10 lags of the sample ACF with and without normalization:

```
randn('state', 0);
y = randn(1000, 1);
[ACF, Lags] = autocorr(y, 10);
[Lags  ACF  ACF*var(y,1)]
ans =
         0    1.0000    0.8893
    1.0000    0.0111    0.0099
    2.0000   -0.0230   -0.0205
    3.0000    0.0194    0.0173
    4.0000    0.0068    0.0061
    5.0000   -0.0371   -0.0330
    6.0000    0.0241    0.0215
    7.0000    0.0101    0.0090
    8.0000   -0.0011   -0.0010
    9.0000    0.0577    0.0513
   10.0000    0.0526    0.0467
```

Example 3

See “Example: Analysis and Estimation Using the Default Model” on page 2-16.

See Also

`crosscorr`, `parcorr`

`filter` (MATLAB® function)

References

Box, G.E.P., G.M. Jenkins, and 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

Plot or return computed sample cross-correlation function

Syntax

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

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.

Input Arguments

`Series1`	Column vector of observations of the first univariate time series for which `crosscorr` computes or plots the sample cross-correlation function (XCF). The last row of `Series1` contains the most recent observation.
`Series2`	Column vector of observations of the second univariate time series for which `crosscorr` computes or plots the sample XCF. The last row of `Series2` contains the most recent observation.
`nLags`	Positive scalar integer indicating the number of lags of the XCF to compute. If `nLags = []` or is unspecified, `crosscorr` computes the XCF at lags $0, \pm 1, \pm 2, \ldots, \pm T$, where T `= min([20,min([length(Series1), length(Series2)])-1])`.
`nSTDs`	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 unspecified, the default is `2` (that is, approximate 95 percent confidence interval).

Output Arguments

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

Examples

Example 1

1 Create a time-series column vector of 100 Gaussian deviates:

```
randn('state', 100)        % Start from a known state
x = randn(100, 1);         % 100 Gaussian deviates, N(0, 1)
```

2 Create a delayed version of the vector, lagged by four samples:

```
y = lagmatrix(x, 4);    % Delay it by 4 samples
```

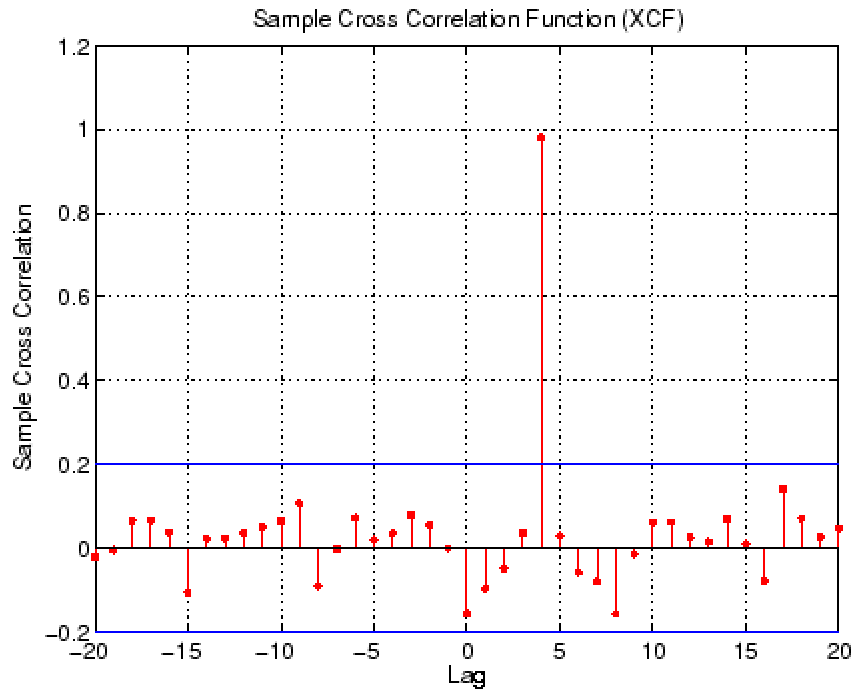
3 Compute the XCF, and then plot it to see the XCF peak at the fourth lag:

```
y(isnan(y)) = 0;
[XCF, Lags, Bounds] = crosscorr(x, y);
[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
```

```
Bounds =
    0.2000
   -0.2000
crosscorr(x, y)        % Use the same example, but plot the XCF
                       % sequence. Note the peak at the 4th lag.
```



Example 2

See “Example: Analysis and Estimation Using the Default Model” on page 2-16.

See Also

`autocorr`, `parcorr`

`filter` (MATLAB® function)

Purpose

Run augmented Dickey-Fuller unit root test based on AR model with drift

Syntax

```
[H,pValue,TestStat,CriticalValue] = ...
  dfARDTest(Y,Lags,Alpha,TestType)
```

Description

`[H,pValue,TestStat,CriticalValue] = ...`
`  dfARDTest(Y,Lags,Alpha,TestType)` performs an augmented Dickey-Fuller univariate unit root test. This test assumes that the true underlying process is a zero drift unit root process. As an alternative, OLS regression estimates a (P+1)th order autoregressive (AR(P+1)) model plus additive constant.

Specifically, if y_t and ε_t are the time series of observed data and model residuals, respectively, and $\Delta y_t = y_t - y_{t-1}$ is the first difference operator, then under the null hypothesis the true underlying process is a zero drift ARIMA(P,1,0) model:

$$\Delta y_t = y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots \zeta_p \Delta y_{t-p} + \varepsilon_t$$

This is equivalent to an integrated AR(P+1) model.

As an alternative, the estimated OLS regression model is

$$\Delta y_t = C + \phi y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots \zeta_p \Delta y_{t-p} + \varepsilon_t$$

for some constant C and AR(1) coefficient $\varphi < 1$.

Input Arguments

`Y`	Time-series vector of observed data tested for a unit root. The last element contains the most recent observation. `dfARDTest` represents missing values as `NaN`s and removes them, thereby reducing the sample size.
`Lags`	(Optional) Scalar or vector of nonnegative integers. This parameter indicates the number of lagged changes (that is, first differences) of `Y` included in the OLS regression model. `Lags` serves as a correction for serial correlation of residuals. If `Lags` is empty or missing, the default is `0` (no correction for serial correlation).
`Alpha`	(Optional) Scalar or vector of significance levels of the test. All elements of the input argument must be 0.001 ≤ `Alpha` ≤ 0.999.
`TestType`	(Optional) Character string indicating the type of unit root test. Possible choices are: • `t`, indicating an OLS t test of the AR(1) coefficient • `AR`, indicating a test of the unstudentized AR(1) coefficient • `F`, indicating a joint OLS F test of a unit root (`Φ = 1`) with zero drift (`C = 0`) `dfARDTest` performs a case-insensitive check of `TestType`. If it is empty or missing, the default is a `t` test.

Output Arguments

`H`	Logical decision vector. Elements of `H = 0` indicate acceptance of the null hypothesis; elements of `H = 1` indicate rejection of the null hypothesis. Each element of `H` is associated with a particular lag of `Lags` and significance level of `Alpha`.
`pValue`	Vector of p-values (significance levels) associated with the test decision vector `H`. Each element of `pValue` represents the probability of observing a test statistic at least as extreme as that calculated from the OLS regression model when the null hypothesis is true. `dfARDTest` obtains p-values by interpolation into the appropriate table of critical values. When a p-value is outside of the range of tabulated significance levels (that is `0.001 <= Alpha <=0.999`), a warning appears. `dfARDTest` then sets `pValue` to the appropriate limit (`pValue = 0.001` or `0.999`).
`TestStat`	Vector of test statistics associated with the decision vector `H`.
`CriticalValue`	Vector of critical values associated with the decision vector `H`.

Notes

You can specify both `Lags` and `Alpha` as scalars or vectors. If you specify both as vectors, they must be the same length (that is, they must have the same number of elements). If you specify one of these parameters as a scalar and the other as a vector, `dfARDTest` performs a scalar expansion to enforce vectors of identical length. If `Lags` is a scalar or an empty matrix, by default, all outputs are column vectors.

All vector outputs are the same length as vector inputs `Alpha` and/or `Lags`. By default, all vector outputs are column vectors. If `Lags` is a row vector, however, all vector outputs are row vectors.

This univariate unit root test is a conventional *single-tailed* test. `dfARDTest` compares the test statistic with the critical value to determine whether the test is accepted or rejected:

- The AR and t tests are *lower-tailed* tests. Reject the null hypothesis if the test statistic is *less than* the critical value.
- The joint F test is an *upper-tailed* test. Reject the null hypothesis if the test statistic is *greater than* the critical value.

See Also

`dfARTest`, `dfTSTest`, `ppARDTest`, `ppARTest`, `ppTSTest`

References

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

Greene, W.H., *Econometric Analysis*, Prentice Hall, Fifth edition, Upper Saddle River, NJ, 2003.

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, *The GARCH of Financial Markets*, Princeton University Press, Princeton, NJ, 1997.

Purpose

Run augmented Dickey-Fuller unit root test based on zero drift AR model

Syntax

```
[H,pValue,TestStat,CriticalValue] = dfARTest(Y,Lags,Alpha,
    TestType)
```

Description

`[H,pValue,TestStat,CriticalValue] = dfARTest(Y,Lags,Alpha,TestType)` performs an augmented Dickey-Fuller univariate unit root test. This test assumes that the true underlying process is a zero drift unit root process. As an alternative, OLS regression estimates a zero drift (P+1)th order autoregressive (AR(P+1)) model.

Specifically, if:

- y_t and ε_t are the time series of observed data and model residuals, respectively, and
- $\Delta y_t = y_t - y_{t-1}$ is the first difference operator

then under the null hypothesis the true underlying process is a zero drift ARIMA(P,1,0) model

$$\Delta y_t = y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots \zeta_p \Delta y_{t-p} + \varepsilon_t$$

Which is equivalent to an integrated AR(P+1) model.

As an alternative, the estimated OLS regression model is

$$\Delta y_t = \phi y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots \zeta_p \Delta y_{t-p} + \varepsilon_t$$

for some AR(1) coefficient $\varphi < 1$.

Input Arguments

`Y`	Time-series vector of observed data tested for a unit root. The last element contains the most recent observation. `dfARTest` represents missing values as `NaN`s and removes them, thereby reducing the sample size.
`Lags`	(Optional) Scalar or vector of nonnegative integers. This parameter indicates the number of lagged changes (that is, first differences) of `Y` included in the OLS regression model (see P above). `Lags` serves as a correction for serial correlation of residuals. If `Lags` is empty or missing, the default is `0` (no correction for serial correlation).
`Alpha`	(Optional) Scalar or vector of significance levels of the test. All elements of the input argument must be 0.001 ≤ `Alpha` ≤ 0.999.
`TestType`	(Optional) Character string indicating the type of unit root test. Possible choices are: • `t`, indicating an OLS t test of the AR(1) coefficient • `AR`, indicating a test of the unstudentized AR(1) coefficient `dfARTest` performs a case-insensitive check of `TestType`. If it is empty or missing, the default is a `t` test.

Output Arguments

`H`
: Logical decision vector. Elements of `H = 0` indicate acceptance of the null hypothesis; elements of `H = 1` indicate rejection of the null hypothesis. Each element of `H` is associated with a particular lag of `Lags` and significance level of `Alpha`.

`pValue`
: Vector of p-values (significance levels) associated with the test decision vector `H`. Each element of `pValue` represents the probability of observing a test statistic at least as extreme as that calculated from the OLS regression model when the null hypothesis is true. `dfARTest` obtains p-values by interpolation into the appropriate table of critical values.

  When a p-value is outside of the range of tabulated significance levels (that is `0.001 <= Alpha <=0.999`), a warning appears. `dfARTest` then sets `pValue` to the appropriate limit (`pValue = 0.001` or `0.999`).

`TestStat`
: Vector of test statistics associated with the decision vector `H`.

`CriticalValue`
: Vector of critical values associated with the decision vector `H`.

Notes

You can specify both `Lags` and `Alpha` as scalars or vectors. If you specify both as vectors, they must be the same length (that is, they must have the same number of elements). If you specify one as a scalar and the other as a vector, `dfARTest` performs a scalar expansion to enforce vectors of identical length. If `Lags` is a scalar or an empty matrix, by default, all outputs are column vectors.

All vector outputs are the same length as vector inputs `Alpha` and/or `Lags`. By default all vector outputs are column vectors. If `Lags` is a row vector, however, all vector outputs are row vectors.

This univariate unit root test is a conventional *lower tailed* test. `dfARTest` compares the test statistic with the critical value to determine whether the test is accepted or rejected. If the test statistic is *less than* the critical value, reject the null hypothesis.

See Also

`dfARDTest`, `dfTSTest`, `ppARDTest`, `ppARTest`, `ppTSTest`

References

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

Greene, W.H., *Econometric Analysis*, Prentice Hall, Fifth edition, Upper Saddle River, NJ, 2003.

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, *The GARCH of Financial Markets*, Princeton University Press, Princeton, NJ, 1997.

dfTSTest

Purpose

Run augmented Dickey-Fuller unit root test based on trend stationary AR model

Syntax

```
[H,pValue,TestStat,CriticalValue] = dfTSTest(Y,Lags,Alpha,
   TestType)
```

Description

`[H,pValue,TestStat,CriticalValue] = dfTSTest(Y,Lags,Alpha,TestType)` performs an augmented Dickey-Fuller univariate unit root test. This test assumes that the true underlying process is a unit root process with drift. As an alternative, OLS regression estimates a trend stationary (P+1)th order autoregressive (AR(P+1)) model plus additive constant.

Specifically, if:

- y_t and ε_t are the time series of observed data and model residuals, respectively, and
- $\Delta y_t = y_t - y_{t-1}$ is the first difference operator

then under the null hypothesis the true underlying process is an ARIMA(P,1,0) model with drift

$$\Delta y_t = C + y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots \zeta_p \Delta y_{t-p} + \varepsilon_t$$

Which is equivalent to an integrated AR(P+1) model.

As an alternative, the estimated OLS regression model is

$$\Delta y_t = C + \phi y_{t-1} + \zeta_1 \Delta y_{t-1} + \zeta_2 \Delta y_{t-2} + \ldots \zeta_p \Delta y_{t-p} + \varepsilon_t$$

for some constant C, AR(1) coefficient $\varphi < 1$, and trend stationary coefficient δ.

Input Arguments

Y	Time-series vector of observed data tested for a unit root. The last element contains the most recent observation. dfTSTest represents missing values as NaNs and removes them, thereby reducing the sample size.
Lags	(Optional) Scalar or vector of nonnegative integers. This parameter indicates the number of lagged changes (that is, first differences) of Y included in the OLS regression model (see P above). Lags serves as a correction for serial correlation of residuals. If Lags is empty or missing, the default is 0 (no correction for serial correlation).
Alpha	(Optional) Scalar or vector of significance levels of the test. All elements of the input argument must be 0.001 ≤ Alpha ≤ 0.999.
TestType	(Optional) Character string indicating the type of unit root test. Possible choices are t, AR, and F, indicating an OLS t test of the AR(1) coefficient, a test of the unstudentized AR(1) coefficient, and a joint OLS F test of a unit root (Φ= 1) with zero trend stationary coefficient (δ = 1), respectively. dfTSTest performs a case-insensitive check of TestType. If it is empty or missing, the default is a t test.

Output Arguments

`H`	Logical decision vector. Elements of `H = 0` indicate acceptance of the null hypothesis; elements of `H = 1` indicate rejection of the null hypothesis. Each element of `H` is associated with a particular lag of `Lags` and significance level of `Alpha`.
`pValue`	Vector of p-values (significance levels) associated with the test decision vector `H`. Each element of `pValue` represents the probability of observing a test statistic at least as extreme as that calculated from the OLS regression model when the null hypothesis is true. `dfTSTest` obtains p-values by interpolation into the appropriate table of critical values. When a p-value is outside of the range of tabulated significance levels (that is `0.001 <= Alpha <=0.999`), a warning appears. `dfTSTest` then sets `pValue` to the appropriate limit (`pValue = 0.001` or `0.999`).
`TestStat`	Vector of test statistics associated with the decision vector `H`.
`CriticalValue`	Vector of critical values associated with the decision vector `H`.

Notes

You can specify both `Lags` and `Alpha` as scalars or vectors. If you specify both as vectors, they must be the same length (that is, they must have the same number of elements). If you specify one as a scalar and the other as a vector, `dfTSTest` performs a scalar expansion to enforce vectors of identical length. If `Lags` is a scalar or an empty matrix, by default, all outputs are column vectors.

All vector outputs are the same length as vector inputs `Alpha` and/or `Lags`. By default, all vector outputs are column vectors. If `Lags` is a row vector, however, all vector outputs are row vectors.

This univariate unit root test is a conventional *single-tailed* test.`dfTSTest` compares the test statistic with the critical value to determine whether the test is accepted or rejected:

- The AR and t tests are *lower-tailed* tests. Reject the null hypothesis if the test statistic is *less than* the critical value.
- The joint F test is an *upper-tailed* test. Reject the null hypothesis if the test statistic is *greater than* the critical value.

See Also

`dfARDTest`, `dfARTest`, `ppARDTest`, `ppARTest`, `ppTSTest`

References

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

Greene, W.H., *Econometric Analysis*, Prentice Hall, Fifth edition, Upper Saddle River, NJ, 2003.

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, *The GARCH of Financial Markets*, Princeton University Press, Princeton, NJ, 1997.

Purpose

Convert finite-order ARMA models to infinite-order autoregressive (AR) models

Syntax

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

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.

Input Arguments

`AR`	*R*-element vector of autoregressive 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 unspecified, the default is `10`.

Output Arguments

`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. 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.

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

If you write this model equation as

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_R y_{t-R} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_M \varepsilon_{t-M}$$

you can specify the `garchar` input coefficient vectors, `AR` and `MA`, 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.

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

In this equation, the *j*th element of the truncated infinite-order autoregressive output vector, π_j or `InfiniteAR(j)`, is consistently the coefficient of the *j*th lag of the observed return series, y_{t-j}. See Box, Jenkins, and Reinsel [10], Section 4.2.3, pages 106-109.

Examples

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

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

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 `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`

References

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

garchcount

Purpose

Count number of GARCH estimation coefficients

Syntax

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

Description

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

Input Arguments

`Coeff`	Specification structure containing coefficient estimates and equality constraints. `Coeff` is an output of the estimation function `garchfit`.

Output Arguments

`NumParams`	Number of estimated parameters, that is, 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.

Examples

See "Akaike and Bayesian Information Criteria" on page 10-6.

See Also

`aicbic`, `garchdisp`, `garchfit`

Purpose

Display GARCH process estimation results

Syntax

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

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 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. The standard error and T-statistic columns of parameters held fixed during the estimation process display `'Fixed'`. This indicates that the parameter is an equality constraint.

Input 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 (that is, the standard errors) of the coefficients in `Coeff`. `Errors` is also an output of the estimation function `garchfit`.

Examples

1. Use `garchfit` to generate the GARCH specification structure `Coeff` and the standard errors structure `Errors`, for a return series of 1000 simulated observations based on a GARCH(1,1) model:

```
spec = garchset('C', 0, 'K', 0.0001,...
   'GARCH', 0.9, 'ARCH', 0.05,'Display', 'off');
randn('state',0);
rand('twister',0);
[e, s, y]           = garchsim(spec, 1000);
[Coeff, Errors]     = garchfit(spec, y);
```

2 Run garchdisp to display the estimation results:

```
garchdisp(Coeff, Errors)
Mean: ARMAX(0,0,0); Variance: GARCH(1,1)

Conditional Probability Distribution: Gaussian
Number of Model Parameters Estimated: 4

                                  Standard          T
  Parameter        Value           Error        Statistic
 -----------    -----------    ------------    -----------
          C     -0.0024756      0.0012919        -1.9163
          K     4.6585e-005    5.3358e-005        0.8731
   GARCH(1)     0.93927         0.041453         22.6588
    ARCH(1)     0.035442        0.015082          2.3499
```

Tip Setting 'Display' to 'off' suppresses display of the iterative optimization information produced by garchfit.

See Also

garchcount, garchfit

Purpose

Estimate univariate GARCH process parameters

Syntax

```
[Coeff,Errors,LLF,Innovations,Sigmas,Summary] = ...
  garchfit(Series)
[...] = garchfit(Spec,Series)
[...] = garchfit(Spec,Series,X)
[...] = garchfit(Spec,Series,X,...
  PreInnovations,PreSigmas,PreSeries)
garchfit(...)
```

Description

Given an observed univariate return series, `garchfit` estimates the parameters of a conditional mean specification of ARMAX form, and conditional variance specification of GARCH, EGARCH, or GJR form. The estimation process infers the innovations (that is, residuals) from the return series. It then fits the model specification to the return series by maximum likelihood.

- `[Coeff,Errors,LLF,Innovations,Sigmas,Summary] = ... garchfit(Series)` models an observed univariate return series as a constant, *C*, plus GARCH(1,1) conditionally Gaussian innovations. For models more complicated than this one, you must provide model parameters in the GARCH specification structure `Spec`.
- `[...]  = 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.
- `[...]  = garchfit(Spec,Series,X)` provides a regression component `X` for the conditional mean.
- `[...]  = garchfit(Spec,Series,X,... PreInnovations,PreSigmas,PreSeries)` uses presample observations, contained in the time-series column vectors `PreInnovations`, `PreSigmas`, and `PreSeries`, to infer the outputs `Innovations` and `Sigmas`. These vectors form the conditioning set used to initiate the inverse filtering, or inference, process. If you provide no explicit presample data, the necessary presample

observations derive from conventional time-series techniques (see "Automatically Minimizing Transient Effects" on page 4-7).

If you specify at least one, but fewer than three, sets of presample data, `garchsim` does not attempt to derive presample observations for those you omit. When specifying your own presample data, be sure to specify all data required for the given conditional mean and variance models. See "User-Specified Presample Observations" on page 6-12.

`garchfit(...)` with input arguments as shown but with no output arguments, displays the final parameter estimates and standard errors. It also produces a tiered plot of the original return series, the inferred innovations, and the corresponding conditional standard deviations.

Input Arguments

`Spec`	GARCH specification structure containing the conditional mean and variance specifications. It also contains the optimization parameters needed for the estimation. Create this structure by calling `garchset`, or use the `Coeff` output structure returned by `garchfit`.
`Series`	Time-series column vector of observations of the underlying univariate return series of interest. `Series` is the response variable representing the time series to be fitted to conditional mean and variance specifications. The last element of `Series` holds the most recent observation.

`X`	Time-series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (for example, 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 unspecified, the conditional mean has no regression component.
`PreInnovations`	Time-series column vector of presample innovations that `garchfit` uses to condition the recursive mean and variance models. This column vector can have any number of rows, provided it contains sufficient observations to initialize the mean and variance equations. That is, if `M` and `Q` are the number of lagged innovations required by the conditional mean and variance equations, respectively, then `PreInnovations` must have at least `max(M,Q)` rows. If the number of rows exceeds `max(M,Q)`, then `garchfit` uses only the last (that is, most recent) `max(M,Q)` rows.

