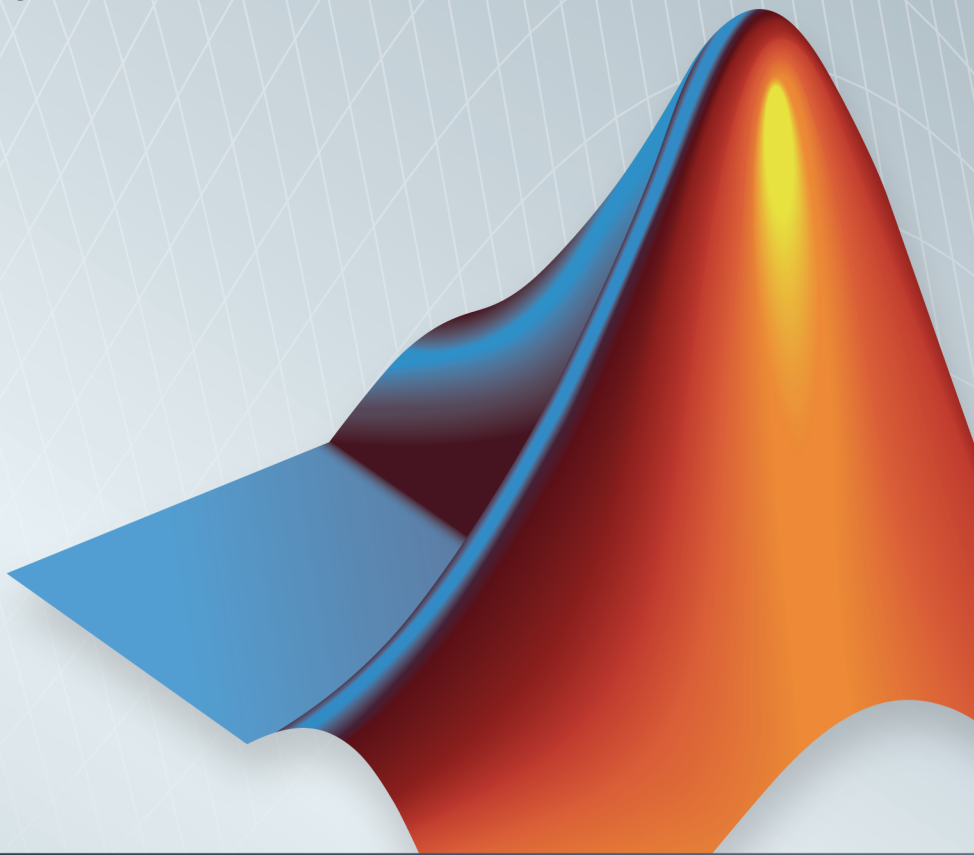


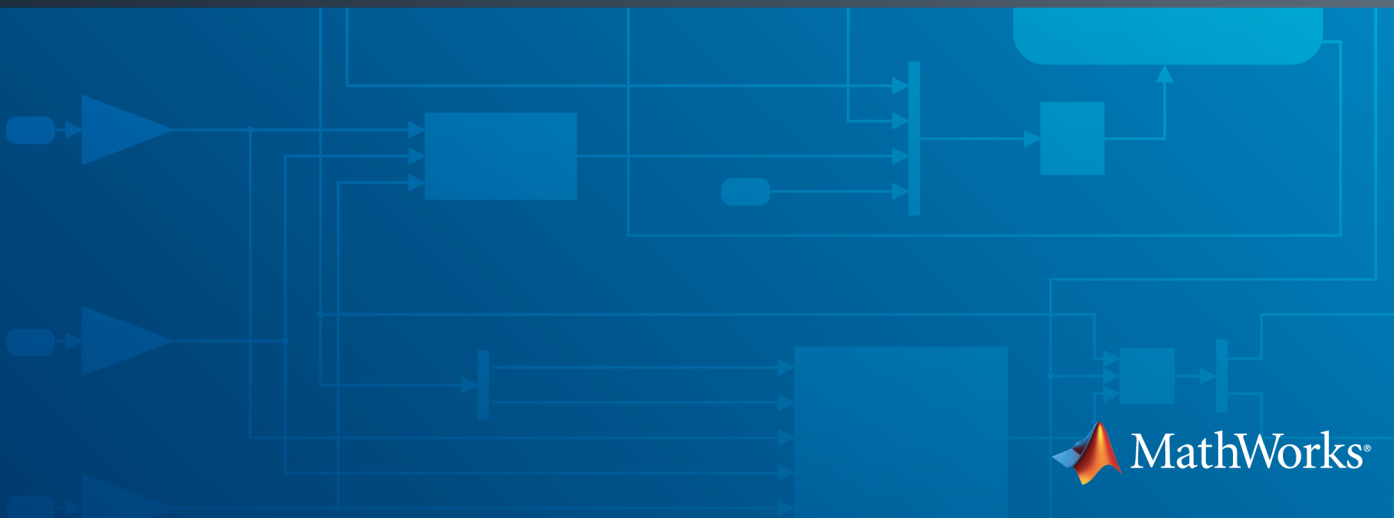
Financial Toolbox™

User's Guide

R2014b



MATLAB®



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Financial Toolbox™ User's Guide

© COPYRIGHT 1995–2014 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

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

Revision History

October 1995	First printing	
January 1998	Second printing	Revised for Version 1.1
January 1999	Third printing	Revised for Version 2.0 (Release 11)
November 2000	Fourth printing	Revised for Version 2.1.2 (Release 12)
May 2003	Online only	Revised for Version 2.3 (Release 13)
June 2004	Online only	Revised for Version 2.4 (Release 14)
August 2004	Online only	Revised for Version 2.4.1 (Release 14+)
September 2005	Fifth printing	Revised for Version 2.5 (Release 14SP3)
March 2006	Online only	Revised for Version 3.0 (Release 2006a)
September 2006	Sixth printing	Revised for Version 3.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.4 (Release 2008a)
October 2008	Online only	Revised for Version 3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.6 (Release 2009a)
September 2009	Online only	Revised for Version 3.7 (Release 2009b)
March 2010	Online only	Revised for Version 3.7.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.8 (Release 2010b)
April 2011	Online only	Revised for Version 4.0 (Release 2011a)
September 2011	Online only	Revised for Version 4.1 (Release 2011b)
March 2012	Online only	Revised for Version 4.2 (Release 2012a)
September 2012	Online only	Revised for Version 5.0 (Release 2012b)
March 2013	Online only	Revised for Version 5.1 (Release 2013a)
September 2013	Online only	Revised for Version 5.2 (Release 2013b)
March 2014	Online only	Revised for Version 5.3 (Release 2014a)
October 2014	Online only	Revised for Version 5.4 (Release 2014b)