`PreSigmas`	Time-series column vector of positive presample conditional standard deviations that `garchfit` uses to condition the recursive variance model. This vector can have any number of rows, as long as it contains sufficient observations to initialize the conditional variance equation. That is, if `P` and `Q` are the number of lagged conditional standard deviations and lagged innovations required by the conditional variance equation, respectively, then `PreSigmas` must have at least `P` rows for GARCH and GJR models, and at least `max(P,Q)` rows for EGARCH models. If the number of rows exceeds the requirement, then `garchfit` uses only the last (that is, most recent) rows.
`PreSeries`	Time-series column vector of presample observations of the return series of interest that `garchfit` uses to condition the recursive mean model. This vector can have any number of rows, provided it contains sufficient observations to initialize the conditional mean equation. Thus, if `R` is the number of lagged observations of the return series required by the conditional mean equation, then `PreSeries` must have at least `R` rows. If the number of rows exceeds `R`, then `garchfit` uses only the last (that is, most recent) `R` rows.

Output Arguments

`Coeff`	GARCH specification structure containing the estimated coefficients. `Coeff` is of the same form as the `Spec` input structure. Toolbox functions such as `garchset`, `garchget`, `garchsim`, `garchinfer`, and `garchpred` can accept either `Spec` or `Coeff` as input arguments.
`Errors`	Structure containing the estimation errors (that is, the standard errors) of the coefficients. `Errors` is of the same form as the `Spec` and `Coeff` structures. If an error occurs in the calculation of the standard errors, `garchfit` sets all fields associated with estimated coefficients to `NaN`.
`LLF`	Optimized log-likelihood objective function value associated with the parameter estimates found in `Coeff`. `garchfit` performs the optimization using the Optimization Toolbox™ `fmincon` function.
`Innovations`	Innovations (that is, residuals) time-series column vector inferred from `Series`. The size of `Innovations` is the same as the size of `Series`. If an error occurs, `garchfit` returns `Innovations` as a vector of `NaN`s.
`Sigmas`	Conditional standard deviation vector corresponding to `Innovations`. The size of `Sigmas` is the same as the size of `Series`. If an error occurs, `garchfit` returns `Sigmas` as a vector of `NaN`s.
`Summary`	Structure of summary information about the optimization process. The fields and their possible values are as follows:

`exitFlag`	Describes the exit condition: • `Positive` — Log-likelihood objective function converged to a solution. • `0` — Maximum number of function evaluations or iterations was exceeded. • `Negative` — Log-likelihood objective function did not converge to a solution.
`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`
`constraints`	One of the following strings: • `No Boundary Constraints` • `Boundary Constraints Active; Errors May Be Inaccurate`

`covMatrix`	Covariance matrix of the parameter estimates
`iterations`	Number of iterations
`functionCalls`	Number of function evaluations
`lambda`	Structure, output by `fmincon`, containing the Lagrange multipliers at the solution `x`

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. For more information, see Section 5.8 in Hamilton (References follow).

Examples

Example 1

The following example uses `garchfit` to estimate the parameters for a return series of 1000 simulated observations based on a GARCH(1,1) model. Because the `'Display'` parameter defaults to `'on'`, `garchfit` displays diagnostic and iterative information:

```
spec = garchset('C',0,'K',0.0001,'GARCH',0.9,'ARCH',0.05);
randn('state', 0); rand('twister', 0);
[e,s,y]       = garchsim(spec,1000);
[Coeff,Errors] = garchfit(spec,y);
```

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

Number of variables: 4

Functions
 Objective:                            garchllfn
 Gradient:                             finite-differencing
 Hessian:                              finite-differencing (or Quasi-Newton)
 Nonlinear constraints:                armanlc
 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:          4

Algorithm selected
   medium-scale


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

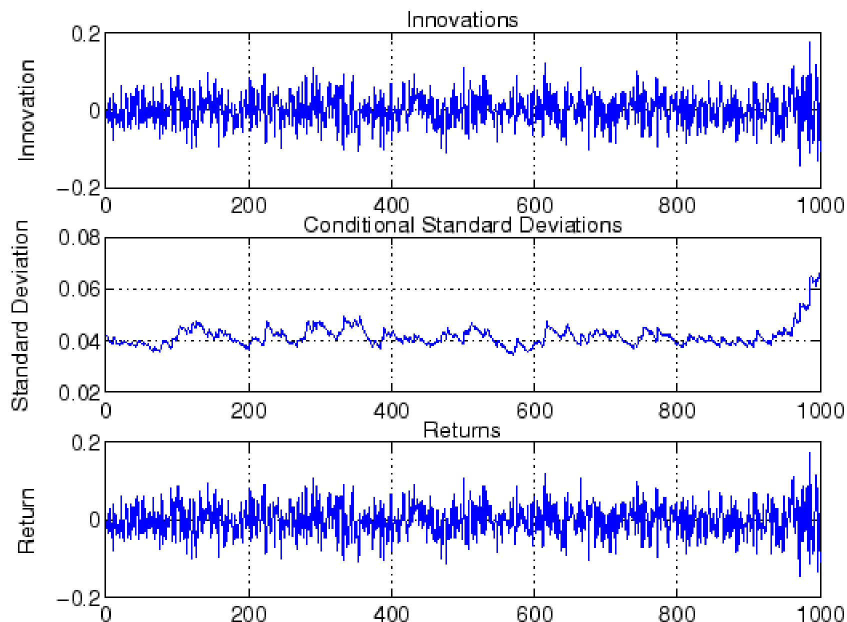
```
                              Max     Line search  Directional  First-order
 Iter F-count        f(x)   constraint   steplength   derivative  optimality Procedure
    0      5     -1762.16   -9.98e-005
    1     21     -1762.62  -9.975e-005     0.000488     1.32e+004    1.47e+004
    2     33     -1763.04  -9.897e-005      0.00781           126    2.13e+005
    3     40     -1764.68  -7.423e-005         0.25          4.18    1.28e+005
    4     53     -1764.72  -7.477e-005      0.00391          6.92    1.12e+005
    5     59     -1765.27  -4.128e-005          0.5         0.216     3.2e+003
    6     72     -1765.28   -4.75e-005      0.00391           4.2    2.99e+004
    7     82     -1765.28  -4.619e-005       0.0313         0.066    3.02e+004
    8     93     -1765.29  -4.832e-005       0.0156           0.2    2.48e+003
    9     98     -1765.29  -4.639e-005            1     -0.000171         14.6
   10    117     -1765.29  -4.639e-005    -6.1e-005    -1.03e-005         14.6
   11    124     -1765.29  -4.638e-005         0.25    -5.52e-006         2.34
Hessian modified twice
   12    132     -1765.29  -4.638e-005        0.125     7.71e-006           38
Hessian modified twice
   13    138     -1765.29  -4.638e-005          0.5     1.05e-006          126
Hessian modified twice
Optimization terminated: magnitude of directional derivative in search
direction less than 2*options.TolFun and maximum constraint violation
is less than options.TolCon.
No active inequalities.
```

Example 2

Using the same data as in Example 1, the example sets `'Display'` to `'off'` and calls `garchfit` with no output arguments. `garchfit` then displays the final parameter estimates and standard errors, then produces a tiered plot:

```
spec = garchset(spec,'Display','off');
garchfit(spec, y);
Mean: ARMAX(0, 0, 0); Variance: GARCH(1, 1)
Conditional Probability Distribution: Gaussian
Number of Model Parameters Estimated: 4

                                          Standard          T
```

```
 Parameter       Value         Error       Statistic
-----------   -----------   ------------   -----------
          C    -0.0024759    0.0012919       -1.9165
          K    4.6877e-005   5.3555e-005      0.8753
   GARCH(1)    0.93904       0.041604        22.5707
    ARCH(1)    0.035503      0.015123         2.3477

 Log Likelihood Value: 1765.29
```



See Also

`garchpred`, `garchset`, `garchsim`

`fmincon` (Optimization Toolbox function)

References

Bollerslev, T., "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return," *Review of Economics and Statistics*, Vol. 69, 1987, pp 542-547.

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

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

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, 1995.

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

Engle, R.F., D.M. Lilien, and R.P. Robins, "Estimating Time Varying Risk Premia in the Term Structure: The ARCH-M Model," *Econometrica*, Vol. 59, 1987, pp 391-407.

Glosten, L.R., R. Jagannathan, and D.E. Runkle, "On the Relation Between Expected Value and the Volatility of the Nominal Excess Return on Stocks," *The Journal of Finance*, Vol.48, 1993, pp 1779-1801.

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

Nelson, D.B., "Conditional Heteroskedasticity in Asset Returns: A New Approach," *Econometrica*, Vol. 59, 1991, pp 347-370.

Purpose

Get value of GARCH specification structure parameter

Syntax

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

Description

`ParameterValue = garchget(Spec,ParameterName)` returns the value of the specified parameter from the GARCH specification structure `Spec`.

Input Arguments

`Spec`	GARCH specification structure returned by `garchset`, or the output (`Coeff`) of the estimation function `garchfit`.
`ParameterName`	String indicating the name of the parameter whose value `garchget` extracts from `Spec`. It is sufficient to type only the leading characters that uniquely identify a parameter name. See `garchset` for a list of valid parameter names. `ParameterName` is case insensitive.

Output Arguments

`ParameterValue`	Value of the named parameter extracted from the structure `Spec`. `garchget` returns the appropriate model default value if the specified parameter is undefined in the specification structure.

Examples

1 Create a `GARCH(P=1, Q=1)` model spec:

```
Spec = garchset('P', 1, 'Q', 1)
Spec =
     Comment: 'Mean: ARMAX(0, 0, ?); Variance: GARCH(1, 1)'
```

```
 Distribution: 'Gaussian'
            C: []
VarianceModel: 'GARCH'
            P: 1
            Q: 1
            K: []
        GARCH: []
         ARCH: []
```

2 Retrieve the value of the parameter P:

```
P = garchget(Spec, 'P')       % Retrieve the order P
P =
     1
```

See Also

garchfit, garchpred, garchset, garchsim

garchinfer

Purpose

Infer GARCH innovation processes from return series

Syntax

```
[Innovations,Sigmas,LLF] = garchinfer(Spec,Series)
[...] = garchinfer(Spec,Series,X)
[...] = garchinfer(Spec,Series,X,...
  PreInnovations,PreSigmas,PreSeries)
```

Description

- `[Innovations,Sigmas,LLF] = garchinfer(Spec,Series)`, given a conditional mean specification of ARMAX form and conditional variance specification of GARCH, EGARCH, or GJR form, infers the innovations and conditional standard deviations from an observed univariate return series. Since `garchinfer` is an interface to the appropriate log-likelihood objective function, the log-likelihood value is also computed for convenience.
- `[...]  = garchinfer(Spec,Series,X)` accepts a time-series regression matrix `X` of observed explanatory data. `garchinfer` treats each column of `X` as an individual time series, and uses it as an explanatory variable in the regression component of the conditional mean.
- `[...]  = garchinfer(Spec,Series,X,...  PreInnovations,PreSigmas,PreSeries)` uses presample observations, represented by the time-series matrices or column vectors `PreInnovations`, `PreSigmas`, and `PreSeries`, to infer the outputs `Innovations` and `Sigmas`. These vectors form the conditioning set used to initiate the inverse filtering, or inference, process.

If you specify the presample data as matrices, the number of columns (realizations) of each *must* be the same as the number of columns (realizations) of the `Series` input. In this case, `garchinfer` uses the presample information of a given column to infer the residuals and standard deviations of the corresponding column of `Series`. If you specify the presample data as column vectors, `garchinfer` applies the vectors to each column of `Series`.

If you provide no explicit presample data, `garchinfer` derives the necessary presample observations using conventional time-series techniques, as described in“Automatically Minimizing Transient Effects” on page 4-7.

If you specify at least one, but fewer than three, sets of presample data, `garchsim` does not attempt to derive presample observations for those you omit. When specifying your own presample data, be sure to specify all data required by the given conditional mean and variance models. See “User-Specified Presample Observations” on page 6-12.

Input Arguments

`Spec`	GARCH specification structure that contains the conditional mean and variance specifications. It also contains the optimization parameters needed for the estimation. Create this structure by calling `garchset`, or by using the `Coeff` output structure returned by `garchfit`.
`Series`	Time-series matrix or column vector of observations of the underlying univariate return series of interest. `Series` is the response variable representing the time series fitted to conditional mean and variance specifications. Each column of `Series` in an independent realization (that is, path). The last row of `Series` holds the most recent observation of each realization.

`X`	Time-series regression matrix of explanatory variables. Typically, `X` is a regression matrix of asset returns (for example, the return series of an equity index). 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`) observations below the last `NaN` in each column of `X` must equal or exceed the number of valid observations below the last `NaN` in `Series`. If the number of valid observations in a column of `X` exceeds that of `Series`, `garchinfer` uses only the most recent. If `X = []` or is unspecified, the conditional mean has no regression component.
`PreInnovations`	Time-series matrix or column vector of presample innovations on which the recursive mean and variance models are conditioned. This array can have any number of rows, provided it contains sufficient observations to initialize the mean and variance equations. That is, if `M` and `Q` are the number of lagged innovations required by the conditional mean and variance equations, respectively, then `PreInnovations` must have at least `max(M,Q)` rows. If the number of rows exceeds `max(M,Q)`, then `garchinfer` uses only the last (that is, most recent) `max(M,Q)` rows. If `PreInnovations` is a matrix, then the number of columns must be the same as the number of columns in `Series`. If `PreInnovations` is a column vector, then `garchinfer` applies the vector to each column (realization) of `Series`.

PreSigmas	Time-series matrix or column vector of positive presample conditional standard deviations on which the recursive variance model is conditioned. This array can have any number of rows, provided it contains sufficient observations to initialize the conditional variance equation. For example, if P and Q are the number of lagged conditional standard deviations and lagged innovations required by the conditional variance equation, respectively, then PreSigmas must have: • At least P rows for GARCH and GJR models, and • At least max(P,Q) rows for EGARCH models. If the number of rows exceeds the requirement, then garchinfer uses only the last (most recent) rows. If PreSigmas is a matrix, then the number of columns must be the same as the number of columns in Series. If PreSigmas is a column vector, then garchinfer applies the vector to each column (realization) of Series.
PreSeries	Time-series matrix or column vector of presample observations of the return series of interest on which the recursive mean model is conditioned. This array can have any number of rows, provided it contains sufficient observations to initialize the conditional mean equation. Thus, if R is the number of lagged observations of the return series required by the conditional mean equation, then PreSeries must have at least R rows. If the number of rows exceeds R, then garchinfer uses only the last (most recent) R rows. If PreSeries is a matrix, then the number of columns must be the same as the number of

columns in Series. If PreSeries is a column vector, then garchinfer applies the vector to each column (realization) of Series.

Output Arguments

Innovations	Innovations time-series matrix inferred from Series. The size of Innovations is the same as the size of Series.
Sigmas	Conditional standard deviation time-series matrix corresponding to Innovations. The size of Sigmas is the same as the size of Series.
LLF	Row vector of log-likelihood objective function values for each realization of Series. The length of LLF is the same as the number of columns in Series.

Remarks

garchinfer performs essentially the same operation as garchfit, but without optimization. garchfit calls the appropriate log-likelihood objective function indirectly via the iterative numerical optimizer. garchinfer, however, allows you direct access to the same suite of log-likelihood objective functions.

These garchinfer inputs:

- Series
- PreInnovations
- PreSigmas
- PreSeries

And outputs:

- `Innovations`
- `Sigmas`

are column-oriented time-series arrays in which each column is associated with a unique realization, or random path. For `garchfit`, these same inputs and outputs cannot have multiple columns; they must all represent single realizations of a univariate time series.

For additional details about estimation and inverse filtering, see "Maximum Likelihood Estimation" on page 6-2 and "Presample Observations" on page 6-12.

Examples

- "Presample Data and Transient Effects" on page 6-24
- "Generating Presample Observations" on page 7-6
- "Estimating the Model" on page 11-3

See Also

`garchfit`, `garchpred`, `garchset`, `garchsim`

References

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

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)
```

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

Arguments

`AR`	*R*-element vector of autoregressive 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 unspecified, the default is `10`.

Output Arguments

`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. 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.

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

If you write this model equation as

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_R y_{t-R} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_M \varepsilon_{t-M}$$

you can specify the `garchma` input coefficient vectors, `AR` and `MA`, 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.

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

The *j*th element of the truncated infinite-order moving-average output vector, ψ_j or `InfiniteMA(j)`, is consistently the coefficient of the *j*th lag of the innovations process, ε_{t-j}, in this equation. See Box, Jenkins, and Reinsel [10], Section 5.2.2, pages 139-141.

Examples

Calculate 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 (that is, `10 - 1`) weights of the infinite order MA approximation.

```
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
```

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

Note 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.

See Also `garchar`, `garchpred`

References

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

garchplot

Purpose

Plot matched univariate innovations, volatility, and return series

Syntax

```
garchplot(Innovations,Sigmas,Series)
```

Description

`garchplot(Innovations,Sigmas,Series)` 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, 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; it allocates no space to the array in the tiered figure window. `garchplot` displays valid (nonempty) `Innovations`, `Sigmas`, and `Series` arrays in the top, center, and bottom plots, respectively. Because `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 can become cluttered if you try to display more than a few realizations of each input at one time.

Input Arguments

`Innovations`	Time-series column vector or matrix of innovations. As a column vector, `Innovations` represents a single realization of a univariate time series. The first element of this time series 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 `garchplot` does not display it.
`Sigmas`	Time-series column vector or matrix of conditional standard deviations. In general, `Innovations` and `Sigmas` are the same size, and form a matching pair of arrays. If `Sigmas = []`, then `garchplot` does not display it.
`Series`	Time-series column vector or matrix of asset returns. In general, `Series` is the same size as `Innovations` and `Sigmas`, and `garchplot` organizes it in the same way. If `Series = []` or is unspecified, then `garchplot` does not display it.

Examples

Example 1

Plot `Innovations`, `Sigmas`, and `Series`, assuming that they are not empty:

```
garchplot(Innovations)
garchplot(Innovations, [], Series)
garchplot([], Sigmas, Series)
garchplot(Innovations, Sigmas, Series)
garchplot(Innovations, Sigmas, [])
garchplot(Innovations, Sigmas)
```

Example 2

1. Load the default GARCH(1,1) model to model the Deutschmark/British pound foreign-exchange series `DEM2GBP`:

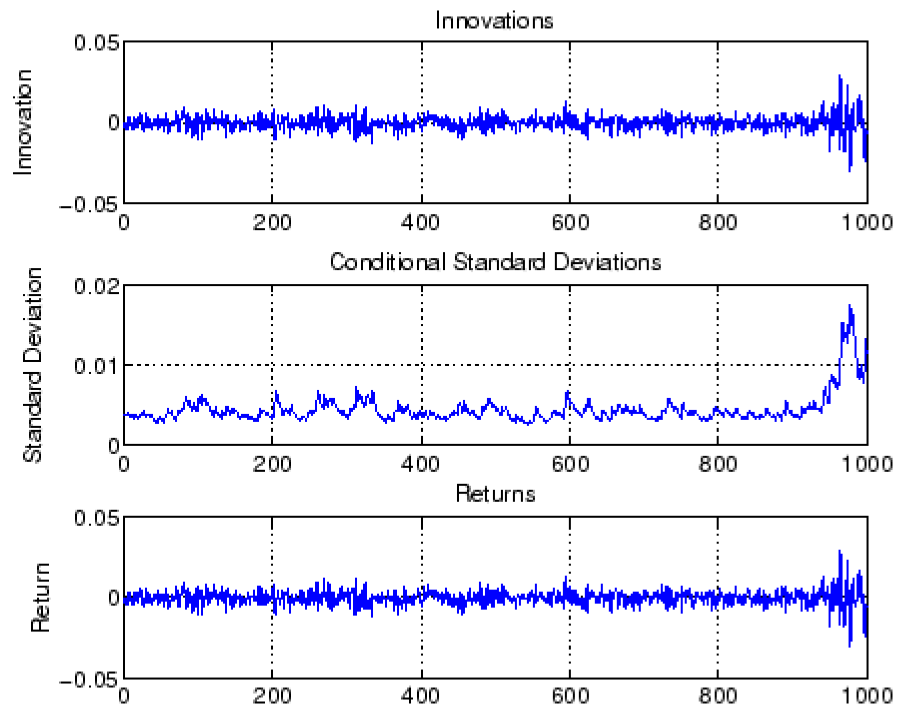
   ```
   load garchdata
   dem2gbp = price2ret(DEM2GBP);
   ```

2. Use the estimated model to generate a single path of 1000 observations for return series, innovations, and conditional standard deviation processes:

```
[coeff, errors, LLF, innovations, sigmas] = garchfit(dem2gbp);
randn('state', 0);
rand('twister', 0);
[e, s, y] = garchsim(coeff, 1000);
```

3 Plot the results:

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



See Also

`garchfit`, `garchpred`, `garchsim`

Purpose

Perform univariate GARCH process forecasting

Syntax

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

Description

`garchpred` forecasts the conditional mean of the univariate return series and the standard deviation of the innovations `NumPeriods` into the future. It uses specifications for the conditional mean and variance of an observed univariate return series as input. `garchpred` also computes volatility forecasts of asset returns over multiperiod holding intervals, and the standard errors of conditional mean forecasts. The conditional mean is of general ARMAX form and the conditional variance can be of GARCH, EGARCH, or GJR form. (See “Conditional Mean and Variance Models” on page 2-7.)

- `[SigmaForecast,MeanForecast] = ... garchpred(Spec,Series,NumPeriods)` uses the conditional mean and variance specifications defined in `Spec` to forecast the conditional mean, `MeanForecast`, of the univariate return series and the standard deviation, `SigmaForecast`, of the innovations `NumPeriods` into the future. The `NumPeriods` default is `1`.
- `[SigmaForecast,MeanForecast] = ... garchpred(Spec,Series,NumPeriods,X,XF)` includes the time-series regression matrix of observed explanatory data `X` and the time-series regression matrix of forecasted explanatory data `XF` in the calculation of `MeanForecast`.

  For `MeanForecast`, if you specify `X`, you must also specify `XF`. Typically, `X` is the same regression matrix of observed returns, if any, that you used for simulation (`garchsim`) or estimation (`garchfit`).
- `[SigmaForecast,MeanForecast,SigmaTotal,MeanRMSE] = ... garchpred(Spec,Series,Numperiods)` also computes the

volatility forecasts, SigmaTotal, of the cumulative returns for assets held for multiple periods, and the standard errors MeanRMSE associated with MeanForecast.

Input Arguments

Spec	Specification structure for the conditional mean and variance models. You can create Spec using 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 (path). The last row of Series holds the most recent observation of each realization. garchpred treats those observations as valid that are below the most recent NaN in any column. 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	Positive scalar integer representing the forecast horizon of interest. You specify NumPeriods in periods. It should be compatible with the sampling frequency of Series. If NumPeriods = [] or is unspecified, the default is 1.

`X`	Time-series regression matrix of observed explanatory data that represents the past history of the explanatory data. Typically, `X` is a regression matrix of asset returns, for example, the return series of an equity index. 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 most recent number of valid (non-`NaN`) observations in each column of `X` must equal or exceed the most recent number of valid 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` is `[]` or is unspecified, the conditional mean has no regression component.
`XF`	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 contains 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` is `[]` or is unspecified, the conditional mean forecast (`MeanForecast`) has no regression component.

Output Arguments

`SigmaForecast` Matrix of conditional standard deviations of future innovations (model residuals) on a per period basis. The standard deviations derive from the minimum mean square error (MMSE) forecasts associated with the recursive volatility model, for example, `'GARCH'`, `'GJR'`, or `'EGARCH'`, specified for the `'VarianceModel'` parameter in `Spec`. For GARCH(P,Q) and GJR(P,Q) models, `SigmaForecast` is the square root of the MMSE conditional variance forecasts. For EGARCH(P,Q) models, `SigmaForecast` is the square root of the exponential of the MMSE forecasts of the logarithm of conditional variance.

`SigmaForecast` has `NumPeriods` rows and the same number of columns as `Series`. The first row contains the standard deviation in the first period for each realization of `Series`, the second row contains the standard deviation in the second period, and so on. If you specify a forecast horizon greater than 1 (`NumPeriods > 1`), `garchpred` returns the per-period standard deviations of all intermediate horizons as well. In this case, the last row contains the standard deviation at the specified forecast horizon.

`MeanForecast` Matrix of MMSE 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 nonempty for `MeanForecast` to have a regression component. If `X` and `XF` are empty (`[]`) or is unspecified, `MeanForecast` is

	based on an ARMA model. If you specify `X` and `XF`, `MeanForecast` is based on the full ARMAX model.
`SigmaTotal`	Matrix of MMSE 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 standard deviations of the cumulative returns obtained if an asset was held for the entire `NumPeriods` forecast horizon. If you specify `X` or `XF`, `SigmaTotal` is `[]`.
`MeanRMSE`	Matrix of root mean square errors (RMSE) associated with `MeanForecast`. That is, `MeanRMSE` is the conditional standard deviation of the forecast errors (the standard error of the forecast) of the corresponding `MeanForecast` matrix. `MeanRMSE` is the same size as `MeanForecast`. `garchpred` organizes `MeanRMSE` the same way if the conditional mean is modeled as a stationary/invertible ARMA process. If you specify `X` or `XF`, `MeanRMSE` is `[]`.

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.

Notes

EGARCH(P,Q) models represent the logarithm of the conditional variance as the output of a linear filter. As such, the minimum mean

square error forecasts derived from EGARCH(P,Q) models are optimal for the logarithm of the conditional variance. They are, however, generally downward-biased forecasts of the conditional variance process itself. The output arrays `SigmaForecast`, `SigmaTotal`, and `MeanRMSE` are based upon the conditional variance forecasts. Thus, these outputs generally underestimate their true expected values for conditional variances derived from EGARCH(P,Q) models. The important exception is the one-period-ahead forecast, which is unbiased in all cases.

Examples

- "Examples: Computing Forecasts" on page 7-9
- "Forecasting" on page 11-5

See Also

`garchfit`, `garchinfer`, `garchma`, `garchset`, `garchsim`

References

Baillie, R.T., and T. Bollerslev, "Prediction in Dynamic Models with Time-Dependent Conditional Variances," *Journal of GARCH*, Vol. 52, 1992, pp 91-113.

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

Bollerslev, T., "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return," *The Review Economics and Statistics*, Vol. 69, 1987, pp 542-547.

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

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, 1995.

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

Engle, R.F., D.M. Lilien, and R.P. Robins, "Estimating Time Varying Risk Premia in the Term Structure: The ARCH-M Model," *Econometrica*, Vol. 59, 1987, pp 391-407.

Glosten, L.R., R. Jagannathan, and D.E. Runkle, “On the Relation Between Expected Value and the Volatility of the Nominal Excess Return on Stocks,” *Journal of Finance*, Vol.48, 1993, pp 1779-1801.

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

Nelson, D.B., “Conditional Heteroskedasticity in Asset Returns: A New Approach,” *Econometrica*, Vol. 59, 1991, pp 347-370.

garchset

Purpose

Create or modify GARCH specification structure

Syntax

```
Spec = garchset(param1,val1,param2,val2,...)
Spec = garchset(OldSpec, param1,val1,...)
Spec = garchset
garchset
```

Description

- `Spec = garchset(param1,val1,param2,val2,...)` creates a GARCH model specification structure `Spec` using the parameter-value pairs specified in the input argument list. Use `garchget` to retrieve the values of specification structure parameters.
- `Spec = garchset(OldSpec, param1,val1,...)` modifies an existing GARCH specification structure `OldSpec` by changing the named parameters to the specified values. `garchset` returns an error if the new parameter values would create an invalid model.
- `Spec = garchset` creates a GARCH specification structure `Spec` for the GARCH Toolbox™ default model. The conditional mean equation for this model is a simple constant plus additive noise. The conditional variance equation of the additive noise is a GARCH(1,1) model.

  You can use this `Spec` as input to `garchfit`, but you cannot use it as input to `garchinfer`, `garchpred`, or `garchsim`.
- `garchset` (with no input arguments and no output arguments) displays all parameter names (and their default values, where appropriate).

Input Arguments

`param1`, `param2`, ...	String representing a valid parameter field of the output structure `Spec`. "Parameters" on page 13-69 lists the valid parameters and describes their allowed values. A parameter name needs to include only sufficient leading characters to uniquely identify the parameter. Parameter names are case insensitive.
`val1`, `val2`, ...	Value assigned to the corresponding parameter.
`OldSpec`	Existing GARCH specification structure as generated by `garchset` or `garchfit`.

Output Arguments

`Spec`	GARCH specification structure containing the style, orders, and coefficients (if specified) of the conditional mean and variance specifications of a GARCH model. It also contains the parameters associated with the Optimization Toolbox™ `fmincon` function.

Parameters

A GARCH specification structure includes these parameters. Except as noted, `garchset` sets all parameters you do not specify to their respective defaults.

- "General Parameters" on page 13-70
- "Conditional Mean Parameters" on page 13-70
- "Conditional Variance Parameters" on page 13-71
- "Equality Constraint Parameters" on page 13-72
- "Optimization Parameters" on page 13-73

General Parameters

Parameter	Value	Description
`Comment`	String. Default is a model summary.	User-defined summary comment. An example of the default is `'Mean: ARMAX(0, 0, ?); Variance: GARCH(1, 1)'`.
`Distribution`	`'T'` or `'Gaussian'`. Default is `'Gaussian'`.	Conditional distribution of innovations.
`DoF`	Scalar. Default = `[]`.	Degrees of freedom parameter for `t` distributions (must be > 2).

Conditional Mean Parameters

If you specify coefficient vectors `AR` and `MA`, but not their corresponding model orders `R` and `M`, `garchset` infers the values of the model orders from the lengths of the coefficient vectors.

Parameter	Value	Description
`R`	Nonnegative integer scalar. Default is `0`.	Autoregressive model order of an ARMA(R,M) model.
`M`	Nonnegative integer scalar. Default is `0`.	Moving-average model order of an ARMA(R,M) model.
`C`	Scalar coefficient. Default is `[]`.	Conditional mean constant. If `C = NaN`, `garchfit` ignores `C`, effectively fixing `C = 0`, without requiring initial estimates for the remaining parameters.

Parameter	Value	Description
AR	R-element vector. Default is [].	Conditional mean autoregressive coefficients that imply a stationary polynomial.
MA	M-element vector. Default is [].	Conditional mean moving-average coefficients that imply an invertible polynomial.
Regress	Vector of coefficients. Default is [].	Conditional mean regression coefficients.

Conditional Variance Parameters

If you specify coefficient vectors GARCH and ARCH, but not their corresponding model orders P and Q, garchset infers the values of the model orders from the lengths of the coefficient vectors.

Parameter	Value	Description
VarianceModel	'GARCH', 'EGARCH', 'GJR', or 'Constant'. Default is 'GARCH'.	Conditional variance model.
P	Nonnegative integer scalar. P must be 0 if Q is 0. Default is 0.	Model order of GARCH(P,Q), EGARCH(P,Q), and GJR(P,Q) models.
Q	Nonnegative integer scalar. Default is 0.	Model order of GARCH(P,Q), EGARCH(P,Q), and GJR(P,Q) models.
K	Scalar coefficient. Default is [].	Conditional variance constant.

Parameter	Value	Description
GARCH	P-element vector. Default is [].	Coefficients related to lagged conditional variances.
ARCH	Q-element vector. Default is [].	Coefficients related to lagged innovations (residuals).
Leverage	Q-element vector. Default is [].	Leverage coefficients for asymmetric EGARCH(P,Q) and GJR(P,Q) models.

Equality Constraint Parameters

The garchfit function uses these parameters only during estimation. Use these parameters cautiously. The problem can experience difficulty converging if the fixed value is not well-suited to the data at hand.

Parameter	Value	Description
FixDoF	Logical scalar. Default is [].	Equality constraint indicator for DoF parameter.
FixC	Logical scalar. Default is [].	Equality constraint indicator for C constant.
FixAR	R-element logical vector. Default is [].	Equality constraint indicator for AR coefficients.
FixMA	M-element logical vector. Default is [].	Equality constraint indicator for MA coefficients.

Parameter	Value	Description
FixRegress	Logical vector. Default is [].	Equality constraint indicator for the REGRESS coefficients.
FixK	Logical scalar. Default is [].	Equality constraint indicator for the K constant.
FixGARCH	P-element logical vector. Default is [].	Equality constraint indicator for the GARCH coefficients.
FixARCH	Q-element logical vector. Default is [].	Equality constraint indicator for the ARCH coefficients.
FixLeverage	Q-element logical vector. Default is [].	Equality constraint indicator for Leverage coefficients.

Optimization Parameters

garchfit uses the following parameters when calling the Optimization Toolbox fmincon function during estimation.

Parameter	Value	Description
Display	'on' or 'off'. Default is 'on'.	Display iterative optimization information.
MaxFunEvals	Positive integer. Default = (100*number of estimated parameters).	Maximum number of objective function evaluations allowed.
MaxIter	Positive integer. Default is 400.	Maximum number of iterations allowed.

Parameter	Value	Description
TolCon	Positive scalar. Default is 1e-007.	Termination tolerance on the constraint violation.
TolFun	Positive scalar. Default is 1e-006.	Termination tolerance on the objective function value.
TolX	Positive scalar. Default is 1e-006.	Termination tolerance on parameter estimates.

Examples

1 Create a GARCH(1,1) model:

```
spec = garchset('P', 1, 'Q', 1)
spec =
          Comment: 'Mean: ARMAX(0, 0, ?);
         Variance: GARCH(1, 1)'
     Distribution: 'Gaussian'
                C: []
    VarianceModel: 'GARCH'
                P: 1
                Q: 1
                K: []
            GARCH: []
             ARCH: []
```

2 Change the model to a GARCH(1,2) model:

```
spec = garchset(spec, 'Q', 2)
spec =
          Comment: 'Mean: ARMAX(0, 0, ?); Variance: GARCH(1, 2)'
     Distribution: 'Gaussian'
                C: []
    VarianceModel: 'GARCH'
                P: 1
```

```
        Q: 2
        K: []
    GARCH: []
     ARCH: []
```

In each case, `garchset` displays the relevant fields in the specification structure.

Tip Use `garchget` to retrieve the values of individual fields.

See Also

`garchfit`, `garchget`, `garchpred`, `garchsim`

`fmincon` (Optimization Toolbox function)

garchsim

Purpose

Perform univariate GARCH process simulation

Syntax

```
[Innovations,Sigmas,Series] = garchsim(Spec)
[...] = garchsim(Spec,NumSamples,NumPaths)
[...] = garchsim(Spec,NumSamples,NumPaths, State)
[...] = garchsim(Spec,NumSamples,NumPaths,State,X)
[...] = garchsim(Spec,NumSamples,NumPaths,State,X,Tolerance)
[...] = garchsim(Spec,NumSamples,NumPaths,State,X,Tolerance, ...)
  PreInnovations,PreSigmas,PreSeries)
```

Description

- `[Innovations,Sigmas,Series] = garchsim(Spec)`, given specifications for the conditional mean and variance of a univariate time series, simulates a sample path with 100 observations for the return series, innovations, and conditional standard deviation processes. The conditional mean can be of general ARMA form and the conditional variance of general GARCH, EGARCH, or GJR form.
- `[...]  = garchsim(Spec,NumSamples,NumPaths)` simulates `NumPaths` sample paths. Each path is sampled at `NumSamples` observations.
- `[...]  = garchsim(Spec,NumSamples,NumPaths, State)` specifies the State time-series matrix of the standardized (zero mean, unit variance), independent, identically distributed random noise process.
- `[...]  = garchsim(Spec,NumSamples,NumPaths,State,X)` accepts a time-series regression matrix `X` of observed explanatory data. `garchsim` treats each column of `X` as an individual time series, and uses it as an explanatory variable in the regression component of the conditional mean.
- `[...]  = garchsim(Spec,NumSamples,NumPaths,State,X,Tolerance)` accepts a scalar transient response tolerance, such that 0 < `Tolerance` ≤ 1. `garchsim` estimates the number of observations needed for the magnitude of the impulse response, which begins at 1, to decay below the `Tolerance` value. The number of observations associated with the transient decay period is subject to a maximum of 10,000 to prevent out-of-memory conditions. When you specify

presample observations (`PreInnovations`, `PreSigmas`, and `PreSeries`), `garchsim` ignores the value of `Tolerance`.

Use `Tolerance` to manage the conflict between transient minimization and memory usage. Smaller `Tolerance` values generate output processes that more closely approximate true steady-state behavior, but require more memory for the additional filtering required. Conversely, larger `Tolerance` values require less memory, but produce outputs in which transients tend to persist.

If you do not explicitly specify presample data (see below), the impulse response estimates are based on the magnitude of the largest eigenvalue of the autoregressive polynomial.

- `[...] = garchsim(Spec,NumSamples,NumPaths,State,X,Tolerance, ...) PreInnovations,PreSigmas,PreSeries)` uses presample observations, contained in the time-series matrices or column vectors `PreInnovations`, `PreSigmas`, and `PreSeries`, to simulate the outputs `Innovations`, `Sigmas`, and `Series`, respectively. When specified, `garchsim` uses these presample arrays to initiate the filtering process, and thus form the conditioning set upon which the simulated realizations are based.

If you specify the presample data as matrices, they *must* have `NumPaths` columns. `garchsim` uses the presample information from a given column to initiate the simulation of the corresponding column of the `Innovations`, `Sigmas`, and `Series` outputs. If you specify the presample data as column vectors, `garchsim` applies the vectors to each column of the corresponding `Innovations`, `Sigmas`, and `Series` outputs.

If you provide no explicit presample data, `garchsim` automatically derives the necessary presample observations, as described in "Automatically Minimizing Transient Effects" on page 4-7.

`PreInnovations` and `PreSigmas` are usually companion inputs. Although both are optional, when specified, they are typically entered together. A notable exception would be a GARCH(0,Q) (that is, an ARCH(Q)) model in which the conditional variance equation does not

require lagged conditional variance forecasts. Similarly, `PreSeries` is only necessary when you want to simulate the output return `Series`, and when the conditional mean equation has an autoregressive component.

If the conditional mean or the conditional variance equation (“Conditional Mean and Variance Models” on page 2-7) is not recursive, then certain presample information is not needed to jump-start the models. However, specifying redundant presample information is *not* an error, and `garchsim` ignores presample observations you specify for models that require no such information.

Input 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 the conditional variance of general GARCH form.
`NumSamples`	(optional) Positive integer indicating the number of observations `garchsim` generates for each path of the `Innovations`, `Sigmas`, and `Series` outputs. If `NumSamples = []` or is unspecified, the default is `100`.
`NumPaths`	(Optional) Positive integer indicating the number of sample paths (realizations) `garchsim` generates for the `Innovations`, `Sigmas`, and `Series` outputs. If `NumPaths = []` or is unspecified, the default is `1`; that is, `Innovations`, `Sigmas` and `Series` are column vectors.

`PreInnovations`	Time-series matrix or column vector of presample innovations on which the recursive mean and variance models are conditioned. This array can have any number of rows, provided it contains sufficient observations to initialize the mean and variance equations. That is, if `M` and `Q` are the number of lagged innovations required by the conditional mean and variance equations, respectively, then `PreInnovations` must have at least `max(M,Q)` rows. If the number of rows exceeds `max(M,Q)`, then `garchsim` uses only the last (most recent) `max(M,Q)` rows. If `PreInnovations` is a matrix, then it must have `NumPaths` columns.
`PreSigmas`	Time-series matrix or column vector of positive presample conditional standard deviations on which the recursive variance model is conditioned. This array can have any number of rows, provided it contains sufficient observations to initialize the conditional variance equation. That is, if `P` and `Q` are the number of lagged conditional standard deviations and lagged innovations required by the conditional variance equation, respectively, then `PreSigmas` must have at least `P` rows for GARCH and GJR models, and at least `max(P,Q)` rows for EGARCH models. If the number of rows exceeds the requirement, then `garchsim` uses only the last (most recent) rows. If `PreSigmas` is a matrix, then it must have `NumPaths` columns.

`PreSeries`	Time-series matrix or column vector of presample observations of the return series of interest on which the recursive mean model is conditioned. This array can have any number of rows, provided it contains sufficient observations to initialize the conditional mean equation. Thus, if `R` is the number of lagged observations of the return series required by the conditional mean equation, then `PreSeries` must have at least `R` rows. If the number of rows exceeds `R`, then `garchsim` uses only the last (most recent) `R` rows. If `PreSeries` is a matrix, then it must have `NumPaths` columns.
`State`	Time series matrix of standardized (mean zero, unit variance), independent, identically distributed disturbances that drive the output `Innovations` time series process. When specified, `State` must have `NumPaths` columns and at least `NumSamples` rows. The first row contains the oldest observation and the last row the most recent. `garchsim` automatically generates additional presample observations required to minimize transients, if any, based on the distribution found in the input specification structure `Spec`. `garchsim` then prepends these to the input `State` time series matrix. If `State` has more observations (rows) than necessary, then `garchsim` uses only the most recent observations. If `State` is empty or missing, `garchsim` automatically generates a noise process of appropriate size and distribution.

`Tolerance`	Scalar transient response tolerance, such that `0 < Tolerance ≤ 1`. `garchsim` ignores this tolerance parameter if you specify presample conditioning information (`PreInnovations`, `PreSigmas`, and `PreSeries`). If `Tolerance` is empty or missing, the default is `0.01` (1%).
`X`	Time-series regression matrix of observed explanatory data. Typically, `X` is a matrix of asset returns (for example, 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 unspecified, the conditional mean has no regression component. If specified, then at least the most recent `NumSamples` observations of each return series must be valid (non-`NaN`). When the number of valid observations in each series exceeds `NumSamples`, `garchsim` uses only the most recent `NumSamples` observations of `X`.

Output Arguments

`Innovations`	`NumSamples` by `NumPaths` matrix of innovations, representing a mean zero, discrete-time stochastic process. The `Innovations` time series follows the conditional variance specification defined in `Spec`. Rows are sequential observations, columns are realizations.
`Sigmas`	`NumSamples` by `NumPaths` matrix of conditional standard deviations of the corresponding `Innovations` matrix. `Innovations` and `Sigmas` are the same size. Rows are sequential observations. Columns are 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 observations. Columns are realizations.

Examples

State as a Standardized Noise Matrix

1 When `State` is specified, it represents a user-defined time-series matrix of standardized (mean zero, unit variance), i.i.d. disturbances `{z(t)}` that drive the output time-series processes `{e(t)}`, `{s(t)}`, and `{y(t)}`.

For example, if you run `garchsim` once, then standardize the simulated residuals and pass them into `garchsim` as the i.i.d. state noise input for a second run, the standardized residuals from both runs will be identical. This verifies that the specified input state noise matrix is indeed the "in-sample" i.i.d. noise process `{z(t)}` for both:

```
spec   = garchset('C', 0.0001, 'K', 0.00005, ...
'GARCH', 0.8, 'ARCH', 0.1);
```

```
[e1, s1, y1] = garchsim(spec, 100, 1);
z1 = e1./s1; % Standardize residuals
[e2, s2, y2] = garchsim(spec, 100, 1, z1);
z2 = e2./s2; % Standardize residuals
```

In this case, `z1` = `z2`. However, although the “in-sample” standardized noise processes are identical, in the absence of presample data the simulated output processes `{e(t)}`, `{s(t)}`, and `{y(t)}` differ. This is because additional standardized noise observations necessary to minimize transients must be simulated from the distribution, `'Gaussian'` or `'T'`, found in the specification structure.

2 Specify all required presample data and repeat the experiment:

```
[e3,s3,y3] = garchsim(spec,100,1,[],[],[],...
0.02,0.06);
z3 = e3./s3; % Standardize residuals
[e4,s4,y4]   = garchsim(spec,100,1,z3,...
[],[],0.02,0.06);
z4 = e4./s4; % Standardize residuals
```

In this case, `e3` = `e4`, `s3` = `s4`, `y3` = `y4` and `z3` = `z4`.

More Examples

- “Simulating Single and Multiple Paths” on page 4-2
- “Fitting a Model to a Simulated Return Series” on page 8-3
- “Forecasting Using Monte Carlo Simulation” on page 11-7
- “Market Risk Using GARCH, Bootstrapping and Filtered Historical Simulation”
- “Market Risk Using GARCH, Extreme Value Theory, and Copulas”

See Also

`garchfit`, `garchget`, `garchpred`, `garchset`

`rand`, `randn` (MATLAB® function)

References

Bollerslev, T., “A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return,” *Review of Economics and Statistics*, Vol. 69, 1987, pp 542-547.

Bollerslev, T., “Generalized Autoregressive Conditional Heteroskedasticity,” *Journal of GARCH*, Vol. 31, 1986, pp 307-327.

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

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, 1995.

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

Engle, R.F., D.M. Lilien, and R.P. Robins, “Estimating Time Varying Risk Premia in the Term Structure: The ARCH-M Model,” *Econometrica*, Vol. 59,1987, pp 391-407.

Glosten, L.R., R. Jagannathan, and D.E. Runkle, “On the Relation Between Expected Value and the Volatility of the Nominal Excess Return on Stocks,” *The Journal of Finance*, Vol.48, 1993, pp 1779-1801.

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

Nelson, D.B., “Conditional Heteroskedasticity in Asset Returns: A New Approach,” *Econometrica*, Vol. 59, 1991, pp 347-370.

Purpose

Run Hodrick-Prescott filter

Syntax