1	Getting Started	
	Financial Toolbox Product Description	1-2
	Key Features	1-2
	Expected Users	1-3
	Analyze Sets of Numbers Using Matrix Functions	1-4
	Introduction	1-4
	Key Definitions	1-4
	Referencing Matrix Elements	1-5
	Transposing Matrices	1-6
	Matrix Algebra Refresher	1-7
	Introduction	1-7
	Adding and Subtracting Matrices	1-7
	Multiplying Matrices	1-8
	Dividing Matrices	1-12
	Solving Simultaneous Linear Equations	1-13
	Operating Element by Element	1-16
	Using Input and Output Arguments with Functions	1-17
	Input Arguments	1-17
	Output Arguments	1-19
	Interest Rate Arguments	1-20

2	Performing Common Financial Tasks	
	Introduction	2-2

Handle and Convert Dates	2-4
Date Formats	2-4
Date Conversions	2-5
Current Date and Time	2-8
Determining Dates	2-8
Format Currency	2-11
Charting Financial Data	2-12
Introduction	2-12
High-Low-Close Chart Example	2-13
Bollinger Chart Example	2-14
Analyzing and Computing Cash Flows	2-16
Introduction	2-16
Interest Rates/Rates of Return	2-16
Present or Future Values	2-17
Depreciation	2-18
Annuities	2-18
Pricing and Computing Yields for Fixed-Income Securities	2-20
Introduction	2-20
Fixed-Income Terminology	2-20
Framework	2-24
Default Parameter Values	2-25
Coupon Date Calculations	2-27
Yield Conventions	2-28
Pricing Functions	2-28
Yield Functions	2-29
Fixed-Income Sensitivities	2-30
Treasury Bills Defined	2-33
Computing Treasury Bill Price and Yield	2-34
Introduction	2-34
Treasury Bill Repurchase Agreements	2-34
Treasury Bill Yields	2-36
Term Structure of Interest Rates	2-38
Introduction	2-38
Deriving an Implied Zero Curve	2-39

Pricing and Analyzing Equity Derivatives	2-41
Introduction	2-41
Sensitivity Measures	2-41
Analysis Models	2-42

Portfolio Analysis

3

Analyzing Portfolios	3-2
Portfolio Optimization Functions	3-3
Portfolio Construction Examples	3-5
Introduction	3-5
Efficient Frontier Example	3-5
Portfolio Selection and Risk Aversion	3-8
Introduction	3-8
Optimal Risky Portfolio	3-9
Constraint Specification	3-12
Example	3-12
Linear Constraint Equations	3-14
Specifying Additional Constraints	3-17
Active Returns and Tracking Error Efficient Frontier	3-19

Mean-Variance Portfolio Optimization Tools

4

Portfolio Optimization Theory	4-2
Portfolio Optimization Problems	4-2
Portfolio Problem Specification	4-2
Return Proxy	4-3
Risk Proxy	4-5
Portfolio Set for Portfolio Optimization	4-7
Default Portfolio Problem	4-14

Portfolio Object Workflow	4-16
Portfolio Object	4-17
Portfolio Object Properties and Methods	4-17
Working with Portfolio Objects	4-21
Setting and Getting Properties	4-22
Displaying Portfolio Objects	4-22
Saving and Loading Portfolio Objects	4-22
Estimating Efficient Portfolios and Frontiers	4-23
Arrays of Portfolio Objects	4-23
Subclassing Portfolio Objects	4-24
Conventions for Representation of Data	4-24
Constructing the Portfolio Object	4-26
Syntax	4-26
Portfolio Problem Sufficiency	4-27
Constructor Examples	4-27
Common Operations on the Portfolio Object	4-34
Naming a Portfolio Object	4-34
Configuring the Assets in the Asset Universe	4-34
Setting Up a List of Asset Identifiers	4-35
Truncating and Padding Asset Lists	4-36
Setting Up an Initial or Current Portfolio	4-37
Asset Returns and Moments of Asset Returns	4-41
Assignment Using the Portfolio Constructor	4-41
Assignment Using the setAssetMoments Method	4-43
Scalar Expansion of Arguments	4-43
Estimating Asset Moments from Prices or Returns	4-45
Estimating Asset Moments with Missing Data	4-48
Estimating Asset Moments from Time Series Data	4-50
Working with a Riskless Asset	4-51
Working with Transaction Costs	4-52
Working with Portfolio Constraints	4-57
Setting Default Constraints for Portfolio Weights	4-57
Working with Bound Constraints	4-60
Working with Budget Constraints	4-62
Working with Group Constraints	4-63
Working with Group Ratio Constraints	4-66
Working with Linear Equality Constraints	4-69
Working with Linear Inequality Constraints	4-71

Working with Average Turnover Constraints	4-73
Working with One-Way Turnover Constraints	4-75
Validate the Portfolio Problem	4-79
Validating a Portfolio Set	4-79
Validating Portfolios	4-81
Estimate Efficient Portfolios	4-84
Obtaining Portfolios Along the Entire Efficient Frontier	4-84
Obtaining Endpoints of the Efficient Frontier	4-85
Obtaining Efficient Portfolios for Target Returns	4-87
Obtaining Efficient Portfolios for Target Risks	4-90
Efficient Portfolio that Maximizes Sharpe Ratio	4-92
Choosing and Controlling the Solver	4-94
Estimate Efficient Frontiers	4-96
Obtaining Portfolio Risks and Returns	4-96
Plotting the Efficient Frontier	4-98
Postprocessing Results	4-104
Setting Up Tradable Portfolios	4-104
Troubleshooting Portfolio Optimization Results	4-105
Asset Allocation	4-109
Defining the Portfolio Problem	4-109
Simulating Asset Prices	4-110
Setting Up the Portfolio Object	4-111
Validating the Portfolio Problem	4-113
Plotting the Efficient Frontier	4-113
Evaluating Gross vs. Net Portfolio Returns	4-114
Analyzing Descriptive Properties of the Portfolio Structures	4-115
Obtaining a Portfolio at the Specified Return Level on the Efficient Frontier	4-116
Obtaining a Portfolio at the Specified Risk Levels on the Efficient Frontier	4-116
Displaying the Final Results	4-118
Portfolio Optimization Examples	4-121