```
hpfilter(S)
hpfilter(S,smoothing)
T = hpfilter(...)
[T,C] = hpfilter(...)
```

Description

- `hpfilter(S)` uses a Hodrick-Prescott filter and a default smoothing parameter of 1600 to separate the columns of `S` into trend and cyclical components. `S` is an *m*-by-*n* matrix with *m* samples from *n* time series. A plot displays each time series together with its trend (the time series with the cyclic component removed).
- `hpfilter(S,smoothing)` applies the smoothing parameter `smoothing` to the columns of `S`. If `smoothing` is a scalar, `hpfilter` applies it to all columns. If `S` has *n* columns and `smoothing` is a conformable vector (*n*-by-1 or 1-by-*n*), `hpfilter` applies the vector components of `smoothing` to the corresponding columns of `S`.

  If the smoothing parameter is `0`, no smoothing takes place. As the smoothing parameter increases in value, the smoothed series becomes more linear. A smoothing parameter of `Inf` produces a linear trend component.

  Appropriate values of the smoothing parameter depend upon the periodicity of the data. The following reference suggests the following values:

  - Yearly — 100
  - Quarterly — 1600
  - Monthly — 14400
- `T = hpfilter(...)` returns the trend components of the columns of `S` in `T`, without plotting.
- `[T,C] = hpfilter(...)` returns the cyclical components of the columns of `S` in `C`, without plotting.

Remarks

The Hodrick-Prescott filter separates a time series y_t into a trend component T_t and a cyclical component C_t such that $y_t = T_t + C_t$. It is equivalent to a cubic spline smoother, with the smoothed portion in T_t.

The objective function for the filter has the form

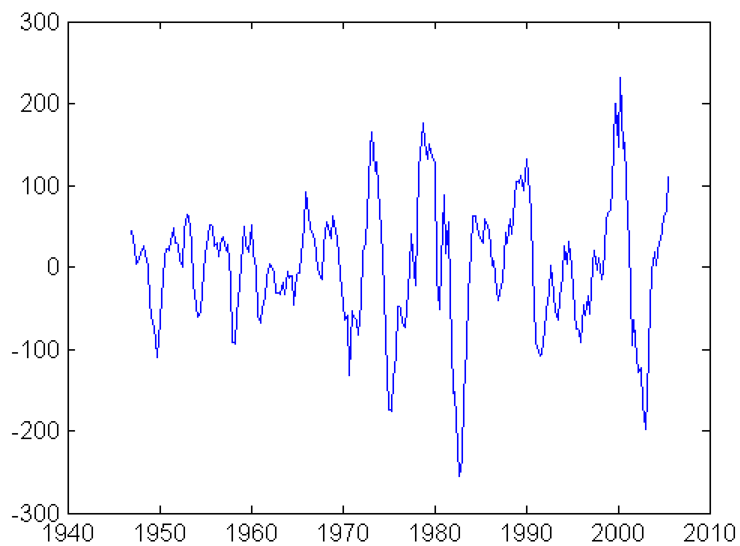
$$\sum_{t=1}^{m} C_t^2 + \lambda \sum_{t=2}^{m-1} ((T_{t+1} - T_t) - (T_t - T_{t-1}))^2$$

where m is the number of samples and λ is the smoothing parameter. The programming problem is to minimize the objective over all T_1, ... , T_m. The first sum minimizes the difference between the time series and its trend component (which is its cyclical component). The second sum minimizes the second-order difference of the trend component (which is analogous to minimization of the second derivative of the trend component).

Example

Plot the cyclical component of the U.S. post-WWII seasonally-adjusted real GNP:

```
load gnp
gnpdate = gnp(:,1);
realgnp = gnp(:,3);
[T,C] = hpfilter(realgnp);
Warning: Missing or empty Smoothing parameter set to 1600.
plot(gnpdate,C)
```



Reference

[1] Robert J. Hodrick and Edward C. Prescott, “Postwar U.S. Business Cycles: An Empirical Investigation,” *Journal of Money, Credit, and Banking*, Vol. 29, No. 1, February 1997, pp. 1-16.

Purpose

Create lagged time-series matrix

Syntax

```
XLAG = lagmatrix(X,Lags)
```

Description

`XLAG = lagmatrix(X,Lags)` creates a lagged (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.

Input Arguments

`X`	Time-series of explanatory data. `X` can be a column vector or a matrix. As a column vector, `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. 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.

Output Arguments

`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. 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.

Examples

Example 1

1 Create a bivariate time-series matrix X with five observations each:

```
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
```

2 Create a lagged matrix XLAG, composed of X and the first two lags of X:

```
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
```

The result, XLAG, is a 5-by-6 matrix.

Example 2

See "Fitting a Regression Model to the Same Return Series" on page 8-5.

See Also

`filter`, `isnan`, and `nan` (MATLAB® functions)

Purpose

Run Ljung-Box Q-statistic lack-of-fit hypothesis test

Syntax

```
[H,pValue,Qstat,CriticalValue] = ...
  lbqtest(Series,Lags,Alpha,DoF)
```

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 = the 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.

Input 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` 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 unspecified, the default is `Lags = min([20, length(Series)-1])`.
`Alpha`	Significance levels. `Alpha` can be a scalar applied to all lags, or a vector the same length as `Lags`. If `Alpha = []` or is unspecified, the default is `0.05`. For all elements, α, of `Alpha`, $0 < \alpha < 1$.
`DoF`	Degrees 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 unspecified, the elements of `Lags` serve as the default degrees of freedom for the chi-square distribution.

Output Arguments

`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`.

Examples

Example 1

1 Create a vector of 100 Gaussian random numbers:

```
randn('state', 100)        % Start from a known state.
Series = randn(100, 1);    % 100 Gaussian deviates ~ N(0, 1)
```

2 Compute the Q-statistic for autocorrelation lags 20 and 25 at the 10 percent significance level:

```
[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
```

Example 2

See "Pre-Estimation Analysis" on page 2-16.

See Also

`archtest`, `autocorr`

References

Box, G.E.P., G.M. Jenkins, and 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

Run Likelihood ratio hypothesis test

Syntax

```
[H,pValue,Ratio,CriticalValue] = ...
  lratiotest(BaseLLF,NullLLF,DoF,Alpha)
```

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. 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`. The asymptotic distribution of the test statistic is chi-square distributed with degrees of freedom equal to the number of restrictions.

Input 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 that you obtained the `NullLLF` values using `garchfit` or `garchinfer`.

DoF	Degrees of freedom (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	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 unspecified, the default is 0.05. For all elements, α, of Alpha, $0 < \alpha < 1$.

Output Arguments

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 $\text{Ratio} = 2(\text{BaseLLF} - \text{NullLLF})$.
CriticalValue	Vector of critical values of the chi-square distribution. CriticalValue is the same size as NullLLF.

Examples

See “Likelihood Ratio Tests” on page 10-3 and “Equality Constraints and Parameter Significance” on page 10-9.

See Also

`garchfit`, `garchinfer`

References

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

Purpose

Plot or return computed sample partial autocorrelation function

Syntax

```
parcorr(Series,nLags,R,nSTDs)
[PartialACF,Lags,Bounds] = ...
  parcorr(Series,nLags,R,nSTDs)
```

Description

- `parcorr(Series,nLags,R,nSTDs)` computes and plots the sample partial autocorrelation 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.

Input Arguments

`Series`	Vector of observations of a univariate time series for which `parcorr` returns or plots the sample partial autocorrelation function (partial ACF). The last element of `Series` contains the most recent observation of the stochastic sequence.
`nLags`	Positive scalar integer indicating the number of lags of the partial ACF to compute. If `nLags = []` or is unspecified, `parcorr` computes the partial ACF sequence at lags 0, 1, 2, ...,T , where T `= min([20,length(Series)-1])`.

`R`	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 greater than `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 $\frac{1}{\sqrt{N}}$ for lags greater than `R`. If `R = []` or is unspecified, the default is `0`. The value of `R` must be less than `nLags`.
`nSTDs`	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 (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 $\pm(\frac{nSTDs}{\sqrt{N}})$. If `nSTDs = []` or is unspecified, the default is `2` (approximate 95 percent confidence interval).

Output Arguments

`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, that is, `PartialACF(1) = 1 =` OLS regression

	coefficient of `Series` regressed upon itself. `parcorr` includes this element as a reference.
`Lags`	Vector of lags, of length `nLags` + 1. The elements correspond to the elements of `PartialACF`.
`Bounds`	Two-element vector indicating the approximate upper and lower confidence bounds, assuming that `Series` is an AR(R) process. `Bounds` is approximate for lags greater than `R` only.

Examples

Example 1

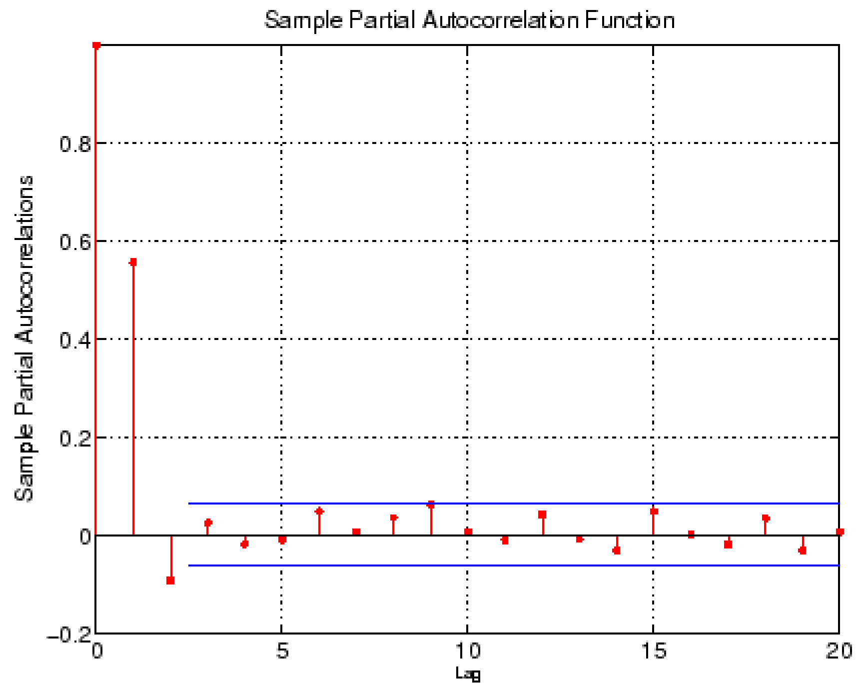
1 Create a stationary AR(2) process from a sequence of 1000 Gaussian deviates:

```
randn('state', 0);
x = randn(1000, 1);
y = filter(1, [1 -0.6 0.08], x);
[PartialACF, Lags, Bounds] = parcorr(y, [], 2);
[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
```

2 Visually assess whether the partial ACF is zero for lags greater than `2`:

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



Example 2

See "Pre-Estimation Analysis" on page 2-16.

See Also

`autocorr`, `crosscorr`

`filter` (MATLAB® function)

References

Box, G.E.P., G.M. Jenkins, and 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

Run Phillips-Perron unit root test based on AR(1) model with drift

Syntax

```
[H,pValue,TestStat,CriticalValue] = ...
  ppARDTest(Y,Lags,Alpha,TestType)
```

Description

`[H,pValue,TestStat,CriticalValue] = ...`
  `ppARDTest(Y,Lags,Alpha,TestType)` performs a Phillips-Perron univariate unit root test. This test assumes that the true underlying process is a zero drift unit root process. As an alternative, OLS regression estimates a first-order autoregressive (AR(1)) model plus additive constant.

Specifically, consider y_t and ε_t to be the time series of observed data and model residuals, respectively. In this case, under the null hypothesis,`ppARDTest` assumes the true underlying process is

$$y_t = y_{t-1} + \varepsilon_t$$

As an alternative, the estimated OLS regression model is

$$y_t = C + \phi y_{t-1} + \varepsilon_t$$

for some constant C and AR(1) coefficient $\varphi < 1$.

Input Arguments

`Y`	Time-series vector of observed data tested for a unit root. The last element contains the most recent observation. `ppARDTest` represents missing values as `NaN`s and removes them, thereby reducing the sample size.
`Lags`	(Optional) Scalar or vector of nonnegative integers. This parameter indicates the number of autocovariance lags included in the Newey-West estimation of the asymptotic variance of the sample mean of the residuals. `Lags` serves as a correction for serial

	correlation of residuals. If empty or missing, the default is 0 (no correction for serial correlation).
Alpha	(Optional) Scalar or vector of significance levels of the test. All elements of the input argument must be 0.001 ≤ Alpha ≤ 0.999.
TestType	(Optional) Character string indicating the type of unit root test. Possible choices are t and AR, indicating an OLS t test of the AR coefficient and a test of the unstudentized AR coefficient, respectively. ppARDTest performs a case-insensitive check of TestType. If it is empty or missing, the default is a t test.

Output Arguments

H	Logical decision vector. Elements of H = 0 indicate acceptance of the null hypothesis; elements of H = 1 indicate rejection of the null hypothesis. Each element of H is associated with a particular lag of Lags and significance level of Alpha.
pValue	Vector of p-values (significance levels) associated with the test decision vector H. Each element of pValue represents the probability of observing a test statistic at least as extreme as that calculated from the OLS regression model when the null hypothesis is true. ppARDTest obtains p-values by interpolation into the appropriate table of critical values. When a p-value is outside of the range of tabulated significance levels (0.001 <= Alpha <=0.999), a warning appears. ppARDTest then sets pValue to the appropriate limit (pValue = 0.001 or 0.999).

`TestStat`	Vector of test statistics associated with the decision vector `H`.
`CriticalValue`	Vector of critical values associated with the decision vector `H`.

Notes

You can specify both `Lags` and `Alpha` as scalars or vectors. If you specify both as vectors, they must be the same length (that is, they must have the same number of elements). If you specify one as a scalar and the other as a vector, `ppARDTest` performs a scalar expansion to enforce vectors of identical length. If `Lags` is a scalar or an empty matrix, by default, all outputs are column vectors.

All vector outputs are the same length as vector inputs `Alpha` and/or `Lags`. By default all vector outputs are column vectors. If `Lags` is a row vector, however, all vector outputs are row vectors.

This univariate unit root test is a conventional *lower-tailed* test. `ppARDTest` compares the test statistic with the critical value to determine whether the test is accepted or rejected. If the test statistic is *less than* the critical value, then reject the null hypothesis.

See Also

`dfARDTest`, `dfARTest`, `dfTSTest`, `ppARTest`, `ppTSTest`

References

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

Greene, W.H., *Econometric Analysis*, Prentice Hall, Fifth edition, Upper Saddle River, NJ, 2003.

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, *The GARCH of Financial Markets*, Princeton University Press, Princeton, NJ, 1997.

Purpose

Run Phillips-Perron unit root test based on zero drift AR(1) model

Syntax

```
[H,pValue,TestStat,CriticalValue] = ...
  ppARTest(Y,Lags,Alpha,TestType)
```

Description

`[H,pValue,TestStat,CriticalValue] = ...`
`ppARTest(Y,Lags,Alpha,TestType)` performs a Phillips-Perron univariate unit root test. This test assumes that the true underlying process is a zero drift unit root process. As an alternative, OLS regression estimates a zero drift first-order autoregressive (AR(1)) model.

Specifically, consider y_t and ε_t to be the time series of observed data and model residuals, respectively. Then under the null hypothesis, `ppARTest` assumes that the true underlying process is

$$y_t = y_{t-1} + \varepsilon_t$$

As an alternative, the estimated OLS regression model is

$$y_t = \phi y_{t-1} + \varepsilon_t$$

for some AR(1) coefficient $\varphi < 1$.

Input Arguments

`Y`	Time-series vector of observed data tested for a unit root. The last element contains the most recent observation. `ppARTest` represents missing values as `NaN`s and removes them, thereby reducing the sample size.
`Lags`	(Optional) Scalar or vector of nonnegative integers. This parameter indicates the number of autocovariance lags included in the Newey-West estimation of the asymptotic variance of the sample mean of the residuals. `Lags` serves as a correction for serial

correlation of residuals. If empty or missing, the default is `0` (no correction for serial correlation).

`Alpha` (Optional) Scalar or vector of significance levels of the test. All elements of the input argument must be 0.001 ≤ `Alpha` ≤ 0.999.

`TestType` (Optional) Character string indicating the type of unit root test. Possible choices are `t` and `AR`, indicating an OLS `t` test of the AR coefficient and a test of the unstudentized AR coefficient, respectively. `ppARTest` performs a case-insensitive check of `TestType`. If it is empty or missing, the default is a `t` test. .

Output Arguments

`H` Logical decision vector. Elements of `H = 0` indicate acceptance of the null hypothesis; elements of `H = 1` indicate rejection of the null hypothesis. Each element of `H` is associated with a particular lag of `Lags` and significance level of `Alpha`.

`pValue` Vector of p-values (significance levels) associated with the test decision vector `H`. Each element of `pValue` represents the probability of observing a test statistic at least as extreme as that calculated from the OLS regression model when the null hypothesis is true. `ppARTest` obtains p-values by interpolation into the appropriate table of critical values.

When a p-value is outside of the range of tabulated significance levels (`0.001 <= Alpha <=0.999`), a warning appears. `ppARTest` then sets `pValue` to the appropriate limit (`pValue = 0.001` or `0.999`).

TestStat	Vector of test statistics associated with the decision vector H.
CriticalValue	Vector of critical values associated with the decision vector H.

Notes

You can specify both Lags and Alpha as scalars or vectors. If you specify both as vectors, they must be the same length (that is, they must have the same number of elements). If you specify one as a scalar and the other as a vector, ppARTest performs a scalar expansion to enforce vectors of identical length. If Lags is a scalar or an empty matrix, by default, all outputs are column vectors.

All vector outputs are the same length as vector inputs Alpha and/or Lags. By default, all vector outputs are column vectors. If Lags is a row vector, however, all vector outputs are row vectors.

This univariate unit root test is a conventional *lower-tailed* test. ppARTest compares the test statistic with the critical value to determine whether the test is accepted or rejected. If the test statistic is *less than* the critical value, reject the null hypothesis.

See Also

dfARDTest, dfARTest, dfTSTest, ppARDTest, ppTSTest

References

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

Greene, W.H., *Econometric Analysis*, Prentice Hall, Fifth edition, Upper Saddle River, NJ, 2003.

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, *The GARCH of Financial Markets*, Princeton University Press, Princeton, NJ, 1997.

ppTSTest

Purpose

Run Phillips-Perron unit root test based on trend stationary AR(1) model

Syntax

```
[H,pValue,TestStat,CriticalValue] = ...
  ppTSTest(Y,Lags,Alpha,TestType)
```

Description

`[H,pValue,TestStat,CriticalValue] = ...`
`ppTSTest(Y,Lags,Alpha,TestType)` performs a Phillips-Perron univariate unit root test. This test assumes that the true underlying process is a unit root process with drift. As an alternative, OLS regression estimates a trend stationary first-order autoregressive (AR(1)) model plus additive constant.

Specifically, consider y_t and ε_t to be the time series of observed data and model residuals, respectively. Then under the null hypothesis, `ppTSTest` assumes that the true underlying process is

$$y_t = C + y_{t-1} + \varepsilon_t$$

for an arbitrary constant C. As an alternative, the estimated OLS regression model is

$$y_t = C + \phi y_{t-1} + \delta t + \varepsilon_t$$

for some constant C, AR(1) coefficient $\varphi < 1$, and trend stationary coefficient δ.

Input Arguments

`Y`	Time-series vector of observed data tested for a unit root. The last element contains the most recent observation. `ppTSTest` represents missing values as `NaN`s and removes them, thereby reducing the sample size.
`Lags`	(Optional) Scalar or vector of nonnegative integers. This parameter indicates the number of autocovariance lags included in the Newey-West estimation of the asymptotic variance of the sample mean of the residuals. `Lags` serves as a correction for serial correlation of residuals. If empty or missing, the default is `0` (no correction for serial correlation).
`Alpha`	(Optional) Scalar or vector of significance levels of the test. All elements of the input argument must be 0.001 ≤ `Alpha` ≤ 0.999.
`TestType`	(Optional) Character string indicating the type of unit root test. Possible choices are `t` and `AR`, indicating an OLS `t` test of the AR coefficient and a test of the unstudentized AR coefficient, respectively. `ppTSTest` performs a case-insensitive check of `TestType`. If it is empty or missing, the default is a `t` test.

Output Arguments

`H`	Logical decision vector. Elements of `H = 0` indicate acceptance of the null hypothesis; elements of `H = 1` indicate rejection of the null hypothesis. Each element of `H` is associated with a particular lag of `Lags` and significance level of `Alpha`.
`pValue`	Vector of p-values (significance levels) associated with the test decision vector `H`. Each element of `pValue` represents the probability of observing a test statistic at least as extreme as that calculated from the OLS regression model when the null hypothesis is true. `ppTSTest` obtains p-values by interpolation into the appropriate table of critical values. When a p-value is outside of the range of tabulated significance levels (`0.001 <= Alpha <=0.999`), a warning appears. `ppTSTest` then sets `pValue` to the appropriate limit (`pValue` = `0.001` or `0.999`).
`TestStat`	Vector of test statistics associated with the decision vector `H`.
`CriticalValue`	Vector of critical values associated with the decision vector `H`.

Notes

You can specify `Lags` and `Alpha` as scalars or vectors. If you specify both as vectors, they must be the same length (that is, they must have the same number of elements). If one is specified as a scalar and the other as a vector, `ppTSTest` performs a scalar expansion to enforce identical-length vectors. If `Lags` is a scalar or an empty matrix, all outputs are column vectors by default.

All vector outputs are the same length as vector inputs `Alpha` and/or `Lags`. By default all vector outputs are column vectors. If `Lags` is a row vector, however, all vector outputs are row vectors.

This univariate unit root test is a conventional *lower-tailed* test. `ppTSTest` compares the test statistic with the critical value to determine whether the test is accepted or rejected. If the test statistic is *less than* the critical value, reject the null hypothesis.

See Also

`dfARDTest`, `dfARTest`, `dfTSTest`, `ppARDTest`, `ppARTest`

References

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

Greene, W.H., *Econometric Analysis*, Prentice Hall, Fifth edition, Upper Saddle River, NJ, 2003.

Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, *The GARCH of Financial Markets*, Princeton University Press, Princeton, NJ, 1997.

Purpose

Convert price series to return series

Syntax

```
[RetSeries,RetIntervals] = ...
  price2ret(TickSeries,TickTimes,Method)
```

Description

`[RetSeries,RetIntervals] = ...`
`price2ret(TickSeries,TickTimes,Method)` computes asset returns for `NUMOBS` price observations of `NUMASSETS` assets.

Input Arguments

`TickSeries` Time series of price data. `TickSeries` can be a column vector 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 that the observations across a given row occur at the same time for all columns, where each column is a price series of an individual asset.

`TickTimes` 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 (for example, yearly). If `TickTimes = []` or is unspecified, then

price2ret assumes sequential observation times from 1, 2, ..., NUMOBS.

Method — Character string indicating the compounding method to compute asset returns. If Method = 'Continuous', = [], or is unspecified, then price2ret computes continuously compounded returns. If Method = 'Periodic', then price2ret assumes simple periodic returns. Method is case insensitive.

Output Arguments

RetSeries — Array of asset returns:

- When TickSeries is a NUMOBS element column vector, RetSeries is a NUMOBS-1 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). It then normalizes it by the time interval between successive price observations.

  Assuming that

  RetIntervals(*i*) = TickTimes(*i*+1) – TickTimes(*i*)

  then if Method is 'Continuous', [], or is unspecified, price2ret computes the continuously compounded returns as

  RetSeries(*i*) = log [TickSeries(*i*+1)/TickSeries(*i*)]/RetIntervals(*i*)

  If Method is 'Periodic', then price2ret computes the simple returns as

	`RetSeries(`i`) = [TickSeries(`i`+1)/TickSeries(`i`)] – 1 /RetIntervals(`i`)`
`RetIntervals`	`NUMOBS-1` element vector of times between observations. If `TickTimes` is `[]` or is unspecified, `price2ret` assumes that all intervals are `1`.

Examples

1 Create a stock price process continuously compounded at 10 percent:

```
S = 100*exp(0.10 * [0:19]'); % Create the stock price series
```

2 Convert the price series to a 10 percent return series:

```
R = price2ret(S);   % Convert the price series to a 10 percent
                    % return series
[S [R;NaN]]  % Pad the return series so vectors are of same
             % length. price2ret computes the ith return from
             % the ith and xth 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 return series to price series

Syntax

```
[TickSeries,TickTimes] = ...
  ret2price(RetSeries,StartPrice,RetIntervals,StartTime,Method)
```

Description

```
[TickSeries,TickTimes] = ...
  ret2price(RetSeries,StartPrice,RetIntervals,StartTime,Method)
```

generates price series for the specified assets, given the asset starting prices and the return observations for each asset.