Portfolio Optimization Theory	5-2
Portfolio Optimization Problems	5-2
Portfolio Problem Specification	5-2
Return Proxy	5-3
Risk Proxy	5-5
Portfolio Set for Portfolio Optimization	5-7
Default Portfolio Problem	5-14
PortfolioCVaR Object Workflow	5-16
PortfolioCVaR Object	5-17
PortfolioCVaR Object Properties and Methods	5-17
Working with PortfolioCVaR Objects	5-21
Setting and Getting Properties	5-22
Displaying PortfolioCVaR Objects	5-23
Saving and Loading PortfolioCVaR Objects	5-23
Estimating Efficient Portfolios and Frontiers	5-23
Arrays of PortfolioCVaR Objects	5-23
Subclassing PortfolioCVaR Objects	5-24
Conventions for Representation of Data	5-24
Constructing the PortfolioCVaR Object	5-26
Syntax	5-26
PortfolioCVaR Problem Sufficiency	5-27
Constructor Examples	5-27
Common Operations on the PortfolioCVaR Object	5-34
Naming a PortfolioCVaR Object	5-34
Configuring the Assets in the Asset Universe	5-34
Setting Up a List of Asset Identifiers	5-35
Truncating and Padding Asset Lists	5-36
Setting Up an Initial or Current Portfolio	5-37
Asset Returns and Scenarios	5-41
How Stochastic Optimization Works	5-41
What Are Scenarios?	5-42
Setting Scenarios Using the PortfolioCVaR Constructor ...	5-42
Setting Scenarios Using the setScenarios Method	5-44
Estimating the Mean and Covariance of Scenarios	5-44
Simulating Normal Scenarios	5-45

Simulating Normal Scenarios from Returns or Prices	5-45
Simulating Normal Scenarios with Missing Data	5-47
Simulating Normal Scenarios from Time Series Data	5-49
Simulating Normal Scenarios with Mean and Covariance	5-50
Working with a Riskless Asset	5-51
Working with Transaction Costs	5-51
Working with CVaR Portfolio Constraints	5-56
Setting Default Constraints for Portfolio Weights	5-56
Working with Bound Constraints	5-59
Working with Budget Constraints	5-61
Working with Group Constraints	5-63
Working with Group Ratio Constraints	5-66
Working with Linear Equality Constraints	5-68
Working with Linear Inequality Constraints	5-70
Working with Average Turnover Constraints	5-72
Working with One-Way Turnover Constraints	5-74
Validate the CVaR Portfolio Problem	5-78
Validating a CVaR Portfolio Set	5-78
Validating CVaR Portfolios	5-80
Estimate Efficient Portfolios	5-83
Obtaining Portfolios Along the Entire Efficient Frontier	5-83
Obtaining Endpoints of the Efficient Frontier	5-85
Obtaining Efficient Portfolios for Target Returns	5-87
Obtaining Efficient Portfolios for Target Risks	5-89
Choosing and Controlling the Solver	5-92
Estimate Efficient Frontiers	5-95
Obtaining CVaR Portfolio Risks and Returns	5-95
Obtaining Portfolio Standard Deviation and VaR	5-97
Plotting the Efficient Frontier	5-98
Postprocessing Results	5-104
Setting Up Tradable Portfolios	5-104
Working with Other Portfolio Objects	5-105
Troubleshooting CVaR Portfolio Optimization Results	5-108

Portfolio Optimization Theory	6-2
Portfolio Optimization Problems	6-2
Portfolio Problem Specification	6-2
Return Proxy	6-3
Risk Proxy	6-5
Portfolio Set for Portfolio Optimization	6-7
Default Portfolio Problem	6-14
PortfolioMAD Object Workflow	6-16
PortfolioMAD Object	6-17
PortfolioMAD Object Properties and Methods	6-17
Working with PortfolioMAD Objects	6-21
Setting and Getting Properties	6-22
Displaying PortfolioMAD Objects	6-22
Saving and Loading PortfolioMAD Objects	6-23
Estimating Efficient Portfolios and Frontiers	6-23
Arrays of PortfolioMAD Objects	6-23
Subclassing PortfolioMAD Objects	6-24
Conventions for Representation of Data	6-24
Constructing the PortfolioMAD Object	6-26
Syntax	6-26
PortfolioMAD Problem Sufficiency	6-27
Constructor Examples	6-27
Common Operations on the PortfolioMAD Object	6-34
Naming a PortfolioMAD Object	6-34
Configuring the Assets in the Asset Universe	6-34
Setting Up a List of Asset Identifiers	6-35
Truncating and Padding Asset Lists	6-36
Setting Up an Initial or Current Portfolio	6-37
Asset Returns and Scenarios	6-41
How Stochastic Optimization Works	6-41
What Are Scenarios?	6-42
Setting Scenarios Using the PortfolioMAD Constructor	6-42
Setting Scenarios Using the setScenarios Method	6-44
Estimating the Mean and Covariance of Scenarios	6-44
Simulating Normal Scenarios	6-45

Simulating Normal Scenarios from Returns or Prices	6-45
Simulating Normal Scenarios with Missing Data	6-47
Simulating Normal Scenarios from Time Series Data	6-49
Simulating Normal Scenarios for Mean and Covariance	6-50
Working with a Riskless Asset	6-51
Working with Transaction Costs	6-51
Working with MAD Portfolio Constraints	6-56
Setting Default Constraints for Portfolio Weights	6-56
Working with Bound Constraints	6-59
Working with Budget Constraints	6-61
Working with Group Constraints	6-62
Working with Group Ratio Constraints	6-65
Working with Linear Equality Constraints	6-68
Working with Linear Inequality Constraints	6-70
Working with Average Turnover Constraints	6-72
Working with One-Way Turnover Constraints	6-74
Validate the MAD Portfolio Problem	6-78
Validating a MAD Portfolio Set	6-78
Validating MAD Portfolios	6-80
Estimate Efficient Portfolios	6-83
Obtaining Portfolios Along the Entire Efficient Frontier	6-83
Obtaining Endpoints of the Efficient Frontier	6-85
Obtaining Efficient Portfolios for Target Returns	6-87
Obtaining Efficient Portfolios for Target Risks	6-89
Choosing and Controlling the Solver	6-92
Estimate Efficient Frontiers	6-94
Obtaining MAD Portfolio Risks and Returns	6-94
Obtaining Portfolio Standard Deviation	6-96
Plotting the Efficient Frontier	6-97
Postprocessing Results	6-104
Setting Up Tradable Portfolios	6-104
Working with Other Portfolio Objects	6-105
Troubleshooting MAD Portfolio Optimization Results	6-108

Investment Performance Metrics

7

Overview of Performance Metrics	7-2
Performance Metrics Types	7-2
Performance Metrics	7-2
Using the Sharpe Ratio	7-5
Introduction	7-5
Sharpe Ratio	7-5
Using the Information Ratio	7-7
Introduction	7-7
Information Ratio	7-7
Tracking Error	7-9
Introduction	7-9
Tracking Error	7-9
Risk-Adjusted Return	7-10
Introduction	7-10
Risk-Adjusted Return	7-10
Sample and Expected Lower Partial Moments	7-12
Introduction	7-12
Sample Lower Partial Moments	7-12
Expected Lower Partial Moments	7-13
Maximum and Expected Maximum Drawdown	7-15
Introduction	7-15
Maximum Drawdown	7-15
Expected Maximum Drawdown	7-18

Credit Risk Analysis

8

Credit Rating by Bagging Decision Trees	8-2
--	-----

Estimation of Transition Probabilities	8-21
Introduction	8-21
Estimate Transition Probabilities	8-21
Estimate Transition Probabilities for Different Rating Scales	8-24
Estimate Point-in-Time and Through-the-Cycle Probabilities	8-25
Estimate <i>t</i> -Year Default Probabilities	8-28
Estimate Bootstrap Confidence Intervals	8-29
Group Credit Ratings	8-30
Work with Nonsquare Matrices	8-32
Remove Outliers	8-34
Estimate Probabilities for Different Segments	8-35
Work with Large Datasets	8-36
Forecasting Corporate Default Rates	8-39
Credit Quality Thresholds	8-69
Introduction	8-69
Compute Credit Quality Thresholds	8-69
Visualize Credit Quality Thresholds	8-70
About Credit Scorecards	8-73
What Is a Credit Scorecard?	8-73
Credit Scorecard Development Process	8-76
Credit Scorecard Modeling Workflow	8-78
Troubleshooting Credit Scorecard Results	8-80
Predictor Name Is Unspecified and the Parser Returns an Error	8-80
Using bininfo or plotbins Before Binning	8-80
If Categorical Data Is Given as Numeric or Vice Versa	8-83
NaNs Returned When Scoring a “Test” Dataset	8-84
Case Study for a Credit Scorecard Analysis	8-87

Regression with Missing Data

9

Multivariate Normal Regression	9-2
Introduction	9-2

Multivariate Normal Linear Regression	9-2
Maximum Likelihood Estimation	9-3
Special Case of Multiple Linear Regression Model	9-4
Least-Squares Regression	9-5
Mean and Covariance Estimation	9-5
Convergence	9-5
Fisher Information	9-6
Statistical Tests	9-7
Maximum Likelihood Estimation with Missing Data	9-8
Introduction	9-8
ECM Algorithm	9-9
Standard Errors	9-9
Data Augmentation	9-10
Multivariate Normal Regression Functions	9-11
Multivariate Normal Regression Without Missing Data	9-12
Multivariate Normal Regression With Missing Data	9-13
Least-Squares Regression with Missing Data	9-13
Multivariate Normal Parameter Estimation with Missing Data	9-14
Support Functions	9-14
Multivariate Normal Regression Types	9-15
Regressions	9-15
Multivariate Normal Regression	9-16
Multivariate Normal Regression Without Missing Data	9-16
Multivariate Normal Regression with Missing Data	9-16
Least-Squares Regression	9-16
Least-Squares Regression Without Missing Data	9-17
Least-Squares Regression with Missing Data	9-17
Covariance-Weighted Least Squares	9-17
Covariance-Weighted Least Squares Without Missing Data	9-18
Covariance-Weighted Least Squares with Missing Data	9-18
Feasible Generalized Least Squares	9-18
Feasible Generalized Least Squares Without Missing Data	9-18
Feasible Generalized Least Squares with Missing Data	9-19
Seemingly Unrelated Regression	9-19
Seemingly Unrelated Regression Without Missing Data	9-20
Seemingly Unrelated Regression with Missing Data	9-20
Mean and Covariance Parameter Estimation	9-21
Troubleshooting Multivariate Normal Regression	9-21
Slow Convergence	9-22
Nonrandom Residuals	9-22
Nonconvergence	9-23

Portfolios with Missing Data	9-24
Valuation with Missing Data	9-30
Introduction	9-30
Capital Asset Pricing Model	9-30
Estimation of the CAPM	9-31
Estimation with Missing Data	9-32
Estimation of Some Technology Stock Betas	9-32
Grouped Estimation of Some Technology Stock Betas	9-34
References	9-37

Solving Sample Problems

10

Introduction	10-2
Common Problems in Finance	10-3
Sensitivity of Bond Prices to Interest Rates	10-3
Bond Portfolio for Hedging Duration and Convexity	10-6
Bond Prices and Yield Curve Parallel Shifts	10-8
Bond Prices and Yield Curve Nonparallel Shifts	10-12
Greek-Neutral Portfolios of European Stock Options	10-14
Term Structure Analysis and Interest-Rate Swaps	10-17
Producing Graphics with the Toolbox	10-21
Introduction	10-21
Plotting an Efficient Frontier	10-21
Plotting Sensitivities of an Option	10-24
Plotting Sensitivities of a Portfolio of Options	10-26

Financial Time Series Analysis

11

Analyzing Financial Time Series	11-2
Creating Financial Time Series Objects	11-3
Introduction	11-3

Using the Constructor	11-3
Transforming a Text File	11-13
Visualizing Financial Time Series Objects	11-16
Introduction	11-16
Using chartfts	11-16
Zoom Tool	11-19
Combine Axes Tool	11-22

Using Financial Time Series

12

Introduction	12-2
Working with Financial Time Series Objects	12-3
Introduction	12-3
Financial Time Series Object Structure	12-3
Data Extraction	12-4
Object-to-Matrix Conversion	12-6
Indexing a Financial Time Series Object	12-7
Financial Time Series Operations	12-14
Data Transformation and Frequency Conversion	12-17
Using Time Series to Predict Equity Return	12-22
Overview	12-22
Load the Data	12-23
Create Financial Time Series Objects	12-23
Create Closing Prices Adjustment Series	12-24
Adjust Closing Prices and Make Them Spot Prices	12-25
Create Return Series	12-25
Regress Return Series Against Metric Data	12-25
Plot the Results	12-26
Calculate the Dividend Rate	12-27