Input Arguments

`RetSeries`	Time-series array of returns. `RetSeries` can be a column vector 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 that 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`	A `NUMASSETS` element vector of initial prices for each asset, or a single scalar initial price applied to all assets. If `StartPrice = []` or is unspecified, all asset prices start at `1`.

`RetIntervals`	A `NUMOBS` element vector of time intervals between return observations, or a single scalar interval applied to all observations. If `RetIntervals` is `[]` or is unspecified, `ret2price` assumes that 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`	Character string indicating the compounding method used to compute asset returns. If `Method` is `'Continuous'`, `[]`, or is unspecified, then `ret2price` computes continuously compounded returns. If `Method` is `'Periodic'` then `ret2price` computes simple periodic returns. `Method` is case insensitive.

Output Arguments

`TickSeries` Array of asset prices:

- When `RetSeries` is a `NUMOBS` element column vector, `TickSeries` is a `NUMOBS+1` 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`.

Examples

Example 1

1 Create a stock price process continuously compounded at 10 percent:

```
S = 100*exp(0.10 * [0:19]');  % Create the stock price series
```

2 Compute 10 percent returns for reference:

```
R = price2ret(S);             % Convert the price series to a
                              % 10 percent return series
```

3 Convert the resulting return series to the original price series, and compare results:

```
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
```

Example 2

This example compares the relative price performance of the NASDAQ and the NYSE indexes (see “Example Financial Time-Series Data Sets” on page 1-12).

1 Convert the prices to returns:

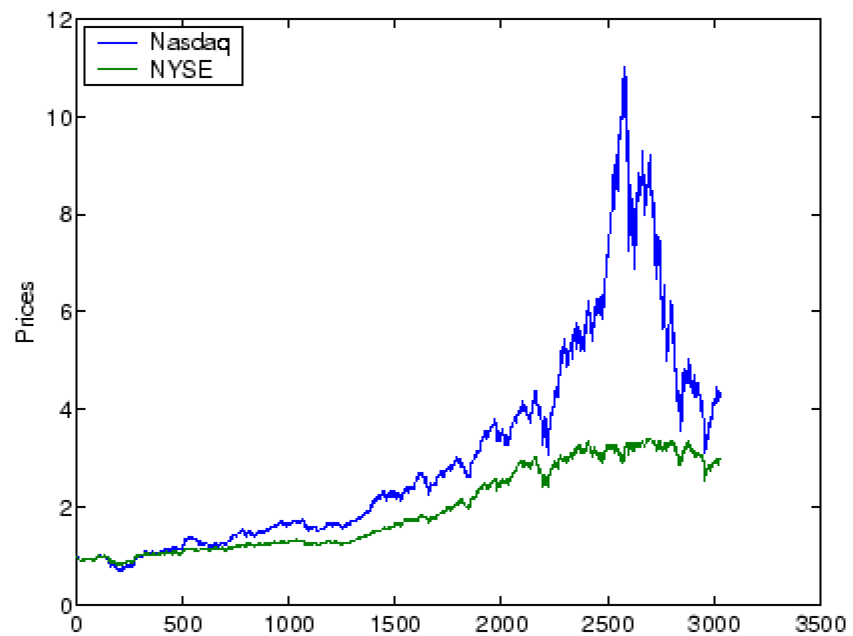
```
load garchdata
nasdaq = price2ret(NASDAQ);
```

2 Convert the returns back to prices, specifying the same starting price, 100, for each series:

```
nyse = price2ret(NYSE);
```

3 Plot both series:

```
plot(ret2price(price2ret([NASDAQ NYSE]), 100))
ylabel('Prices')
legend('Nasdaq', 'NYSE', 2)
```



The blue (upper) plot shows the NASDAQ price series. The green (lower) plot shows the NYSE price series.

See Also

price2ret

Method Reference

Monte Carlo Simulation of Stochastic Differential Equations (SDEs) (p. 14-2)	Perform Monte Carlo simulation of multivariate diffusion processes
Stochastic Differential Equation (SDE) Class Constructors (p. 14-3)	Create SDE models

Monte Carlo Simulation of Stochastic Differential Equations (SDEs)

interpolate	Perform Brownian interpolation of stochastic differential equations (SDEs)
simByEuler	Perform Euler simulation of stochastic differential equations (SDEs)
simBySolution	Simulate approximate solution of diagonal-drift `HWV` and `GBM` processes
simulate	Simulate multivariate stochastic differential equations (SDEs)
ts2func	Convert time-series arrays to callable functions of time and state

Stochastic Differential Equation (SDE) Class Constructors

bm	Construct Brownian motion models (objects of class `BM`)
cev	Create constant elasticity of variance models (objects of class `CEV`)
cir	Create Cox-Ingersoll-Ross mean-reverting square root diffusion models (objects of class `CIR`)
diffusion	Create diffusion-rate model components
drift	Create drift-rate model components
gbm	Create generalized geometric Brownian motion models (objects of class `GBM`)
hwv	Create Hull-White/Vasicek mean-reverting Gaussian diffusion models (objects of class `HWV`)
sde	Create stochastic differential equation models (objects of class `SDE`)
sdeddo	Create stochastic differential equation from drift and diffusion models (objects of class `SDEDDO`)
sdeld	Create stochastic differential equation from linear drift-rate models (objects of class `SDELD`)
sdemrd	Create stochastic differential equation (SDE) from mean-reverting drift-rate models (objects of class `SDEMRD`)

15

Methods — Alphabetical List

Purpose

Construct Brownian motion models (objects of class `BM`)

Syntax

```
BM = bm(Mu, Sigma)
BM = bm(Mu, Sigma, 'Name1', Value1, 'Name2', Value2, ...)
```

Classes

`BM`

Description

Use this class constructor to create and display Brownian motion (sometimes called *arithmetic Brownian motion* or *generalized Wiener process*) objects that derive from the `SDELD` (SDE with drift rate expressed in linear form) class. You can use `BM` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Brownian motion stochastic processes. This enables you to transform a vector of `NBROWNS` uncorrelated, zero-drift, unit-variance rate Brownian components into a vector of `NVARS` Brownian components with arbitrary drift, variance rate, and correlation structure.

The `bm` method allows you to simulate any vector-valued BM process of the form:

$$dX_t = \mu(t)dt + V(t)dW_t \qquad \textbf{(15-1)}$$

where:

- X_t is an `NVARS`-by-1 state vector of process variables.
- μ is an `NVARS`-by-1 drift-rate vector.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-1 vector of (possibly) correlated zero-drift/unit-variance rate Brownian components.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Mu`	`Mu` represents μ in Equation 15-1. If you specify `Mu` as an array, it must be an `NVARS`-by-1 column vector representing the drift rate (the expected instantaneous rate of drift, or time trend). If you specify `Mu` as a function, it calculates the expected instantaneous rate of drift. This function must generate an `NVARS`-by-1 column vector when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`Sigma`	`Sigma` represents the parameter V in Equation 15-1. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify `Sigma` as a function, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: a real-valued scalar observation

time t and an `NVARS`-by-1 state vector X_t. Although the constructor does not enforce restrictions on the sign of this argument, `Sigma` is usually specified as a positive value.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-1 column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `bm` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `bm` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `bm` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.

`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`BM` Object of class `BM` with the following displayed parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `Simulation`: A simulation function or method

- `Mu`: Access function for the input argument `Mu`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `bm` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

"Creating Brownian Motion (BM) Models" on page 5-21

See Also

`drift`, `diffusion`, `sdeld`

Purpose

Create constant elasticity of variance models (objects of class CEV)

Syntax

```
CEV = cev(Return, Alpha, Sigma)

CEV = cev(Return, Alpha, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Classes

CEV

Description

Use this class constructor to create and display CEV objects, which derive from the SDELD (SDE with drift rate expressed in linear form) class. You can use CEV objects to simulate sample paths of NVARS state variables driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time stochastic processes.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t \quad \textbf{(15-2)}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- μ is an NVARS-by-NVARS (generalized) expected instantaneous rate of return matrix.
- D is an NVARS-by-NVARS diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an NVARS-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully

captures all implementation details, which are clearly associated with a parametric form.

- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Return`	`Return` represents the parameter μ in Equation 15-2. If you specify `Return` as an array, it is a `NVARS`-by-`NVARS` 2-dimensional matrix that represents the expected (mean) instantaneous rate of return. If you specify `Return` as a function, it calculates the expected instantaneous rate of return. This function must generate an `NVARS`-by-`NVARS` matrix when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`Alpha`	`Alpha` determines the format of the parameter D in Equation 15-2. If you specify `Alpha` as an array, it represents an `NVARS`-by-1 column vector of exponents. If you specify it as a function, `Alpha` must return an `NVARS`-by-1 column vector of exponents when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

Although the constructor does not enforce restrictions on the sign of Alpha, this exponent is usually specified as a positive value.

Sigma

Sigma represents the parameter V in Equation 15-2. If you specify Sigma as an array, it represents an NVARS-by-NBROWNS 2-dimensional matrix of instantaneous volatility rates. In this case, each row of Sigma corresponds to a particular state variable. Each column of Sigma corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

If you specify it as a function, Sigma must generate an NVARS-by-NBROWNS matrix of volatility rates when invoked with two inputs: a real-valued scalar observation time t and an NVARS-by-1 state vector X_t.

Although the constructor does not enforce restrictions on the sign of Sigma, it is usually specified as a positive value.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-1 column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `cev` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `cev` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `cev` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`CEV` Object of class `CEV` with the following displayed parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `Simulation`: A simulation function or method
- `Return`: Access function for the input argument `Return`, callable as a function of time and state
- `Alpha`: Access function for the input argument `Alpha`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `cev` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

- "Creating Constant Elasticity of Variance (CEV) Models" on page 5-22
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects

See Also

`drift`, `diffusion`, `sdeld`

Purpose

Create Cox-Ingersoll-Ross mean-reverting square root diffusion models (objects of class CIR)

Syntax

```
CIR = cir(Speed, Level, Sigma)
CIR = cir(Speed, Level, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Classes

CIR

Description

Use this class constructor to create and display CIR objects, which derive from the SDEMRD (SDE with drift rate expressed in mean-reverting form) class. You can use CIR objects to simulate sample paths of NVARS state variables expressed in mean-reverting drift-rate form. These state variables are driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time CIR stochastic processes with square root diffusions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\frac{1}{2}})V(t)dW_t \tag{15-3}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- S is an NVARS-by-NVARS matrix of mean reversion speeds (the rate of mean reversion).
- L is an NVARS-by-1 vector of mean reversion levels (long-run mean or level).
- D is an NVARS-by-NVARS diagonal matrix, where each element along the main diagonal is the square root of the corresponding element of the state vector.
- V is an NVARS-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Speed` — `Speed` represents S in Equation 15-3. If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate or speed at which the state vector reverts to its long-run average `Level`).

If you specify `Speed` as a function, it must generate an `NVARS`-by-`NVARS` matrix of reversion rates when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

`Level` — `Level` represents L in Equation 15-3. If you specify `Level` as an array, it must be an `NVARS`-by-1 column vector of reversion levels.

If you specify `Level` as a function, it must generate an `NVARS`-by-1 column vector of reversion levels when

invoked with two inputs: t and an NVARS-by-1 state vector X_t.

Sigma	Sigma represents the parameter V in Equation 15-3. If you specify Sigma as an array, it must be an NVARS-by-NBROWNS 2-dimensional matrix of instantaneous volatility rates. In this case, each row of Sigma corresponds to a particular state variable. Each column of Sigma corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify Sigma as a function, it must generate an NVARS-by-NBROWNS matrix of volatility rates when invoked with two inputs: a real-valued scalar observation time t and an NVARS-by-1 state vector X_t.

Note Although the constructor does not enforce restrictions on the sign of these input arguments, they are usually specified as positive values.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `cir` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `cir` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `cir` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.

`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`CIR` — Object of class `CIR` with the following displayed parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `Simulation`: A simulation function or method

- `Speed`: Access function for the input argument `Speed`, callable as a function of time and state
- `Level`: Access function for the input argument `Level`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t followed by a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `cir` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

"Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models" on page 5-25

See Also

`drift`, `diffusion`, `sdeddo`

Purpose

Create diffusion-rate model components

Syntax

```
DiffusionRate = diffusion(Alpha, Sigma)
```

Classes

`Diffusion`

Description

Use the `diffusion` class constructor to specify the diffusion-rate component of continuous-time stochastic differential equations (SDEs). The diffusion-rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The diffusion-rate specification can be any `NVARS`-by-`NBROWNS` matrix-valued function G of the general form:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t) \qquad \textbf{(15-4)}$$

associated with a vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X_t is an `NVARS`-by-1 state vector of process variables.
- dW_t is an `NBROWNS`-by-1 Brownian motion vector.
- D is an `NVARS`-by-`NVARS` diagonal matrix, in which each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is described in "Input Arguments" on page 15-20.

The diffusion-rate specification is very flexible, and it provides direct parametric support for static volatilities and state vector exponents. It is also extensible, and provides indirect support for dynamic/nonlinear models via an interface. This enables you to specify virtually any diffusion-rate specification.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Alpha` — `Alpha` determines the format of the parameter D in Equation 15-4. If you specify `Alpha` as an array, it must be an `NVARS`-by-1 column vector of exponents. If you specify `Alpha` as a function, it must return an `NVARS`-by-1 column vector of exponents when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

`Sigma` — `Sigma` represents the parameter V in Equation 15-4. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify `Sigma` as a function, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates

when invoked with two inputs: a real-valued scalar observation time t and an NVARS-by-1 state vector X_t.

Note Although the diffusion constructor enforces no restrictions on the signs of these volatility parameters, they are usually specified as positive values.

Output Arguments

DiffusionRate — Object of class diffusion that encapsulates the composite diffusion-rate specification, with the following displayed parameters:

- Rate: The diffusion-rate function, G. Rate is the diffusion-rate calculation engine. It accepts the current time t and an NVARS-by-1 state vector X_t as inputs, and returns an NVARS-by-1 diffusion-rate vector.
- Alpha: Access function for the input argument Alpha.
- Sigma: Access function for the input argument Sigma.

Remarks

When you specify the input arguments Alpha and Sigma as MATLAB arrays, they are associated with a specific parametric form. By contrast, when you specify either Alpha or Sigma as a function, you can customize virtually any diffusion-rate specification.

Accessing the output diffusion-rate parameters Alpha and Sigma with no inputs simply returns the original input specification. Thus, when you invoke diffusion-rate parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke diffusion-rate parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters `Alpha` and `Sigma` accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Specifically, parameters `Alpha` and `Sigma` evaluate the corresponding diffusion-rate component. Even if you originally specified an input as an array, `diffusion` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

"Creating Drift and Diffusion Objects" on page 5-16

See Also

`drift`, `sdeddo`

Purpose

Create drift-rate model components

Syntax

```
DriftRate = drift(A, B)
```

Classes

`Drift`

Description

Use this constructor to specify the drift-rate component of continuous-time stochastic differential equations (SDEs). The drift-rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The drift-rate specification can be any `NVARS`-by-1 vector-valued function F of the general form:

$$F(t, X_t) = A(t) + B(t)X_t \tag{15-5}$$

associated with a vector-valued SDE of the form

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X_t is an `NVARS`-by-1 state vector of process variables.
- dW_t is an `NBROWNS`-by-1 Brownian motion vector.
- A and B are described in "Input Arguments" on page 15-24.

The drift-rate specification is very flexible, and it provides direct parametric support for static/linear drift models. It is also extensible, and provides indirect support for dynamic/nonlinear models via an interface. This enables you to specify virtually any drift-rate specification.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`A`	This argument represents the parameter A in Equation 15-5. If you specify `A` as an array, it must be an `NVARS`-by-1 column vector. If you specify `A` as a function, it must return an `NVARS`-by-1 column vector when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`B`	This argument represents the parameter B in Equation 15-5. If you specify `B` as an array, it must be an `NVARS`-by-`NVARS` 2-dimensional matrix. If you specify `B` as a function, it must return an `NVARS`-by-`NVARS` column vector when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

Output Arguments

`DriftRate` Object of class `drift` that encapsulates the composite drift-rate specification, with the following displayed parameters:

- `Rate`: The drift-rate function, F. `Rate` is the drift-rate calculation engine. It accepts the current time t and an `NVARS`-by-1 state vector X_t as inputs, and returns an `NVARS`-by-1 drift-rate vector.
- `A`: Access function for the input argument `A`.
- `B`: Access function for the input argument `B`.

Remarks

When you specify the input arguments `A` and `B` as MATLAB arrays, they are associated with a linear drift parametric form. By contrast, when you specify either `A` or `B` as a function, you can customize virtually any drift-rate specification.

Accessing the output drift-rate parameters `A` and `B` with no inputs simply returns the original input specification. Thus, when you invoke drift-rate parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke drift-rate parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters `A` and `B` accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Specifically, parameters `A` and `B` evaluate the corresponding drift-rate component. Even if you originally specified an input as an array, `drift` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

"Creating Drift and Diffusion Objects" on page 5-16

See Also

`diffusion`, `sdeddo`

gbm

Purpose

Create generalized geometric Brownian motion models (objects of class GBM)

Syntax

```
GBM = gbm(Return, Sigma)
```

```
GBM = gbm(Return, Sigma, 'Name1', Value1, 'Name2', Value2,
...)
```

Classes

GBM

Description

Use this constructor to create and display GBM objects, which derive from the CEV (constant elasticity of variance) class. You can use GBM objects to simulate sample paths of NVARS state variables driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time GBM stochastic processes.

This method allows simulation of vector-valued GBM processes of the form:

$$dX_t = \mu(t) X_t dt + D(t, X_t) V(t) dW_t \quad \textbf{(15-6)}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- μ is an NVARS-by-NVARS generalized expected instantaneous rate of return matrix.
- D is an NVARS-by-NVARS diagonal matrix, where each element along the main diagonal is the square root of the corresponding element of the state vector.
- V is an NVARS-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Return` — `Return` represents the parameter μ in Equation 15-6. If you specify `Return` as an array, it must be an `NVARS`-by-`NVARS` matrix representing the expected (mean) instantaneous rate of return. If you specify it as a function, `Return` must return an `NVARS`-by-`NVARS` matrix when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

`Sigma` — `Sigma` represents the parameter V in Equation 15-6. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify it as a function, `Sigma` must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two

inputs: a real-valued scalar observation time t and an NVARS-by-1 state vector X_t.Although the gbm constructor enforces no restrictions on the sign of Sigma volatilities, they are usually specified as positive values.

Optional Input Arguments

You specify optional input arguments as variable-length lists of matching parameter name/value pairs: 'Name1', Value1, 'Name2', Value2, ... and so on. The following rules apply when specifying parameter-name pairs:

- You specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

StartTime	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for StartTime, the default is 0.
StartState	Scalar, NVARS-by-1 column vector, or NVARS-by-NTRIALS matrix of initial values of the state variables. If StartState is a scalar, gbm applies the same initial value to all state variables on all trials. If StartState is a column vector, gbm applies a unique initial value to each state variable on all trials. If StartState is a matrix, gbm applies a unique initial value to each state variable on each trial.

	If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix.A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`GBM` Object of class `gbm` with the following displayed parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at `StartTime`
- `Correlation`: Access function for the `Correlation` input, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state

- `Simulation`: A simulation function or method
- `Return`: Access function for the input argument `Return`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `gbm` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

- "Creating Geometric Brownian Motion (GBM) Models" on page 5-23
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects

See Also

`drift`, `diffusion`, `cev`

Purpose

Create Hull-White/Vasicek mean-reverting Gaussian diffusion models (objects of class `HWV`)

Syntax

```
HWV = hwv(Speed, Level, Sigma)
HWV = hwv(Speed, Level, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Classes

`HWV`

Description

Use this constructor to create and display `HWV` objects, which derive from the `SDEMRD` (SDE with drift rate expressed in mean-reverting form) class. You can use `HWV` objects to simulate sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form. These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time `HWV` stochastic processes with Gaussian diffusions.

This method allows you to simulate vector-valued `HWV` processes of the form:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t \tag{15-7}$$

where:

- X_t is an `NVARS`-by-1 state vector of process variables.
- S is an `NVARS`-by-`NVARS` of mean reversion speeds (the rate of mean reversion).
- L is an `NVARS`-by-1 vector of mean reversion levels (long-run mean or level).
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- $dW_t$is an `NBROWNS`-by-1 Brownian motion vector.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Speed` — `Speed` represents the function S in Equation 15-7. If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate at which the state vector reverts to its long-run average `Level`). If you specify `Speed` as a function, it calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

`Level` — `Level` represents the function L in Equation 15-7. If you specify `Level` as an array, it must be an `NVARS`-by-1 column vector of reversion levels. If you specify `Level` as a function, it must generate an `NVARS`-by-1 column vector of reversion levels when called with two inputs: a

	real-valued scalar observation time t and an NVARS-by-1 state vector X_t.
Sigma	Sigma represents the parameter V in Equation 15-7. If you specify Sigma as an array, it must be an NVARS-by-NBROWNS matrix of instantaneous volatility rates. In this case, each row of Sigma corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify it as a function, Sigma must return an NVARS-by-NBROWNS matrix of volatility rates when invoked with two inputs: a real-valued scalar observation time t and an NVARS-by-1 state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of any of these input arguments, each argument is usually specified as a positive value.

Optional Input Arguments

You specify optional input arguments as variable-length lists of matching parameter name/value pairs: 'Name1', Value1, 'Name2', Value2, ... and so on. The following rules apply when specifying parameter-name pairs:

- You specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

StartTime	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for StartTime, the default is 0.
StartState	Scalar, NVARS-by-1 column vector, or NVARS-by-NTRIALS matrix of initial values of the state variables. If StartState is a scalar, hwv applies the same initial value to all state variables on all trials. If StartState is a column vector, hwv applies a unique initial value to each state variable on all trials. If StartState is a matrix, hwv applies a unique initial value to each state variable on each trial. If you do not specify a value for StartState, all variables start at 1.
Correlation	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify Correlation as an NBROWNS-by-NBROWNS positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an NBROWNS-by-NBROWNS positive semidefinite correlation matrix. A Correlation matrix represents a static condition. As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure. If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.
Simulation	A user-defined simulation function or SDE simulation method. If you do not specify a value for Simulation, the default method is simulation by Euler approximation (simByEuler).

Output Arguments

`HWV` Object of class `hwv` with the following displayed parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at `StartTime`
- `Correlation`: Access function for the `Correlation` input, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `Simulation`: A simulation function or method
- `Speed`: Access function for the input argument `Speed`, callable as a function of time and state
- `Level`: Access function for the input argument `Level`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, hwv treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

"Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 5-26

See Also

drift, diffusion, sdeddo

Purpose

Perform Brownian interpolation of stochastic differential equations (SDEs)

Syntax

```
[XT, T] = SDE.interpolate(T, Paths)
[XT, T] = SDE.interpolate(T, Paths, 'Name1', Value1,
'Name2', Value2, ...)
```

Classes

All classes in the SDE class hierarchy.

Description

This method performs a Brownian interpolation into a user-specified time-series array, based on a piecewise-constant Euler sampling approach.

Consider a vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X is an *NVARS*-by-1 state vector.
- F is an *NVARS*-by-1 drift-rate vector-valued function.
- G is an *NVARS*-by-*NBROWNS* diffusion-rate matrix-valued function.
- W is an *NBROWNS*-by-1 Brownian motion vector.

Given a user-specified time-series array associated with this equation, this method performs a Brownian (stochastic) interpolation by sampling from a conditional Gaussian distribution. This sampling technique is sometimes called a *Brownian bridge*.

Note Unlike simulation methods, the `interpolate` method does not support user-specified noise processes.

Input Arguments

`SDE`	Stochastic differential equation model.
`T`	`NTIMES` element vector of interpolation times. The length of this vector determines the number of rows in the interpolated output time series `XT`.
`Paths`	`NPERIODS`-by-`NVARS`-by-`NTRIALS` time-series array of sample paths of correlated state variables. For a given trial, each row of this array is the transpose of the state vector X_t at time t. `Paths` is the initial time-series array into which `interpolate` performs the Brownian interpolation.

Optional Input Arguments

You specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- You specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`Times`	Vector of monotonically increasing observation times associated with the time-series input `Paths`. If you do not specify a value for this parameter, `Times` is a zero-based, unit-increment column vector of length `NPERIODS`.
`Refine`	Scalar logical flag that indicates whether `interpolate` uses the interpolation times you request (see `T`) to refine the interpolation as new information becomes available. If you do not specify a value for this argument or set it to `FALSE` (the default value), `interpolate` bases the interpolation only on the state information specified in `Paths`. If you set `Refine` to `TRUE`, `interpolate` inserts all new interpolated states into the existing `Paths` array as they become available. This refines the interpolation grid available to subsequent interpolation times for the duration of the current trial.
`Processes`	Function or cell array of functions that indicates a sequence of background processes or state vector adjustments of the form $X_t = P(t, X_t)$ The `interpolate` method runs processing functions at each interpolation time. They must accept the current interpolation time t, and the current state vector X_t, and return a state vector that may be an adjustment to the input state. If you specify more than one processing function, `interpolate` invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and so on. If you do not specify a processing function, `interpolate` makes no adjustments and performs no processing.

Output Arguments

XT	NTIMES-by-NVARS-by-NTRIALS time-series array of interpolated state variables. For a given trial, each row of this array is the transpose of the interpolated state vector X_t at time t. XT is the interpolated time series formed by interpolating into the input Paths time-series array.
T	NTIMES-by-1 column vector of interpolation times associated with the output time series XT. If the input interpolation time vector T contains no missing observations (NaNs), this output is the same time vector as T, but with the NaNs removed. This reduces the length of T and the number of rows of XT.

Remarks

- The interpolate method assumes that all model parameters are piecewise-constant, and evaluates them from the most recent observation time in Times that precedes a specified interpolation time in T. This is consistent with the Euler approach of Monte Carlo simulation.
- When an interpolation time falls outside the interval specified by Times, an Euler simulation extrapolates the time series by using the nearest available observation.
- The user-defined time series Paths and corresponding observation Times must be fully observed (no missing observations denoted by NaNs).
- The interpolate method assumes that the user-specified time-series array Paths is associated with the SDE object. For example, the Times/Paths input pair is the result of an initial course-grained simulation. However, the interpolation ignores the initial conditions of the SDE object (StartTime and StartState), allowing the user-specified Times/Paths input series to take precedence.

Examples

Stochastic Interpolation Without Refinement

Many applications require knowledge of the state vector at intermediate sample times that are initially unavailable. One way to approximate these intermediate states is to perform a deterministic interpolation. However, deterministic interpolation techniques fail to capture the correct probability distribution at these intermediate times. Brownian (or stochastic) interpolation captures the correct joint distribution by sampling from a conditional Gaussian distribution. This sampling technique is sometimes referred to as a *Brownian Bridge*.

The default stochastic interpolation technique is designed to interpolate into an existing time series and ignore new interpolated states as additional information becomes available. This technique is the usual notion of interpolation, which is called *Interpolation without refinement*.

Alternatively, the interpolation technique may insert new interpolated states into the existing time series upon which subsequent interpolation is based, thereby refining information available at subsequent interpolation times. This technique is called *interpolation with refinement*.

Interpolation without refinement is a more traditional technique, and is most useful when the input series is closely spaced in time. In this situation, interpolation without refinement is a good technique for inferring data in the presence of missing information, but is inappropriate for extrapolation. Interpolation with refinement is more suitable when the input series is widely spaced in time, and is useful for extrapolation.

The stochastic interpolation method is available to any model. It is best illustrated, however, by way of a constant-parameter Brownian motion process. Consider a correlated, bivariate Brownian motion (`BM`) model of the form:

$$dX_{1t} = 0.3dt + 0.2dW_{1t} - 0.1dW_{2t}$$
$$dX_{2t} = 0.4dt + 0.1dW_{1t} - 0.2dW_{2t}$$
$$E[dW_{1t}dW_{2t}] = \rho dt = 0.5dt$$

1 Create a BM object to represent the bivariate model:

```
mu    = [0.3 ; 0.4];
sigma = [0.2  -0.1 ; 0.1  -0.2];
rho   = [ 1    0.5 ; 0.5    1 ];
obj   = bm(mu,  sigma, 'Correlation', rho);
```

2 Assuming that the drift (Mu) and diffusion (Sigma) parameters are annualized, simulate a single Monte Carlo trial of daily observations for one calendar year (250 trading days):

```
randn('state', 0)
dt    = 1/250;  % 1 trading day = 1/250 years
[X,T] = obj.simulate(250, 'DeltaTime', dt);
```

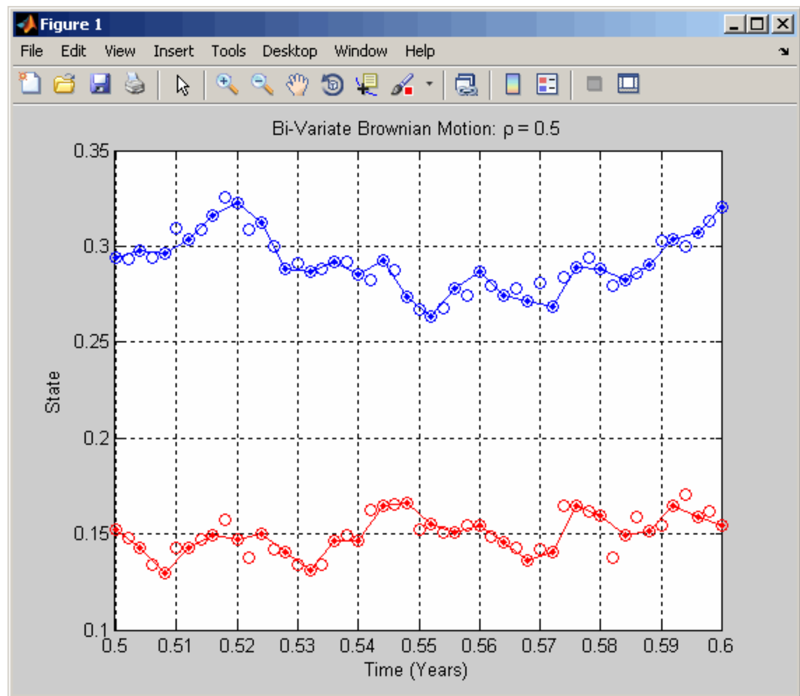
3 At this point, it is helpful to examine a small interval in detail.

a Interpolate into the simulated time series with a Brownian bridge:

```
t = ((T(1) + dt/2) : (dt/2) : (T(end) - dt/2));
x = obj.interpolate(t, X, 'Times', T);
```

b Plot both the simulated and interpolated values:

```
plot(T, X(:,1), '.-red', T, X(:,2), '.-blue'), ...
  grid('on'), hold('on')
plot(t, x(:,1), 'o red', t, x(:,2), 'o blue'), ...
  hold('off')
xlabel('Time (Years)'), ylabel('State')
title('Bi-Variate Brownian Motion: \rho = 0.5')
axis([0.4999 0.6001 0.1 0.35])
```



In this plot:

- The solid red and blue dots indicate the simulated states of the bivariate model.
- The straight lines that connect the solid dots indicate intermediate states that would be obtained from a deterministic linear interpolation.
- Open circles indicate interpolated states.
- Open circles associated with every other interpolated state encircle solid dots associated with the corresponding simulated state. However, interpolated states at the midpoint of each time increment typically deviate from the straight line connecting each solid dot.

Monte Carlo Simulation of Conditional Gaussian Distributions

You can gain additional insight into the behavior of stochastic interpolation by regarding a Brownian bridge as a Monte Carlo simulation of a conditional Gaussian distribution.

This example examines the behavior of a Brownian bridge over a single time increment.

1 Divide a single time increment of length `dt` into 10 subintervals:

```
nTrials  = 25000;  % # of Trials at each time
n        = 125;    % index of simulated state near middle
times    = (T(n) : (dt/10) : T(n + 1));
```

2 In each subinterval, take 25000 independent draws from a Gaussian distribution, conditioned on the simulated states to the left and right:

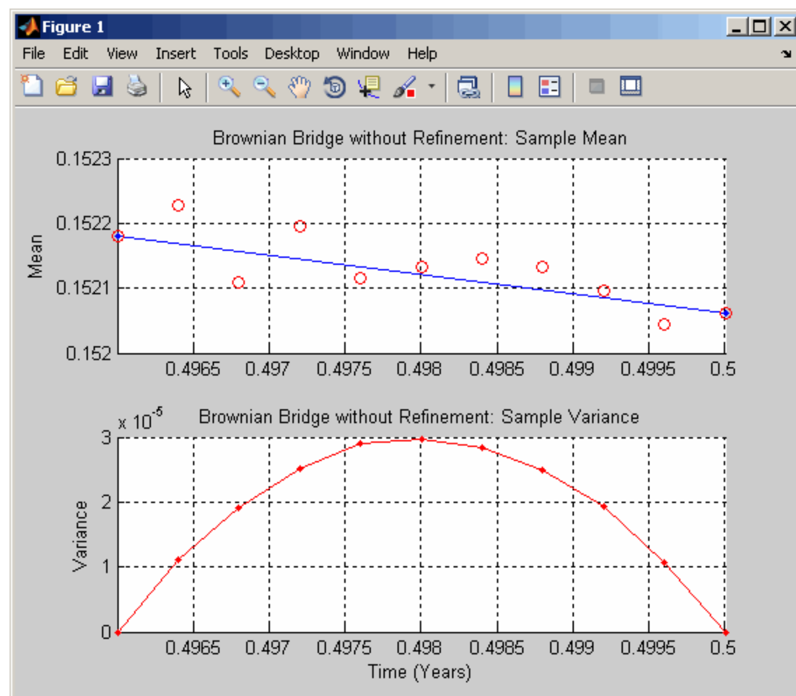
```
average  = zeros(length(times),1);
variance = zeros(length(times),1);
for i = 1:length(times)
    t = times(i);
    x = obj.interpolate(t(ones(nTrials,1)), X, 'Times', T);
    average(i)  = mean(x(:,1));
    variance(i) = var(x(:,1));
end
```

3 Plot the sample mean and variance of each state variable:

Note The following graph plots the sample statistics of the first state variable only, but similar results hold for any state variable.

```
subplot(2,1,1), hold('on'), grid('on')
plot([T(n) T(n + 1)], [X(n,1) X(n + 1,1)], '.-b')
plot(times, average, 'or'), hold('off')
title('Brownian Bridge without Refinement: Sample Mean')
ylabel('Mean')
```

```
limits = axis; axis([T(n) T(n + 1) limits(3:4)])

subplot(2,1,2), hold('on'), grid('on')
plot(T(n), 0, '.-b', T(n + 1), 0, '.-b')
plot(times, variance, '.-r'), hold('off')
title('Brownian Bridge without Refinement: Sample Variance')
xlabel('Time (Years)'), ylabel('Variance')
limits = axis; axis([T(n) T(n + 1) limits(3:4)])
```



The Brownian interpolation within the chosen interval, *dt*, illustrates the following:

- The conditional mean of each state variable lies on a straight line segment between the original simulated states at each endpoint.

- The conditional variance of each state variable is a quadratic function. This function attains its maximum midway between the interval endpoints, and is zero at each endpoint.
- The maximum variance, although dependent upon the actual model diffusion-rate function *G(t,X)*, is the variance of the sum of `NBROWNS` correlated Gaussian variates scaled by the factor *dt/4*.

The previous plot highlights interpolation without refinement, in that none of the interpolated states take into account new information as it becomes available. If you had performed interpolation with refinement, new interpolated states would have been inserted into the time series and made available to subsequent interpolations on a trial-by-trial basis. In this case, all random draws for any given interpolation time would be identical. Also, the plot of the sample mean would exhibit greater variability, but would still be clustered around the straight line segment between the original simulated states at each endpoint. The plot of the sample variance, however, would be zero for all interpolation times, exhibiting no variability.

See Also

`simulate`, `sde`

Purpose

Create stochastic differential equation models (objects of class SDE)

Syntax

```
SDE = sde(DriftRate, DiffusionRate)

SDE = sde(DriftRate, DiffusionRate, 'Name1', Value1,
'Name2', Value2, ...)
```

Classes

SDE

Description

Use this constructor to create and display SDE objects. You can use SDE objects to simulate sample paths of NVARS state variables driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time stochastic processes.

This method enables you to simulate any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \quad \textbf{(15-8)}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- dW_t is an NBROWNS-by-1 Brownian motion vector.
- F is an NVARS-by-1 vector-valued drift-rate function.
- G is an NVARS-by-NBROWNS matrix-valued diffusion-rate function.

Input Arguments

`DriftRate`	User-defined drift-rate function, denoted by F in Equation 15-8. `DriftRate` is a function that returns an `NVARS`-by-1 drift-rate vector when called with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t. Alternatively, `DriftRate` may also be an object of class `Drift` that encapsulates the drift-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object; it uses no other class information.
`DiffusionRate`	User-defined diffusion-rate function, denoted by G in Equation 15-8. `DiffusionRate` is a function that returns an `NVARS`-by-`NBROWNS` diffusion-rate matrix when called with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t. Alternatively, `DiffusionRate` may also be an object of class `Diffusion` that encapsulates the diffusion-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object; it uses no other class information.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-1 column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sde` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sde` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sde` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE` Object of class `sde` with the following parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `Simulation`: A simulation function or method

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sde` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

- "Creating Base SDE Objects" on page 5-14
- Implementing Multidimensional Equity Market Models, Implementation 1: Using SDE Objects

See Also

`drift`, `diffusion`

sdeddo

Purpose

Create stochastic differential equation from drift and diffusion models (objects of class SDEDDO)

Syntax

```
SDE = sdeddo(DriftRate, DiffusionRate)
```

```
SDE = sdeddo(DriftRate, DiffusionRate, 'Name1', Value1,
'Name2', Value2, ...)
```

Classes

SDEDDO

Description

Use this constructor to create and display SDEDDO objects, specifically instantiated with objects of class drift and diffusion. These restricted SDEDDO objects contain the input drift and diffusion objects; therefore, you can directly access their displayed parameters.

This abstraction also generalizes the notion of drift and diffusion-rate objects as functions that sdeddo evaluates for specific values of time t and state X_t. Like SDE objects, SDEDDO objects allow you to simulate sample paths of NVARS state variables driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time stochastic processes.

This method enables you to simulate any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \tag{15-9}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- dW_t is an NBROWNS-by-1 Brownian motion vector.
- F is an NVARS-by-1 vector-valued drift-rate function.
- G is an NVARS-by-NBROWNS matrix-valued diffusion-rate function.

Input Arguments

`DriftRate`	Object of class `drift` that encapsulates a user-defined drift-rate specification, represented as F in Equation 15-9.
`DiffusionRate`	Object of class `diffusion` that encapsulates a user-defined diffusion-rate specification, represented as G in Equation 15-9.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-1 column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sdeddo` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sdeddo` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sdeddo` applies a unique initial value to each state variable on each trial.

	If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE` Object of class `sdeddo` with the following parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state

- `A`: Access function for the drift-rate property `A`, callable as a function of time and state
- `B`: Access function for the drift-rate property `B` , callable as a function of time and state
- `Alpha`: Access function for the diffusion-rate property `Alpha`, callable as a function of time and state
- `Sigma`: Access function for the diffusion-rate property `Sigma`, callable as a function of time and state
- `Simulation`: A simulation function or method

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdeddo` treats as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

- "Creating Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO)" on page 5-19
- Implementing Multidimensional Equity Market Models, Implementation 2: Using SDEDDO Objects

See Also

`drift`, `diffusion`, `sde`

Purpose

Create stochastic differential equation from linear drift-rate models (objects of class SDELD)

Syntax

```
SDE = sdeld(A, B, Alpha, Sigma)
```

```
SDE = sdeld(A, B, Alpha, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Classes

SDELD

Description

Use this constructor to display SDE objects whose drift rate is expressed in linear drift-rate form and that derive from the SDEDDO (SDE from drift and diffusion objects class).

You can use SDELD objects to simulate sample paths of NVARS state variables expressed in linear drift-rate form. They provide a parametric alternative to the mean-reverting drift form (see sdemrd).

These state variables are driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time stochastic processes with linear drift-rate functions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = (A(t) + B(t)X_t)dt + D(t, X_t^{\alpha(t)})V(t)dW_t \quad \textbf{(15-10)}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- A is an NVARS-by-1 vector.
- B is an NVARS-by-NVARS matrix.
- D is an NVARS-by-NVARS diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an NVARS-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`A`	`A` represents the parameter A in Equation 15-10. If you specify `A` as an array, it must be an `NVARS`-by-1 column vector of intercepts. If you specify `A` as a function, it must generate an `NVARS`-by-1 column vector of intercepts when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`B`	`B` represents the parameter B in Equation 15-10. If you specify `B` as an array, it must be an`NVARS`-by-`NVARS` matrix of state vector coefficients. If you specify `B` as a function, it must generate an `NVARS`-by-`NVARS` matrix of state vector coefficients when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

`Alpha`	`Alpha` determines the format of the parameter D in Equation 15-10. If you specify `Alpha` as an array, it represents an `NVARS`-by-1 column vector of exponents. If you specify it as a function, it must return an `NVARS`-by-1 column vector of exponents when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`Sigma`	`Sigma` represents the parameter V in Equation 15-10. If you specify `Sigma` as an array, it represents is an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify it as a function, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of `Alpha` or `Sigma`, they are usually specified as positive values.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sdeld` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sdeld` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sdeld` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for

`Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE` Object of class `sdeld` with the following parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `A`: Access function for the input argument `A`, callable as a function of time and state
- `B`: Access function for the input argument `B`, callable as a function of time and state
- `Alpha`: Access function for the input argument `Alpha`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state
- `Simulation`: A simulation function or method

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test

the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdeld` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

- "Creating Stochastic Differential Equations from Linear Drift (SDELD)" on page 5-20
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects

See Also

`drift`, `diffusion`, `sdeddo`

Purpose

Create stochastic differential equation (SDE) from mean-reverting drift-rate models (objects of class SDEMRD)

Syntax

```
SDE = sdemrd(Speed, Level, Alpha, Sigma)
```

```
SDE = sdemrd(Speed, Level, Alpha, Sigma, 'Name1', Value1,
'Name2', Value2, ...)
```

Classes

SDEMRD

Description

Use this class constructor to create and display SDE objects whose drift rate is expressed in mean-reverting drift-rate form and which derive from the SDEDDO class (SDE from drift and diffusion objects). You can use SDEMRD objects to simulate of sample paths of NVARS state variables expressed in mean-reverting drift-rate form, and provide a parametric alternative to the linear drift form (see sdeld). These state variables are driven by NBROWNS Brownian motion sources of risk over NPERIODS consecutive observation periods, approximating continuous-time stochastic processes with mean-reverting drift-rate functions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\alpha(t)})V(t)dW_t \quad \textbf{(15-11)}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- S is an NVARS-by-NVARS matrix of mean reversion speeds.
- L is an NVARS-by-1 vector of mean reversion levels.
- D is an NVARS-by-NVARS diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an NVARS-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

You can specify required input parameters as one of the following types:

- A MATLAB® array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

The required input parameters are as follows:

`Speed`	`Speed` represents the parameter S in Equation 15-11. If you specify `Speed` as an array, it represents an `NVARS`-by-`NVARS` 2-dimensional matrix of mean-reversion speeds (the rate or speed at which the state vector reverts to its long-run average `Level`). If you specify `Speed` as a function, `Speed` calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`Level`	`Level` represents the parameter L in Equation 15-11. If you specify `Level` as an array, it must be an `NVARS`-by-1 column vector of reversion levels. If you specify `Level` as a function, it must generate an `NVARS`-by-1 column vector of reversion levels when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

`Alpha`	`Alpha` determines the format of the parameter D in Equation 15-11. If you specify `Alpha` as an array, it must be an `NVARS`-by-1 column vector of exponents. If you specify it as a function, it must return an `NVARS`-by-1 column vector of exponents when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.
`Sigma`	`Sigma` represents the parameter V in Equation 15-11. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. If you specify it as a function, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of any of these input arguments, they are usually specified as positive values.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.

- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-1 column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sdemrd` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sdemrd` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sdemrd` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). You can specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure.

If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.