What Is the Financial Time Series App?	13-2
Getting Started with the Financial Time Series App	13-3
Loading Data with the Financial Time Series App	13-5
Overview	13-5
Obtaining External Data	13-5
Obtaining Internal Data	13-7
Viewing the MATLAB Workspace	13-8
Using the Financial Time Series App	13-10
Creating a Financial Time Series Object	13-10
Merge Financial Time Series Objects	13-11
Converting a Financial Time Series Object to a MATLAB Double-Precision Matrix	13-11
Plotting the Output in Several Formats	13-12
Viewing Data for a Financial Time Series Object in the Data Table	13-13
Modifying Data for a Financial Time Series Object in the Data Table	13-14
Viewing and Modifying the Properties for a FINTS Object	13-16
Using the Financial Time Series App with GUIs	13-17

Financial Time Series Graphical User Interface

Financial Time Series Graphical User Interface	14-2
Main Window	14-2
Using the Financial Time Series GUI	14-7
Getting Started	14-7
Data Menu	14-9
Analysis Menu	14-13
Graphs Menu	14-14
Saving Time Series Data	14-18

Trading Date Utilities

15

Trading Calendars Graphical User Interface	15-2
UICalendar Graphical User Interface	15-4
Using UICalendar in Standalone Mode	15-4
Using UICalendar with an Application	15-5

Technical Analysis

16

Technical Indicators	16-2
Technical Analysis Examples	16-4
Overview	16-4
Moving Average Convergence/Divergence (MACD)	16-4
Williams %R	16-5
Relative Strength Index (RSI)	16-7
On-Balance Volume (OBV)	16-8

Stochastic Differential Equations

17

SDEs	17-2
SDE Modeling	17-2
Trials vs. Paths	17-3
NTRIALS, NPERIODS, and NSTEPS	17-3
SDE Class Hierarchy	17-5
SDE Models	17-7
Introduction	17-7
Creating SDE Objects	17-7
Drift and Diffusion	17-12
Available Models	17-13

SDE Methods	17-14
Base SDE Models	17-15
Overview	17-15
Example: Base SDE Models	17-15
Drift and Diffusion Models	17-18
Overview	17-18
Example: Drift and Diffusion Rates	17-19
Example: SDEDDO Models	17-20
Linear Drift Models	17-22
Overview	17-22
Example: SDELD Models	17-22
Parametric Models	17-24
Creating Brownian Motion (BM) Models	17-24
Example: BM Models	17-24
Creating Constant Elasticity of Variance (CEV) Models	17-25
Creating Geometric Brownian Motion (GBM) Models	17-26
Creating Stochastic Differential Equations from Mean- Reverting Drift (SDEMRD) Models	17-27
Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models	17-28
Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models	17-29
Creating Heston Stochastic Volatility Models	17-31
Simulating Equity Prices	17-33
Simulating Multidimensional Market Models	17-33
Inducing Dependence and Correlation	17-48
Dynamic Behavior of Market Parameters	17-50
Pricing Equity Options	17-56
Simulating Interest Rates	17-59
Simulating Interest Rates	17-59
Ensuring Positive Interest Rates	17-66
Stratified Sampling	17-70
Performance Considerations	17-75
Managing Memory	17-75
Enhancing Performance	17-76

Optimizing Accuracy: About Solution Precision and Error	17-77
Pricing American Basket Options by Monte Carlo Simulation	17-82

Functions — Alphabetical List

18

Bibliography

A

Bond Pricing and Yields	A-2
Term Structure of Interest Rates	A-3
Derivatives Pricing and Yields	A-4
Portfolio Analysis	A-5
Investment Performance Metrics	A-6
Financial Statistics	A-7
Standard References	A-8
Credit Risk Analysis	A-9
Portfolio Optimization	A-10
Stochastic Differential Equations	A-11

Glossary

Getting Started

- “Financial Toolbox Product Description” on page 1-2
- “Expected Users” on page 1-3
- “Analyze Sets of Numbers Using Matrix Functions” on page 1-4
- “Matrix Algebra Refresher” on page 1-7
- “Using Input and Output Arguments with Functions” on page 1-17

Financial Toolbox Product Description

Analyze financial data and develop financial models

Financial Toolbox provides functions for mathematical modeling and statistical analysis of financial data. You can optimize portfolios of financial instruments, optionally taking into account turnover and transaction costs. The toolbox enables you to estimate risk, analyze interest rate levels, price equity and interest rate derivatives, and measure investment performance. Time series analysis functions and an app let you perform transformations or regressions with missing data and convert between different trading calendars and day-count conventions.

Key Features

- Mean-variance and CVaR-based object-oriented portfolio optimization
- Cash flow analysis, risk analysis, financial time-series modeling, date math, and calendar math
- Basic SIA-compliant fixed-income security analysis
- Basic Black-Scholes, Black, and binomial option pricing
- Regression and estimation with missing data
- Basic GARCH estimation, simulation, and forecasting
- Technical indicators and financial charts

Expected Users

In general, this guide assumes experience working with financial derivatives and some familiarity with the underlying models.

In designing Financial Toolbox documentation, we assume that your title is like 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

Analyze Sets of Numbers Using Matrix Functions

In this section...

“Introduction” on page 1-4

“Key Definitions” on page 1-4

“Referencing Matrix Elements” on page 1-5

“Transposing Matrices” on page 1-6

Introduction

Many financial analysis procedures involve *sets* of numbers; for example, a portfolio of securities at various prices and yields. Matrices, matrix functions, and matrix algebra are the most efficient ways to analyze sets of numbers and their relationships. Spreadsheets focus on individual cells and the relationships between cells. While you can think of a set of spreadsheet cells (a range of rows and columns) as a matrix, a matrix-oriented tool like MATLAB[®] software manipulates sets of numbers more quickly, easily, and naturally. For more information, see “Matrix Algebra Refresher” on page 1-7.

Key Definitions

Matrix

A rectangular array of numeric or algebraic quantities subject to mathematical operations; the regular formation of elements into rows and columns. Described as a “*m*-by-*n*” matrix, with *m* the number of rows and *n* the number of columns. The description is always “row-by-column.” For example, here is a 2-by-3 matrix of two bonds (the rows) with different par values, coupon rates, and coupon payment frequencies per year (the columns) entered using MATLAB notation:

```
Bonds = [1000    0.06    2
          500     0.055   4]
```

Vector

A matrix with only one row or column. Described as a “1-by-*n*” or “*m*-by-1” matrix. The description is always “row-by-column.” For example, here is a 1-by-4 vector of cash flows in MATLAB notation:

```
Cash = [1500    4470    5280   -1299]
```

Scalar

A 1-by-1 matrix; that is, a single number.