`Simulation` A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE` Object of class `SDEMRD`, with the following parameters:

- `StartTime`: Initial observation time
- `StartState`: Initial state at time `StartTime`
- `Correlation`: Access function for the `Correlation` input argument, callable as a function of time
- `Drift`: Composite drift-rate function, callable as a function of time and state
- `Diffusion`: Composite diffusion-rate function, callable as a function of time and state
- `Speed`: Access function for the input argument `Speed`, callable as a function of time and state
- `Level`: Access function for the input argument `Level`, callable as a function of time and state
- `Alpha`: Access function for the input argument `Alpha`, callable as a function of time and state
- `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state
- `Simulation`: A simulation function or method

Remarks

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdemrd` treats it as a static function of time and state, thereby guaranteeing that all parameters are accessible by the same interface.

Examples

See "Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD)" on page 5-24.

See Also

`drift`, `diffusion`, `sdeddo`

Purpose

Perform Euler simulation of stochastic differential equations (SDEs)

Syntax

```
[Paths, Times, Z] = SDE.simByEuler(NPERIODS)
```

```
[Paths, Times, Z] = SDE.simByEuler(NPERIODS, 'Name1',
Value1, 'Name2', Value2, ...)
```

Classes

All classes in the SDE class hierarchy.

Description

This method simulates any vector-valued SDE of the form

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X is an *NVARS*-by-1 state vector of process variables (for example, short rates or equity prices) to simulate.
- W is an *NBROWNS*-by-1 Brownian motion vector.
- F is an *NVARS*-by-1 vector-valued drift-rate function.
- G is an *NVARS*-by-*NBROWNS* matrix-valued diffusion-rate function.

`simByEuler` simulates `NTRIALS` sample paths of `NVARS` correlated state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, using the Euler approach to approximate continuous-time stochastic processes.

Input Arguments

`SDE`	Stochastic differential equation object created with the `sdeddo` class constructor.
`NPERIODS`	Positive scalar integer number of simulation periods. The value of `NPERIODS` determines the number of rows of the simulated output series.

Optional Input Arguments

You can specify optional inputs as matching parameter name/value pairs as follows:

- You specify the parameter name as a character string, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`NTRIALS`	Positive scalar integer number of simulated trials (sample paths) of `NPERIODS` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.
`DeltaTime`	Scalar or `NPERIODS`-by-1 column vector of positive time increments between observations. `DeltaTime` represents the familiar *dt* found in stochastic differential equations, and determines the times at which the simulated paths of the output state variables are reported. If you do not specify a value for this argument, the default is `1`.
`NSTEPS`	Positive scalar integer number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`). The `simByEuler` method partitions each time increment *dt* into `NSTEPS` subintervals of length *dt*/`NSTEPS`, and refines the simulation by evaluating the simulated state vector at `NSTEPS - 1` intermediate points. Although `simByEuler` does not report the output state vector at these intermediate points, the refinement improves accuracy by allowing the simulation to more closely approximate the underlying continuous-time process. If you do not specify a value

for NSTEPS, the default is 1, indicating no intermediate evaluation.

`Antithetic` Scalar logical flag that indicates whether simByEuler uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes). When Antithetic is TRUE (logical 1), simByEuler performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs:

- Odd trials (1,3,5,...) correspond to the primary Gaussian paths.
- Even trials (2,4,6,...) are the matching antithetic paths of each pair derived by negating the Gaussian draws of the corresponding primary (odd) trial.

If you specify Antithetic to be any value other than TRUE, simByEuler assumes that it is FALSE (logical 0) by default, and does not perform antithetic sampling. When you specify an input noise process (see Z), simByEuler ignores the value of Antithetic.

`Z` Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. You can specify this argument as a function, or as an (NPERIODS * NSTEPS)-by-NBROWNS-by-NTRIALS 3-dimensional array of dependent random variates. If you specify Z as a function, it must return an NBROWNS-by-1 column vector, and you must call it with two inputs: a real-valued scalar observation time t and an NVARS-by-1 state vector X_t. If you do not specify a value for Z, simByEuler generates correlated Gaussian variates based on the Correlation member of the SDE object.

StorePaths	Scalar logical flag that indicates how the output array Paths is stored and returned to the caller. If StorePaths is TRUE (the default value) or is unspecified, simByEuler returns Paths as a 3-dimensional time-series array. If StorePaths is FALSE (logical 0), simByEuler returns the Paths output array as an empty matrix.
Processes	Function or cell array of functions that indicates a sequence of end-of-period processes or state vector adjustments of the form $X_t = P(t, X_t)$ simByEuler applies processing functions at the end of each observation period. These functions must accept the current observation time t and the current state vector X_t, and return a state vector that may be an adjustment to the input state. If you specify more than one processing function, simByEuler invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and more. If you do not specify a processing function, simByEuler makes no adjustments and performs no processing.

Output Arguments

`Paths`	`(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` 3-dimensional time-series array, consisting of simulated paths of correlated state variables. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t. When the input flag `StorePaths = FALSE`, `simByEuler` returns `Paths` as an empty matrix.
`Times`	`(NPERIODS + 1)`-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.
`Z`	`(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` 3-dimensional time-series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation.

Remarks

- This simulation engine provides a discrete-time approximation of the underlying generalized continuous-time process. The simulation is derived directly from the stochastic differential equation of motion. Thus, the discrete-time process approaches the true continuous-time process only as `DeltaTime` approaches zero.
- The input argument `Z` allows you to directly specify the noise-generation process. This process takes precedence over the `Correlation` parameter of the `SDE` object and the value of the `Antithetic` input flag. If you do not specify a value for `Z`, `simByEuler` generates correlated Gaussian variates, with or without antithetic sampling as requested.
- The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simByEuler` tests the state vector X_t for an all-`NaN` condition. Thus, to signal an early termination of a given trial, all elements of the state vector X_t must be `NaN`. This test enables a user-defined `Processes` function to signal early termination of a trial, and offers significant performance

benefits in some situations (for example, pricing down-and-out barrier options).

Examples

Implementing Multidimensional Equity Market Models, Implementation 5: Using the `simByEuler` Method

See Also

`simBySolution`, `simulate`

Purpose

Simulate approximate solution of diagonal-drift HWV and GBM processes

Syntax

```
[Paths, Times, Z] = OBJ.simBySolution(NPERIODS)
```

```
[Paths, Times, Z] = OBJ.simBySolution(NPERIODS, 'Name1',
Value1,'Name2', Value2, ...)
```

Classes

- GBM
- HWV

Description

The `simBySolution` method simulates `NTRIALS` sample paths of `NVARS` correlated state variables, driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Hull-White/Vasicek (HWV) and geometric Brownian motion (GBM) short-rate models by an approximation of the closed-form solution.

Consider a separable, vector-valued HWV model of the form:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t \tag{15-12}$$

where:

- X is an $NVARS$-by-1 state vector of process variables.
- S is an $NVARS$-by-$NVARS$ matrix of mean reversion speeds (the rate of mean reversion).
- L is an $NVARS$-by-1 vector of mean reversion levels (long-run mean or level).
- V is an $NVARS$-by-$NBROWNS$ instantaneous volatility rate matrix.
- W is an $NBROWNS$-by-1 Brownian motion vector.

or a separable, vector-valued GBM model of the form:

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t \quad \textbf{(15-13)}$$

where:

- X_t is an NVARS-by-1 state vector of process variables.
- μ is an NVARS-by-NVARS generalized expected instantaneous rate of return matrix.
- V is an NVARS-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

The simBySolution method simulates the state vector X_t using an approximation of the closed-form solution of diagonal-drift models.

When evaluating the expressions, simBySolution assumes that all model parameters are piecewise-constant over each simulation period.

In general, this is *not* the exact solution to the models in Equation 5-12 and Equation 15-13, because the probability distributions of the simulated and true state vectors are identical *only* for piecewise-constant parameters.

When parameters are piecewise-constant over each observation period, the simulated process is exact for the observation times at which X_t is sampled.

Input Arguments

OBJ	Hull-White/Vasicek (HWV) or geometric Brownian motion (GBM) model.
NPERIODS	Positive scalar integer number of simulation periods. The value of this argument determines the number of rows of the simulated output series.

Optional Input Arguments

You specify optional input arguments as variable-length lists of matching parameter name/value pairs: 'Name1', Value1, 'Name2', Value2, ... and so on. The following rules apply when specifying parameter-name pairs:

- You specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`NTRIALS`	Positive scalar integer number of simulated trials (sample paths) of `NPERIODS` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.
`DeltaTime`	Scalar or `NPERIODS`-by-1 column vector of positive time increments between observations. `DeltaTime` represents the familiar *dt* found in stochastic differential equations, and determines the times at which `simBySolution` reports the simulated paths of the output state variables. If you do not specify a value for this argument, the default is `1`.
`NSTEPS`	Positive scalar integer number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`). `simBySolution` partitions each time increment *dt* into `NSTEPS` subintervals of length *dt*/`NSTEPS`, and refines the simulation by evaluating the simulated state vector at `NSTEPS - 1` intermediate points. Although `simBySolution` does not report the output state vector at these intermediate points, the refinement improves accuracy by allowing the simulation to more closely approximate the underlying continuous-time process. If you do not specify a value for `NSTEPS`, the default is `1`, indicating no intermediate evaluation.

`Antithetic`	Scalar logical flag that indicates whether antithetic sampling is used to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes). When `Antithetic` is `TRUE` (logical `1`), `simBySolution` performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs: • Odd trials `(1,3,5,...)` correspond to the primary Gaussian paths • Even trials `(2,4,6,...)` are the matching antithetic paths of each pair derived by negating the Gaussian draws of the corresponding primary (odd) trial. If you specify `Antithetic` to be any value other than `TRUE`,`simBySolution` assumes that it is `FALSE` (logical `0`) by default, and does not perform antithetic sampling. When you specify an input noise process (see `Z`), `simBySolution` ignores the value of `Antithetic`.
`Z`	Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. You can specify this argument as a function, or as an `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` array of dependent random variates. If you specify `Z` as a function, it must return an `NBROWNS`-by-1 column vector, and you must call it with two inputs: a real-valued scalar observation time t and an `NVARS`-by-1 state vector X_t. If you do not specify a value for `Z`, `simBySolution` generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object.

`StorePaths`	Scalar logical flag that indicates how `simBySolution` stores the output array `Paths` and returns it to the caller. If `StorePaths` is `TRUE`(the default value) or is unspecified, `simBySolution` returns `Paths` as a 3-dimensional time-series array. If `StorePaths` is `FALSE` (logical `0`), `simBySolution` returns the `Paths` output array as an empty matrix.
`Processes`	Function or cell array of functions that indicates a sequence of end-of-period processes or state vector adjustments of the form $X_t = P(t, X_t)$ `simBySolution` applies processing functions at the end of each observation period. These functions must accept the current observation time t and the current state vector X_t, and return a state vector that may be an adjustment to the input state. If you specify more than one processing function, `simBySolution` invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and more. If you do not specify a processing function, `simBySolution` makes no adjustments and performs no processing.

Output Arguments

`Paths`	`(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` 3-dimensional time-series array, consisting of simulated paths of correlated state variables. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t. When the input flag `StorePaths = FALSE`, `simBySolution` returns `Paths` as an empty matrix.
`Times`	`(NPERIODS + 1)`-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.
`Z`	`(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` 3-dimensional time-series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation.

Remarks

- The input argument `Z` allows you to directly specify the noise generation process. This process takes precedence over the `Correlation` parameter of the `SDE` object and the value of the `Antithetic` input flag. If you do not specify a value for `Z`, `simBySolution` generates correlated Gaussian variates, with or without antithetic sampling as requested.
- Gaussian diffusion models, such as `HWV`, allow negative states. By default, `simBySolution` does nothing to prevent negative states, nor does it guarantee that the model be strictly mean-reverting. Thus, the model may exhibit erratic or explosive growth.
- The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simBySolution` tests the state vector X_t for an all-`NaN` condition. Thus, to signal an early termination of a given trial, all elements of the state vector X_t must be `NaN`. This test enables a user-defined `Processes` function to signal early termination of a trial, and offers significant performance benefits in some situations (for example, pricing down-and-out barrier options).

Examples

Implementing Multidimensional Equity Market Models,
Implementation 6: Using GBM Simulation Methods

See Also

`simByEuler`, `simulate`

simulate

Purpose

Simulate multivariate stochastic differential equations (SDEs)

Syntax

```
[Paths, Times, Z] = SDE.simulate(...)
```

Classes

All classes in the SDE class hierarchy

Description

This method simulates any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \quad \textbf{(15-14)}$$

where:

- X is an *NVARS*-by-1 state vector of process variables (for example, short rates or equity prices) to simulate.
- W is an *NBROWNS*-by-1 Brownian motion vector.
- F is an *NVARS*-by-1 vector-valued drift-rate function.
- G is an *NVARS*-by-*NBROWNS* matrix-valued diffusion-rate function.

`[Paths, Times, Z] = SDE.simulate(...)` simulates `NTRIALS` sample paths of `NVARS` correlated state variables, driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

Input Arguments

`SDE`	Stochastic differential equation model.

Optional Input Arguments

The `simulate` method accepts any variable-length list of input arguments that the simulation method or function referenced by the `SDE.Simulation` parameter requires or accepts. It passes this input list directly to the appropriate SDE simulation method or user-defined simulation function.

Output Arguments

`Paths`	`(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` 3-dimensional time-series array, consisting of simulated paths of correlated state variables. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t.
`Times`	`(NPERIODS + 1)`-by-1 column vector of observation times associated with the simulated paths. Each element of `Times` is associated with a corresponding row of `Paths`.
`Z`	`NTIMES`-by-`NBROWNS`-by-`NTRIALS` 3-dimensional time-series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drove the simulated results found in `Paths`. `NTIMES` is the number of time steps at which `simulate` samples the state vector. `NTIMES` includes intermediate times designed to improve accuracy, which `simulate` does not necessarily report in the `Paths` output time series.

Examples

Antithetic Sampling

Simulation methods allow you to specify a popular *variance reduction* technique called *antithetic sampling*. This technique attempts to replace one sequence of random observations with another of the same expected value, but smaller variance.

In a typical Monte Carlo simulation, each sample path is independent and represents an independent trial. However, antithetic sampling generates sample paths in pairs. The first path of the pair is referred to as the *primary path*, and the second as the *antithetic path*. Any given pair is independent of any other pair, but the two paths within each pair are highly correlated. Antithetic sampling literature often recommends averaging the discounted payoffs of each pair, effectively halving the number of Monte Carlo trials.

This technique attempts to reduce variance by inducing negative dependence between paired input samples, ideally resulting in negative dependence between paired output samples. The greater the extent of negative dependence, the more effective antithetic sampling is.

This example applies antithetic sampling to a path-dependent barrier option. Consider a European up-and-in call option on a single underlying stock. The evolution of this stock's price is governed by a Geometric Brownian Motion (GBM) model with constant parameters:

$$dX_t = 0.05X_t dt + 0.3X_t dW_t$$

Assume the following characteristics:

- The stock currently trades at 105.
- The stock pays no dividends.
- The stock volatility is 30% per annum.
- The option strike price is 100.
- The option expires in 3 months.
- The option barrier is 120.
- The risk-free rate is constant at 5% per annum.

The goal is to simulate various paths of daily stock prices, and calculate the price of the barrier option as the risk-neutral sample average of the discounted terminal option payoff. Since this is a barrier option, you must also determine if and when the barrier is crossed.

This example performs antithetic sampling by explicitly setting the `Antithetic` flag to `true`, and then specifies an end-of-period processing function to record the maximum and terminal stock prices on a path-by-path basis.

1 Create a `GBM` model:

```
barrier   = 120;          % barrier
strike    = 100;          % exercise price
rate      = 0.05;         % annualized risk-free rate
sigma     = 0.3;          % annualized volatility
nPeriods  = 63;           % 63 trading days
dt        = 1 / 252;      % time increment = 252 days
```

```
T         = nPeriods * dt;  % expiration time = 0.25 years
obj      = gbm(rate, sigma, 'StartState', 105);
```

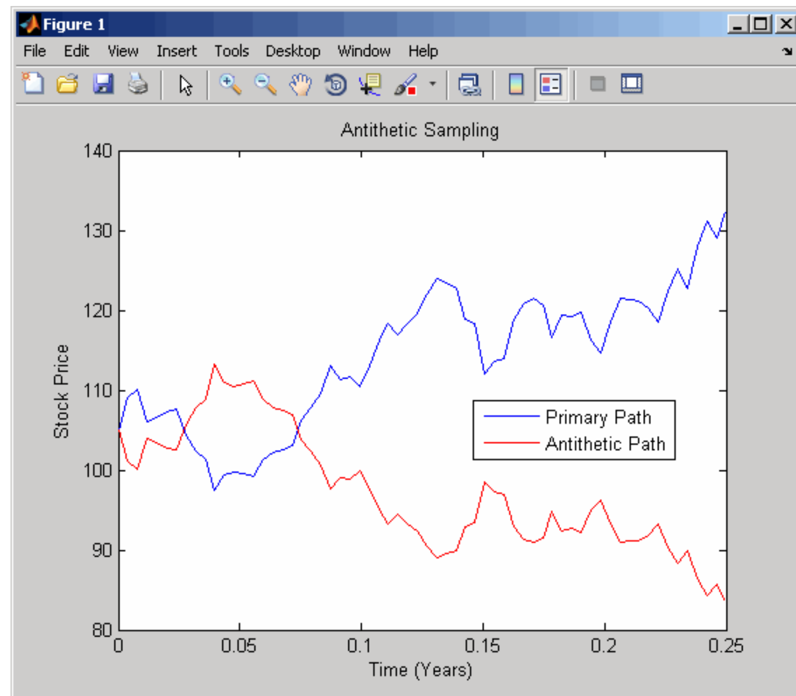
2 Perform a very small-scale simulation that explicitly returns two simulated paths:

```
randn('state', 10)
[X, T] = obj.simBySolution(nPeriods, 'DeltaTime', dt, ...
    'nTrials', 2, 'Antithetic', true);
```

3 Perform antithetic sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs. Odd paths (1,3,5,...) correspond to the primary Gaussian paths. Even paths (2,4,6,...) are the matching antithetic paths of each pair, derived by negating the Gaussian draws of the corresponding primary (odd) path.

Verify this by examining the matching paths of the primary/antithetic pair:

```
plot(T, X(:,:,1), 'blue', T, X(:,:,2), 'red')
xlabel('Time (Years)'), ylabel('Stock Price'), ...
   title('Antithetic Sampling')
legend({'Primary Path' 'Antithetic Path'}, ...
   'Location', 'Best')
```



To price the European barrier option, specify an end-of-period processing function to record the maximum and terminal stock prices. This processing function is accessible by time and state, and is implemented as a nested function with access to shared information that allows the option price and corresponding standard error to be calculated.

1 Simulate 200 paths using the processing function method:

```
randn('state', 10)
nPaths = 200;          % # of paths = 100 sets of pairs
f      = barrierExample(nPeriods, nPaths)
obj.simBySolution(nPeriods, 'DeltaTime' , dt, ...
   'nTrials', nPaths, 'Antithetic', true, ...
   'Processes', f.SaveMaxLast);
```

2 Approximate the option price with a 95% confidence interval:

```
optionPrice   = f.OptionPrice  (strike, rate, barrier);
standardError = f.StandardError(strike, rate, barrier, ...
       true);
lowerBound    = optionPrice - 1.96 * standardError;
upperBound    = optionPrice + 1.96 * standardError;

fprintf('  Up-and-In Barrier Option Price: %8.4f\n', ...
   optionPrice)
fprintf('         Standard Error of Price: %8.4f\n', ...
   standardError)
fprintf(' Confidence Interval Lower Bound: %8.4f\n', ...
   lowerBound)
fprintf(' Confidence Interval Upper Bound: %8.4f\n', ...
   upperBound)

Up-and-In Barrier Option Price:    7.4549
Standard Error of Price:    0.6763
Confidence Interval Lower Bound:   6.1294
Confidence Interval Upper Bound:   8.7805
```

See Also

`simByEuler`, `simBySolution`

Purpose

Convert time-series arrays to callable functions of time and state

Syntax

```
F = ts2func(Array)
F = ts2func(Array, 'Name1', Value1, 'Name2', Value2, ...)
```

Description

The `ts2func` function encapsulates a time-series array associated with a vector of real-valued observation times within a MATLAB® function suitable for Monte Carlo simulation of an `NVARS`-by-1 state vector X_t.

Input Arguments

`Array`	Time-series array to encapsulate within a callable function of time and state. `Array` may be a vector, 2-dimensional matrix, or 3-dimensional array.

Optional Input Arguments

You specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- You specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Valid parameter names are as follows:

`Times`	Vector of monotonically increasing observation times associated with the time-series input array `Array`. If you do not specify a value for this argument, `Times` is a zero-based, unit-increment vector of the same length as that of the dimension of `Array` associated with time (see `TimeDimension`).
`TimeDimension`	Positive scalar integer that specifies which dimension of the input time-series array `Array` is associated with time. The value of this argument cannot be greater than the number of dimensions of `Array`. If you do not specify a value for this argument, the default is `1`, indicating that time is associated with the rows of `Array`.
`StateDimension`	Positive scalar integer that specifies which dimension of the input time-series array `Array` is associated with the `NVARS` state variables. `StateDimension` cannot be greater than the number of dimensions of `Array`. If you do not specify a value for this argument, `ts2func` assigns `StateDimension` the first dimension of `Array` that is not already associated with time (the state vector X_t is associated with the first available dimension of `Array` *not* already assigned to `TimeDimension`).

Output Arguments

`F`	Callable function *F(t)* of a real-valued scalar observation time *t*. You can invoke `F` with a second input (such as an `NVARS`-by-1 state vector *X*), which is a placeholder that `ts2func` ignores. For example, while *F(t)* and *F(t,X)* produce identical results, the latter directly supports SDE simulation methods.

Remarks

- When you specify `Array` as a trivial scalar or a vector (row or column), `ts2func` assumes that it represents a univariate time series.

- `F` returns an array with one fewer dimension than the input time-series array `Array` with which `F` is associated. Thus, when `Array` is a vector, a 2-dimensional matrix, or a 3-dimensional array, `F` returns a scalar, vector, or 2-dimensional matrix, respectively.
- When the scalar time t at which `ts2func` evaluates the function `F` does not coincide with an observation time in `Times`, `F` performs a zero-order-hold interpolation. The only exception is if t precedes the first element of `Times`, in which case $F(t) = F(Times(1))$.
- To support Monte Carlo simulation methods, the output function `F` returns an `NVARS`-by-1 column vector or a 2-dimensional matrix with `NVARS` rows.

See Also

`simByEuler`, `simulate`

A

Bibliography

[1] Ait-Sahalia, Y., "Testing Continuous-Time Models of the Spot Interest Rate", *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385-426.

[2] Ait-Sahalia, Y., "Transition Densities for Interest Rate and Other Nonlinear Diffusions". *The Journal of Finance*, Vol. LIV, No. 4, August 1999.

[3] Baillie, R.T., and T. Bollerslev, "Prediction in Dynamic Models with Time-Dependent Conditional Variances," *Journal of GARCH*, Vol. 52, 1992, pp. 91–113.

[4] Bera, A.K., and H.L. Higgins, "A Survey of ARCH Models: Properties, Estimation and Testing," *Journal of Economic Surveys*, Vol. 7, No. 4, 1993.

[5] Bollerslev, T., "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return," *Review of Economics and Statistics*, Vol. 69, 1987, pp. 542–547.

[6] Bollerslev, T., "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of GARCH*, Vol. 31, 1986, pp. 307–327.

[7] Bollerslev, T., R.Y. Chou, and K.F. Kroner, "ARCH Modeling in Finance: A Review of the Theory and Empirical Evidence," *Journal of GARCH*, Vol. 52, 1992, pp. 5–59.

[8] Bollerslev, T., R.F. Engle, and D.B. Nelson, "ARCH Models," *Handbook of GARCH*, Volume IV, Chapter 49, pp. 2959–3038, Elsevier Science B.V., Amsterdam, The Netherlands, 1994.

[9] Bollerslev, T., and E. Ghysels, “Periodic Autoregressive Conditional Heteroscedasticity,” *Journal of Business and Economic Statistics*, Vol. 14, 1996, pp. 139–151.

[10] Box, G.E.P., G.M. Jenkins, and G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, Third edition, Prentice Hall, Upper Saddle River, NJ, 1994.

[11] Brooks, C., S.P. Burke, and G. Persand, “Benchmarks and the Accuracy of GARCH Model Estimation,” *International Journal of Forecasting*, Vol. 17, 2001, pp. 45–56.

[12] Campbell, J.Y., A.W. Lo, and A.C. MacKinlay, “The GARCH of Financial Markets,” *Nonlinearities in Financial Data*, Chapter 12, Princeton University Press, Princeton, NJ, 1997.

[13] Enders, W., *Applied Econometric Time Series*, John Wiley & Sons, New York, 1995.

[14] Engle, Robert F., “Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation,” *Econometrica*, Vol. 50, 1982, pp. 987–1007.

[15] Engle, Robert F., D.M. Lilien, and R.P. Robins, “Estimating Time Varying Risk Premia in the Term Structure: The ARCH-M Model,” *Econometrica*, Vol. 59, 1987, pp. 391–407.

[16] Glasserman, P., *Monte Carlo Methods in Financial Engineering*, Springer-Verlag, 2004.

[17] Glosten, L.R., R. Jagannathan, and D.E. Runkle, “On the Relation between Expected Value and the Volatility of the Nominal Excess Return on Stocks,” *The Journal of Finance*, Vol. 48, 1993, pp. 1779–1801.

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

[19] Greene, W.H., *Econometric Analysis*, Fifth edition, Prentice Hall, Upper Saddle River, NJ, 2003.

[20] Hagerud, G.E., “Modeling Nordic Stock Returns with Asymmetric GARCH,” *Working Paper Series in Economics and Finance*, No. 164, Department of Finance, Stockholm School of Economics, 1997.

[21] Hagerud, G.E., “Specification Tests for Asymmetric GARCH,” *Working Paper Series in Economics and Finance*, No. 163, Department of Finance, Stockholm School of Economics, 1997.

[22] Hamilton, J.D., *Time Series Analysis*, Princeton University Press, Princeton, NJ, 1994.

[23] Hentschel, L., “All in the Family: Nesting Symmetric and Asymmetric GARCH Models,” *Journal of Financial Economics*, Vol. 39, 1995, pp. 71–104.

[24] Hull, J.C., *Options, Futures, and Other Derivatives*, 5th Edition, Prentice Hall 2002.

[25] Johnson, N.L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, Second edition, John Wiley & Sons, New York, 1995.

[26] Longstaff, F.A., Schwartz, E.S., “Valuing American Options by Simulation: A Simple Least-Squares Approach”, *The Review of Financial Studies*, Spring 2001, Vol. 14, No. 1, pp. 113–147.

[27] McCullough, B.D., and C.G. Renfro, “Benchmarks and Software Standards: A Case Study of GARCH Procedures,” *Journal of Economic and Social Measurement*, Vol. 25, 1998, pp. 59–71.

[28] Nelson, D.B., “Conditional Heteroskedasticity in Asset Returns: A New Approach,” *Econometrica*, Vol. 59, 1991, pp. 347–370.

[29] Peters, J.P., “Estimating and Forecasting Volatility of Stock Indices Using Asymmetric GARCH Models and Skewed Student-t Densities,” working paper, École d’Administration des Affaires, University of Liège, Belgium, March 20, 2001.

[30] Shreve, S.E., *Stochastic Calculus for Finance II: Continuous-Time Models,* Springer-Verlag, 2004.

Examples

Use this list to find examples in the documentation.

Introduction

"Example: Analysis and Estimation Using the Default Model" on page 2-16
"Example: Interpreting Specification Structures" on page 3-6

Simulation

"Simulating Single Paths" on page 4-3
"Simulating Multiple Paths" on page 4-5
"Specifying a Scalar Response Tolerance" on page 4-8

Simulating Univariate Brownian Motion Models

"Example: Creating Base SDE Models" on page 5-14
"Example: Creating Drift and Diffusion Rate Objects as Model Inputs" on page 5-17
"Example: Creating SDEDDO Models" on page 5-19
"Example: Creating SDELD Models" on page 5-20
"Example: Creating BM Models" on page 5-21
"Example: Creating Univariate CEV Models" on page 5-22
"Example: Creating Univariate GBM Models" on page 5-24
"Example: Creating SDEMRD Models" on page 5-25
"Example: Creating CIR Models" on page 5-26
"Example: Creating HWV Models" on page 5-27

Monte Carlo Simulation of Stochastic Differential Equations

"Implementing Multidimensional Equity Market Models" on page 5-29
"Stochastic Interpolation and the Brownian Bridge" on page 5-42
"Inducing Dependence and Correlation" on page 5-48
"Incorporating Dynamic Behavior" on page 5-51
"Ensuring Positive State Variables" on page 5-57
"Black-Scholes Option Pricing" on page 5-60

"User-Specified Random Number Generation: Stratified Sampling" on page 5-63
"Example: Improving SDE Solution Accuracy by Increasing Sampling of the Discrete-Time Process" on page 5-76
"Stochastic Interpolation Without Refinement" on page 15-41
"Monte Carlo Simulation of Conditional Gaussian Distributions" on page 15-44
"Antithetic Sampling" on page 15-83

Estimation

"Specifying Presample Data" on page 6-21
"Presample Data and Transient Effects" on page 6-24
"Alternative Technique for Estimating ARMA(R,M) Parameters" on page 6-30
"Active Lower Bound Constraint" on page 6-30
"Determining Convergence Status" on page 6-34

Forecasting

"Forecasting Using `garchpred`" on page 7-9
"Volatility Forecasting over Multiple Periods" on page 7-12
"Forecasting with Multiple Realizations" on page 7-15

Regression

"Fitting a Model to a Simulated Return Series" on page 8-3
"Fitting a Regression Model to the Same Return Series" on page 8-5
"Ordinary Least Squares Regression" on page 8-11

Unit Root Tests

"Testing GDP by OLS Regression with a Stationary Component" on page 9-14
"Testing T-Bill Rate by OLS Regression with a Drift Component" on page 9-17

Model Selection and Analysis

"Likelihood Ratio Tests" on page 10-3
"Akaike and Bayesian Information Criteria" on page 10-6
"Equality Constraints and Parameter Significance" on page 10-9
"Equality Constraints and Initial Parameter Estimates" on page 10-14
"Examples: Simplicity and Parsimony" on page 10-17

Example Workflow: Estimation, Forecasting, and Monte Carlo Simulation

"Estimating the Model" on page 11-3
"Forecasting" on page 11-5
"Forecasting Using Monte Carlo Simulation" on page 11-7
"Comparing Forecasts with Simulation Results" on page 11-9

Glossary

Akaike information criteria (AIC)

A model-order selection criterion based on parsimony. More complicated models are penalized for the inclusion of additional parameters. *See also* **Bayesian information criteria (BIC)**.

antithetic sampling

A variance reduction technique that pairs a sequence of independent normal random numbers with a second sequence obtained by negating the random numbers of the first. The first sequence simulates increments of one path of Brownian motion, and the second sequence simulates increments of its reflected, or antithetic, path. These two paths form an antithetic pair independent of any other pair.

AR

Autoregressive. AR models include past observations of the dependent variable in the forecast of future observations.

ARCH

Autoregressive Conditional Heteroscedasticity. A time-series technique that uses past observations of the variance to forecast future variances. *See also* **GARCH**.

ARMA

Autoregressive Moving Average. A time-series model that includes both AR and MA components. *See also* **AR** and **MA**.

autocorrelation function (ACF)

Correlation sequence of a random time series with itself. See also **cross-correlation function (XCF)**.

autoregressive

See **AR**.

Bayesian information criteria (BIC)

A model-order selection criterion 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)**.

Brownian motion
A zero-mean continuous-time stochastic process with independent increments (also known as a *Wiener process*).

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* **autocorrelation function (ACF)**.

diffusion
The function that characterizes the random (stochastic) portion of a stochastic differential equation. *See also* **stochastic differential equation**.

discretization error
Errors that may arise due to discrete-time sampling of continuous stochastic processes.

drift
The function that characterizes the deterministic portion of a stochastic differential equation. *See also* **stochastic differential equation**.

equality constraint
A constraint, imposed during parameter estimation, by which a parameter is held fixed at a user-specified value.

Euler approximation
A simulation technique that provides a discrete-time approximation of a continuous-time stochastic process.

excess kurtosis
A characteristic, relative to a standard normal probability distribution, in which 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**.

GARCH
Generalized autoregressive conditional heteroscedasticity. A time-series technique that uses past observations of the variance and variance forecast to forecast future variances. *See also* **ARCH**.

heteroscedasticity
Time-varying, or time-dependent, variance.

homoscedasticity
Time-independent variance. The GARCH Toolbox™ software also refers to homoscedasticity as constant conditional variance.

i.i.d.
Independent, identically distributed.

innovations
A sequence of unanticipated shocks, or disturbances. The GARCH Toolbox™ software 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. A technique designed to minimize the variance of the estimation or forecast error. *See also* **RMSE**.

moving average

See **MA**.

objective function

The function to numerically optimize. In the GARCH Toolbox™ software, the objective function is the log-likelihood function of a random process.

partial autocorrelation function (PACF)

Correlation sequence estimated by fitting successive order autoregressive models to a random time series by least squares. The PACF is useful for identifying the order of an autoregressive model.

path

A random trial of a time-series process.

proportional sampling

A stratified sampling technique that ensures that the proportion of random draws matches its theoretical probability. One of the most common examples of proportional sampling involves stratifying the terminal value of a price process in which each sample path is associated with a single stratified terminal value such that the number of paths equals the number of strata.

See also **stratified sampling**.

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.

stochastic differential equation
A generalization of an ordinary differential equation, with the addition of a noise process, that yields random variables as solutions.

strata
See **stratified sampling**.

stratified sampling
A variance reduction technique that constrains a proportion of sample paths to specific subsets (or *strata*) of the sample space.

time series
Discrete-time sequence of observations of a random process. The type of time series of interest in the GARCH Toolbox™ software 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.

trial
The result of an independent random experiment that computes the average or expected value of a variable of interest and its associated confidence interval.

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.

variance reduction

A sampling technique in which a given sequence of random variables is replaced with another of the same expected value but smaller variance. Variance reduction techniques increase the efficiency of Monte Carlo simulation.

volatility

The risk, or uncertainty, measure associated with a financial time series. The GARCH Toolbox™ software associates volatility with standard deviation.

Wiener process

See **Brownian motion**.

Index

A

ACF 13-7
AIC
  model selection 10-6
`aicbic` 13-2
Akaike information criteria
  model selection 10-6
analysis example
  using default model 2-16
AR model
  converting from ARMA model 13-28
ARCH/GARCH effects
  hypothesis test 13-4
`archtest` 13-4
ARMA model
  converting to AR model 13-28
  converting to MA model 13-54
array size 1-7
arrays
  time series 1-7
asymptotic behavior
  for long-range forecast horizons 7-7
  long-range forecasts 7-7
`autocorr` 13-7
autocorrelation function 13-7
autoregressive model
  converting from ARMA model 13-28

B

Bayesian information criteria
  model selection 10-6
BIC
  model selection 10-6
bm 15-2

C

cev 15-7
cir 15-13
compounding
  continuous and periodic 1-8
conditional mean models 2-9
  regression components 8-2 9-2
conditional standard deviations
  inferred from return series 13-48
  of forecast errors 13-61
  simulating 13-76
conditional variance models 2-10
conditional variances
  constant 8-11
constraints
  active lower bound example 6-30
  equality 10-9
  fixing model parameters 10-9
  tolerance limits 6-18
conventions
  technical 1-7
convergence
  avoiding problems 2-16
  determining status 6-34
  showing little progress 2-16
  suboptimal solution 2-16
  tolerance options 6-16
cross-correlation function 13-12
`crosscorr` 13-12

D

data sets 1-12
  Deutschmark/British Pound FX price series 1-12
  NASDAQ Composite Index 1-13
  New York Stock Exchange Composite Index 1-13
default
  GARCH model 2-13
default model
  estimation and analysis example 2-16
  estimation example 2-16

Deutschmark/British Pound FX price series 1-12
`dfARDTest` 13-16
`dfARTest` 13-20
`dfTSTest` 13-24
diffusion 15-19
distributions
  supported 2-5
drift 15-23

E

EGARCH(P,Q) conditional variance model 2-11
Engle's hypothesis test 13-4
equality constraints
  initial parameter estimates 10-14
  parameter significance 10-9
estimation 6-1
  advanced example 11-3
  control of optimization process 6-15
  convergence 6-16
  convergence to suboptimal solution 2-16
  count of coefficients 13-32
  incorporating a regression model 8-3
  initial parameter estimates 6-4
  maximum likelihood 6-2
  number of function evaluations 6-15
  number of iterations 6-15
  of GARCH process 13-35
  optimization results 6-17
  parameter bounds 6-10
  plotting results 13-58
  premature termination 2-16
  presample observations 6-12
  summary information 13-39
  termination criteria 6-15
  tolerance options 6-16
estimation example
  estimating model parameters 2-24
  post-estimation analysis 2-27
  pre-estimation analysis 2-16
  using default model 2-16

F

fat tails 2-2
filtering 13-85
financial time series
  characteristics 2-2
  modeling 2-2
fixing model constraints 10-9
forecast errors
  conditional standard deviations 13-61
forecast results
  compare to simulation results 11-9
forecasted explanatory data 8-9
forecasting 7-1
  advanced example 11-5
  asymptotic behavior for long-range 7-7
  basic example 7-9
  conditional mean of returns 7-3
  conditional standard deviations of innovations 7-2
  minimum mean square error 7-2
  multiperiod volatility example 7-12
  multiple realizations example 7-15
  plotting results 13-58
  presample data 7-6
  RMSE of mean forecast 7-4
  using a regression model 8-9
  volatility of returns 7-3
function evaluation count
  maximum 6-15
functions
  `aicbic` 13-2
  `archtest` 13-4
  `autocorr` 13-7
  converting from time series 15-88
  `crosscorr` 13-12
  `dfARDtest` 13-16
  `dfARtest` 13-20

dfTStest 13-24
example showing relationships 11-1
garchar 13-28
garchcount 13-32
garchdisp 13-33
garchfit 13-35
garchget 13-46
garchinfer 13-48
garchlot 13-58
garchma 13-54
garchpred 13-61
garchset 13-68
garchsim 13-76
hpfilter 13-85
lagmatrix 13-88
lbqtest 13-91
listed by category 12-1
lratiotest 13-94
parcorr 13-97
ppARDTest 13-102
ppARTest 13-105
ppTSTest 13-108
price2ret 13-112
primary engines 2-14
ret2price 13-116

G

GARCH
brief description 1-3
limitations 1-4
uses for 1-3
GARCH process
forecasting 13-61
inferring innovations 13-48
parameter estimation 13-35
count of coefficients 13-32
displaying results 13-33
simulation 13-76
GARCH specification structure
contents
interpreting 3-6
creating and modifying parameters 3-9
definition of fields 13-69
retrieving parameters 13-46
GARCH Toolbox
conventions and clarifications
compounding 1-8
primary functions 2-14
GARCH(P,Q) conditional variance model 2-10
garchar 13-28
garchcount 13-32
garchdisp 13-33
garchfit 13-35
garchget 13-46
garchinfer 13-48
garchma 13-54
garchplot 13-58
garchpred 13-61
garchset 13-68
garchsim 13-76
gbm 15-26
GJR(P,Q) conditional variance model 2-11

H

Hodrick-Prescott filter 13-85
hpfilter 13-85
hwv 15-31
hypothesis tests
likelihood ratio 13-94
Ljung-Box lack-of-fit 13-91

I

inference
conditional standard deviations 13-48
GARCH innovations 13-48
transient effects example 6-24
using a regression model 8-8

initial parameter estimates 6-4
conditional mean models with regression 6-7
conditional mean models without regression 6-6
conditional variance models 6-7
equality constraints 10-14
innovations
distribution 2-5
forecasting conditional standard deviations 7-2
inferred from return series 13-48
serial dependence 2-5
simulating 13-76
`interpolate` 15-37
iteration count
maximum 6-15

L

lack-of-fit hypothesis test 13-91
lagged time-series matrix 13-88
`lagmatrix` 13-88
`lbqtest` 13-91
length
vector 1-7
leverage effects 2-2
likelihood ratio hypothesis test 13-94
likelihood ratio tests
model selection 10-3
Ljung-Box lack-of-fit hypothesis test 13-91
log-likelihood functions 6-2
optimized value parameters 13-35
long-range forecasting
asymptotic behavior 7-7
`lratiotest` 13-94

M

MA model
converting from ARMA model 13-54
maximum likelihood
estimation 6-2
methods
listed by category 14-1
minimum mean square error
forecasting 7-2
MMSE
forecasting 7-2
model parameters
complete specification 10-14
empty fix fields 10-15
equality constraints 10-9
estimating 2-24
fixing 10-9
parsimony 10-17
model selection and analysis 10-1
AIC and BIC 10-6
correlation in return series 2-19
correlation in squared returns 2-21
Engle's ARCH test 2-24
likelihood ratio tests 10-3
Ljung-Box-Pierce Q-test 2-23
modeling
financial time series 2-2
models
complete specification 10-14
conditional mean and variance 2-7
GARCH default 2-13
Monte Carlo simulation 8-13
advanced example 11-7
compare to forecast results 11-9
moving average model
converting from ARMA model 13-54

N

NASDAQ Composite Index 1-13
New York Stock Exchange Composite Index 1-13
non-stationary time series 1-9
NYSE Composite Index 1-13

O

ordinary least squares regression 8-11

P

PACF 13-97
parameter estimates
  bounds 6-10
  displaying results 13-33
  equality constraints 10-14
  initial 6-4
    automatically generated 6-6
    user-specified 6-4
parameter estimation
  plotting results 13-58
  univariate GARCH process 13-35
`parcorr` 13-97
parsimonious parameterization 10-17
partial autocorrelation function 13-97
plotting
  autocorrelation function 13-7
  cross-correlation function 13-12
  forecasted results 13-58
  parameter estimation results 13-58
  partial autocorrelation function 13-97
  simulation results 13-58
`ppARDTest` 13-102
`ppARTest` 13-105
`ppTSTest` 13-108
precision 1-8
presample data
  estimation
    automatically generated 6-13
    deriving from actual data 6-30
    example 6-21
    user-specified 6-12
  forecasting 7-6
  simulation
    automatically generated 4-7
    user-specified 4-13
price series
  converting from return series 13-116
  converting to return series 13-112
`price2ret` 13-112

R

regression
  in Monte Carlo framework 8-13
regression components
  conditional mean models 8-2 9-2
  estimation 8-3
  forecasting 8-9
  inference 8-8
  simulation 8-8
response tolerance
  for simulated data 4-8
`ret2price` 13-116
return series
  converting from price series 13-112
  converting to price series 13-116
  forecasting conditional mean 7-3
  forecasting RMSE of mean forecast 7-4
  forecasting volatility 7-3
  simulating 13-76

S

sde 15-47
SDE class constructors
  `bm` 15-2
  `cev` 15-7
  `cir` 15-13
  `diffusion` 15-19
  `drift` 15-23
  `gbm` 15-26
  `hwv` 15-31
  `sde` 15-47
  `sdeddo` 15-52
  `sdeld` 15-57

`sdemrd` 15-63
SDE simulation methods
`interpolate` 15-37
`simByEuler` 15-69
`simBySolution` 15-75
`simulate` 15-82
sdeddo 15-52
sdeld 15-57
sdemrd 15-63
SDEs
about 5-2
class hierarchy 5-11
creating user-specified functions 5-69
improving performance 5-73
managing memory consumption 5-72
objects
behavior 5-5
relationship to models 5-5
syntax 5-5
optimizing solution accuracy 5-74
parametric specification 5-7
popular models 5-9
solving problems with SDE models 5-29
Black-Sholes option pricing 5-60
ensuring positive state processes 5-57
implementing multidimensional market models 5-29
incorporating dynamic behavior 5-51
inducing dependence and correlation 5-48
stochastic interpolation and the brownian bridge 5-42
user-specified random number generation: stratified sampling 5-63
terminology 5-3
using objects to create models 5-11
creating base SDE objects 5-14
creating brownian motion (BM) models 5-21
creating constant elasticity of variance (CEV) models 5-22
creating Cox-Ingersoll-Ross (CIR) square root diffusion models 5-25
creating drift and diffusion objects 5-16
creating geometric brownian motion (GBM) models 5-23
creating Hull-White/Vasicek (HWV) gaussian diffusion models 5-26
creating SDEs from drift and diffusion objects (SDEDDO) 5-19
creating SDEs from linear drift (SDELD) 5-20
creating SDEs from mean-reverting drift (SDEMRD) 5-24
shifted time-series matrix 13-88
`simByEuler` 15-69
`simBySolution` 15-75
`simulate` 15-82
simulation 4-1
compare to forecast results 11-9
plotting results 13-58
presample data 4-7
response tolerance 4-8
sample paths 4-2
storage considerations 4-10
univariate GARCH processes 13-76
using a regression model 8-8
using ordinary least squares regression 8-11
size
array and vector 1-7
specification structure
contents
interpreting 3-6
creating and modifying parameters 3-9
definition of fields 13-69

fixing model parameters 10-9
retrieving parameters 13-46
stationary time series 1-9
Stochastic Differential Equations. *See* SDEs

T

termination criteria
estimation 6-15
time series
characteristics of financial 2-2
converting to functions 15-88
correlation of observations 2-5
cyclical component 13-85
modeling financial 2-2
stationary and non-stationary 1-9
stationary, non-stationary 1-9
trend component 13-85
time-series matrix 1-7
lagged or shifted 13-88
tolerance options 6-16
constraint violation 6-18
effect on convergence 6-17
effect on optimization results 6-17
transients
automatic minimization 4-7
in presample simulation data 4-7
inference example 6-24
minimization techniques 4-11
simulation process 4-7
`ts2func` 15-88

V

vector length 1-7
vector size 1-7
volatility
forecasting 7-3
forecasting example 7-12
volatility clustering 2-2

X

XCF 13-12