Referencing Matrix Elements

To reference specific matrix elements, use (row, column) notation. For example:

```
Bonds(1,2)
```

```
ans =
```

```
    0.06
```

```
Cash(3)
```

```
ans =
```

```
5280.00
```

You can enlarge matrices using small matrices or vectors as elements. For example,

```
AddBond = [1000    0.065    2];
```

```
Bonds = [Bonds; AddBond]
```

adds another row to the matrix and creates

```
Bonds =
```

```
    1000    0.06    2
     500    0.055    4
     1000    0.065    2
```

Likewise,

```
Prices = [987.50
          475.00
          995.00]
```

```
Bonds = [Prices, Bonds]
```

adds another column and creates

```
Bonds =
```

```
987.50    1000    0.06    2
475.00    500    0.055    4
995.00    1000    0.065    2
```

Finally, the colon (:) is important in generating and referencing matrix elements. For example, to reference the par value, coupon rate, and coupon frequency of the second bond:

```
BondItems = Bonds(2, 2:4)
```

```
BondItems =
```

```
500.00    0.055    4
```

Transposing Matrices

Sometimes matrices are in the wrong configuration for an operation. In MATLAB, the apostrophe or prime character (') transposes a matrix: columns become rows, rows become columns. For example,

```
Cash = [1500    4470    5280    -1299]'
```

produces

```
Cash =
```

```
1500
4470
5280
-1299
```

Matrix Algebra Refresher

In this section...

“Introduction” on page 1-7

“Adding and Subtracting Matrices” on page 1-7

“Multiplying Matrices” on page 1-8

“Dividing Matrices” on page 1-12

“Solving Simultaneous Linear Equations” on page 1-13

“Operating Element by Element” on page 1-16

Introduction

The explanations in the sections that follow should help refresh your skills for using matrix algebra and using MATLAB functions.

In addition, William Sharpe's *Macro-Investment Analysis* also provides an excellent explanation of matrix algebra operations using MATLAB. It is available on the Web at:

<http://www.stanford.edu/~wfsarpe/mia/mia.htm>

Tip When you are setting up a problem, it helps to “talk through” the units and dimensions associated with each input and output matrix. In the example under “Multiplying Matrices” on page 1-8, one input matrix has “five days' closing prices for three stocks,” the other input matrix has “shares of three stocks in two portfolios,” and the output matrix therefore has “five days' closing values for two portfolios.” It also helps to name variables using descriptive terms.

Adding and Subtracting Matrices

Matrix addition and subtraction operate element-by-element. The two input matrices must have the same dimensions. The result is a new matrix of the same dimensions where each element is the sum or difference of each corresponding input element. For example, consider combining portfolios of different quantities of the same stocks (“shares of stocks A, B, and C [the rows] in portfolios P and Q [the columns] plus shares of A, B, and C in portfolios R and S”).

```
Portfolios_PQ = [100  200
```

```

                500  400
                300  150];

Portfolios_RS = [175  125
                 200  200
                 100  500];

NewPortfolios = Portfolios_PQ + Portfolios_RS

NewPortfolios =

                275          325
                700          600
                400          650
```

Adding or subtracting a scalar and a matrix is allowed and also operates element-by-element.

```
SmallerPortf = NewPortfolios-10

SmallerPortf =

                265.00          315.00
                690.00          590.00
                390.00          640.00
```

Multiplying Matrices

Matrix multiplication does *not* operate element-by-element. It operates according to the rules of linear algebra. In multiplying matrices, it helps to remember this key rule: the inner dimensions must be the same. That is, if the first matrix is m -by- 3 , the second must be 3 -by- n . The resulting matrix is m -by- n . It also helps to “talk through” the units of each matrix, as mentioned in “Analyze Sets of Numbers Using Matrix Functions” on page 1-4.

Matrix multiplication also is *not* commutative; that is, it is not independent of order. $A*B$ does *not* equal $B*A$. The dimension rule illustrates this property. If A is 1 -by- 3 matrix and B is 3 -by- 1 matrix, $A*B$ yields a scalar (1 -by- 1) matrix but $B*A$ yields a 3 -by- 3 matrix.

Multiplying Vectors

Vector multiplication follows the same rules and helps illustrate the principles. For example, a stock portfolio has three different stocks and their closing prices today are:

```
ClosePrices = [42.5  15  78.875]
```

The portfolio contains these numbers of shares of each stock.

```
NumShares = [100
              500
              300]
```

To find the value of the portfolio, multiply the vectors

```
PortfValue = ClosePrices * NumShares
```

which yields:

```
PortfValue =
           3.5413e+004
```

The vectors are 1-by-3 and 3-by-1; the resulting vector is 1-by-1, a scalar. Multiplying these vectors thus means multiplying each closing price by its respective number of shares and summing the result.

To illustrate order dependence, switch the order of the vectors

```
Values = NumShares * ClosePrices
```

```
Values =
1.0e+004 *
    0.4250    0.1500    0.7887
    2.1250    0.7500    3.9438
    1.2750    0.4500    2.3663
```

which shows the closing values of 100, 500, and 300 shares of each stock, not the portfolio value, and meaningless for this example.

Computing Dot Products of Vectors

In matrix algebra, if X and Y are vectors of the same length

$$Y = [y_1, y_2, \dots, y_n]$$

$$X = [x_1, x_2, \dots, x_n]$$

then the dot product

$$X \cdot Y = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$$

is the scalar product of the two vectors. It is an exception to the commutative rule. To compute the dot product in MATLAB, use `sum(X .* Y)` or `sum(Y .* X)`. Just be sure the two vectors have the same dimensions. To illustrate, use the previous vectors.

```
Value = sum(NumShares .* ClosePrices')
```

```
Value =
```

```
3.5413e+004
```

```
Value = sum(ClosePrices .* NumShares')
```

```
Value =
```

```
3.5413e+004
```

As expected, the value in these cases matches the `PortfValue` computed previously.

Multiplying Vectors and Matrices

Multiplying vectors and matrices follows the matrix multiplication rules and process. For example, a portfolio matrix contains closing prices for a week. A second matrix (vector) contains the stock quantities in the portfolio.

```
WeekClosePr = [42.5    15    78.875
               42.125  15.5   78.75
               42.125  15.125  79
               42.625  15.25  78.875
               43     15.25  78.625];
PortQuan = [100
            500
            300];
```

To see the closing portfolio value for each day, simply multiply

```
WeekPortValue = WeekClosePr * PortQuan
```

```
WeekPortValue =
```

```
1.0e+004 *
```



```

3.5412
3.5587
3.5475
3.5550
3.5513

```

The prices matrix is 5-by-3, the quantity matrix (vector) is 3-by-1, so the resulting matrix (vector) is 5-by-1.

Multiplying Two Matrices

Matrix multiplication also follows the rules of matrix algebra. In matrix algebra notation, if A is an m -by- n matrix and B is an n -by- p matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & \cdots & b_{1j} & \cdots & b_{1p} \\ b_{21} & \cdots & b_{2j} & \cdots & b_{2p} \\ \vdots & & \vdots & & \vdots \\ b_{n1} & \cdots & b_{nj} & \cdots & b_{np} \end{bmatrix}$$

then $C = A*B$ is an m -by- p matrix; and the element c_{ij} in the i th row and j th column of C is

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}.$$

To illustrate, assume there are two portfolios of the same three stocks above but with different quantities.

```

Portfolios = [100  200
              500  400
              300  150];

```

Multiplying the 5-by-3 week's closing prices matrix by the 3-by-2 portfolios matrix yields a 5-by-2 matrix showing each day's closing value for both portfolios.

```
PortfolioValues = WeekClosePr * Portfolios
```

```
PortfolioValues =
```

```
1.0e+004 *
```

```
3.5412    2.6331
3.5587    2.6437
3.5475    2.6325
3.5550    2.6456
3.5513    2.6494
```

Monday's values result from multiplying each Monday closing price by its respective number of shares and summing the result for the first portfolio, then doing the same for the second portfolio. Tuesday's values result from multiplying each Tuesday closing price by its respective number of shares and summing the result for the first portfolio, then doing the same for the second portfolio. And so on through the rest of the week. With one simple command, MATLAB quickly performs many calculations.

Multiplying a Matrix by a Scalar

Multiplying a matrix by a scalar is an exception to the dimension and commutative rules. It just operates element-by-element.

```
Portfolios = [100    200
              500    400
              300    150];

DoublePort = Portfolios * 2

DoublePort =
    200         400
   1000         800
    600         300
```

Dividing Matrices

Matrix division is useful primarily for solving equations, and especially for solving simultaneous linear equations (see “Solving Simultaneous Linear Equations” on page 1-13). For example, you want to solve for X in $A^*X = B$.

In ordinary algebra, you would divide both sides of the equation by A , and X would equal B/A . However, since matrix algebra is not commutative ($A^*X \neq X^*A$), different processes apply. In formal matrix algebra, the solution involves matrix inversion. MATLAB, however, simplifies the process by providing two matrix division symbols, left and right (\backslash and $/$). In general,

$X = A \backslash B$ solves for X in $A^*X = B$ and

$X = B/A$ solves for X in $X*A = B$.

In general, matrix A must be a nonsingular square matrix; that is, it must be invertible and it must have the same number of rows and columns. (Generally, a matrix is invertible if the matrix times its inverse equals the identity matrix. To understand the theory and proofs, consult a textbook on linear algebra such as *Elementary Linear Algebra* by Hill listed in Appendix A.) MATLAB gives a warning message if the matrix is singular or nearly so.

Solving Simultaneous Linear Equations

Matrix division is especially useful in solving simultaneous linear equations. Consider this problem: Given two portfolios of mortgage-based instruments, each with certain yields depending on the prime rate, how do you weight the portfolios to achieve certain annual cash flows? The answer involves solving two linear equations.

A linear equation is any equation of the form

$$a_1x + a_2y = b,$$

where a_1 , a_2 , and b are constants (with a_1 and a_2 not both 0), and x and y are variables. (It's a linear equation because it describes a line in the xy -plane. For example, the equation $2x + y = 8$ describes a line such that if $x = 2$, then $y = 4$.)

A system of linear equations is a set of linear equations that you usually want to solve at the same time; that is, simultaneously. A basic principle for exact answers in solving simultaneous linear equations requires that there be as many equations as there are unknowns. To get exact answers for x and y , there must be two equations. For example, to solve for x and y in the system of linear equations

$$\begin{aligned} 2x + y &= 13 \\ x - 3y &= -18, \end{aligned}$$

there must be two equations, which there are. Matrix algebra represents this system as an equation involving three matrices: A for the left-side constants, X for the variables, and B for the right-side constants

$$A = \begin{bmatrix} 2 & 1 \\ 1 & -3 \end{bmatrix}, \quad X = \begin{bmatrix} x \\ y \end{bmatrix}, \quad B = \begin{bmatrix} 13 \\ -18 \end{bmatrix},$$

where $A * X = B$.

Solving the system simultaneously means solving for X . Using MATLAB,

$$A = \begin{bmatrix} 2 & 1 \\ 1 & -3 \end{bmatrix};$$

$$B = \begin{bmatrix} 13 \\ -18 \end{bmatrix};$$

$$X = A \setminus B$$

solves for X in $A * X = B$.

$$X = \begin{bmatrix} 3 & 7 \end{bmatrix}$$

So $x = 3$ and $y = 7$ in this example. In general, you can use matrix algebra to solve any system of linear equations such as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \end{aligned}$$

by representing them as matrices

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

and solving for X in $A * X = B$.

To illustrate, consider this situation. There are two portfolios of mortgage-based instruments, M1 and M2. They have current annual cash payments of \$100 and \$70 per unit, respectively, based on today's prime rate. If the prime rate moves down one percentage point, their payments would be \$80 and \$40. An investor holds 10 units of M1 and 20 units of M2. The investor's receipts equal cash payments times units, or $R = C * U$, for each prime-rate scenario. As word equations:

	M1	M2
Prime flat:	\$100 * 10 units	+ \$70 * 20 units = \$2400 receipts
Prime down:	\$80 * 10 units	+ \$40 * 20 units = \$1600 receipts

As MATLAB matrices:

```
Cash = [100 70
        80 40];
```

```
Units = [10
         20];
```

```
Receipts = Cash * Units
```

```
Receipts =
```

```
    2400
    1600
```

Now the investor asks this question: Given these two portfolios and their characteristics, how many units of each should I hold to receive \$7000 if the prime rate stays flat and \$5000 if the prime drops one percentage point? Find the answer by solving two linear equations.

	M1	M2
Prime flat:	\$100 * x units	+ \$70 * y units = \$7000 receipts
Prime down:	\$80 * x units	+ \$40 * y units = \$5000 receipts

In other words, solve for U (units) in the equation R (receipts) = C (cash) * U (units). Using MATLAB left division

```
Cash = [100 70
        80 40];
```

```
Receipts = [7000
           5000];
```

```
Units = Cash \ Receipts
```

```
Units =
```

```
    43.7500  
    37.5000
```

The investor should hold 43.75 units of portfolio M1 and 37.5 units of portfolio M2 to achieve the annual receipts desired.

Operating Element by Element

Finally, element-by-element arithmetic operations are called *array* operations. To indicate a MATLAB array operation, precede the operator with a period (.). Addition and subtraction, and matrix multiplication and division by a scalar, are already array operations so no period is necessary. When using array operations on two matrices, the dimensions of the matrices must be the same. For example, given vectors of stock dividends and closing prices

```
Dividends = [1.90  0.40  1.56  4.50];  
Prices = [25.625  17.75  26.125  60.50];
```

```
Yields = Dividends ./ Prices
```

```
Yields =
```

```
    0.0741    0.0225    0.0597    0.0744
```

Using Input and Output Arguments with Functions

In this section...

“Input Arguments” on page 1-17

“Output Arguments” on page 1-19

“Interest Rate Arguments” on page 1-20

Input Arguments

Matrix Input

MATLAB software was designed to be a large-scale array (vector or matrix) processor. In addition to its linear algebra applications, the general array-based processing facility can perform repeated operations on collections of data. When MATLAB code is written to operate simultaneously on collections of data stored in arrays, the code is said to be vectorized. Vectorized code is not only clean and concise, but is also efficiently processed by the underlying MATLAB engine.

Because MATLAB can process vectors and matrices easily, most Financial Toolbox functions allow vector or matrix input arguments, rather than just single (scalar) values. For example, the `irr` function computes the internal rate of return of a cash flow stream. It accepts a vector of cash flows and returns a scalar-valued internal rate of return. However, it also accepts a matrix of cash flow streams, a column in the matrix representing a different cash flow stream. In this case, `irr` returns a vector of internal rates of return, each entry in the vector corresponding to a column of the input matrix. Many other toolbox functions work similarly.

As an example, suppose you make an initial investment of \$100, from which you then receive by a series of annual cash receipts of \$10, \$20, \$30, \$40, and \$50. This cash flow stream may be stored in a vector

```
CashFlows = [-100 10 20 30 40 50]'
```

which MATLAB displays as

```
CashFlows =  
-100  
 10  
 20
```

```
30
40
50
```

The `irr` function can compute the internal rate of return of this stream.

```
Rate = irr(CashFlows)
```

The internal rate of return of this investment is

```
Rate =
    0.1201
```

or 12.01%.

In this case, a single cash flow stream (written as an input vector) produces a scalar output – the internal rate of return of the investment.

Extending this example, if you process a matrix of identical cash flow streams

```
Rate = irr([CashFlows CashFlows CashFlows])
```

you should expect to see identical internal rates of return for each of the three investments.

```
Rate =
    0.1201    0.1201    0.1201
```

This simple example illustrates the power of vectorized programming. The example shows how to collect data into a matrix and then use a toolbox function to compute answers for the entire collection. This feature can be useful in portfolio management, for example, where you might want to organize multiple assets into a single collection. Place data for each asset in a different column or row of a matrix, then pass the matrix to a Financial Toolbox function. MATLAB performs the same computation on all of the assets at once.

Matrices of String Input

Enter MATLAB strings surrounded by single quotes ('string').

Strings are stored as character arrays, one ASCII character per element. Thus, the date string


```
DateString = '9/16/2001'
```

is actually a 1-by-9 vector. Strings making up the rows of a matrix or vector all must have the same length. To enter several date strings, therefore, use a column vector and be sure all strings are the same length. Fill in with spaces or zeros. For example, to create a vector of dates corresponding to irregular cash flows

```
DateFields = [ '01/12/2001'
                '02/14/2001'
                '03/03/2001'
                '06/14/2001'
                '12/01/2001'];
```

DateFields actually becomes a 5-by-10 character array.

Don't mix numbers and strings in a matrix. If you do, MATLAB treats all entries as characters. For example,

```
Item = [83 90 99 '14-Sep-1999']
```

becomes a 1-by-14 character array, not a 1-by-4 vector, and it contains

```
Item =
SZc14-Sep-1999
```

Output Arguments

Some functions return no arguments, some return just one, and some return multiple arguments. Functions that return multiple arguments use the syntax

```
[A, B, C] = function(variables...)
```

to return arguments A, B, and C. If you omit all but one, the function returns the first argument. Thus, for this example if you use the syntax

```
X = function(variables...)
```

function returns a value for A, but not for B or C.

Some functions that return vectors accept only scalars as arguments. Why could such functions not accept vectors as arguments and return matrices, where each column in the output matrix corresponds to an entry in the input vector? The answer is that the output

vectors can be variable length and thus will not fit in a matrix without some convention to indicate that the shorter columns are missing data.

Functions that require asset life as an input, and return values corresponding to different periods over that life, cannot generally handle vectors or matrices as input arguments. Those functions are:

<code>amortize</code>	Amortization
<code>depxdb</code>	Fixed declining-balance depreciation
<code>depxdb</code>	General declining-balance depreciation
<code>depxdb</code>	Sum of years' digits depreciation

For example, suppose you have a collection of assets such as automobiles and you want to compute the depreciation schedules for them. The function `depxdb` computes a stream of declining-balance depreciation values for an asset. You might want to set up a vector where each entry is the initial value of each asset. `depxdb` also needs the lifetime of an asset. If you were to set up such a collection of automobiles as an input vector, and the lifetimes of those automobiles varied, the resulting depreciation streams would differ in length according to the life of each automobile, and the output column lengths would vary. A matrix must have the same number of rows in each column.

Interest Rate Arguments

One common argument, both as input and output, is interest rate. All Financial Toolbox functions expect and return interest rates as decimal fractions. Thus an interest rate of 9.5% is indicated as 0.095.

Performing Common Financial Tasks

- “Introduction” on page 2-2
- “Handle and Convert Dates” on page 2-4
- “Format Currency” on page 2-11
- “Charting Financial Data” on page 2-12
- “Analyzing and Computing Cash Flows” on page 2-16
- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-20
- “Treasury Bills Defined” on page 2-33
- “Computing Treasury Bill Price and Yield” on page 2-34
- “Term Structure of Interest Rates” on page 2-38
- “Pricing and Analyzing Equity Derivatives” on page 2-41

Introduction

Financial Toolbox software contains functions that perform many common financial tasks, including:

- “Handle and Convert Dates” on page 2-4

Calendar functions convert dates among different formats (including Excel® formats), determine future or past dates, find dates of holidays and business days, compute time differences between dates, find coupon dates and coupon periods for coupon bonds, and compute time periods based on 360-, 365-, or 366-day years.

- “Format Currency” on page 2-11

The toolbox includes functions for handling decimal values in bank (currency) formats and as fractional prices.

- “Charting Financial Data” on page 2-12

Charting functions produce a variety of financial charts including Bollinger bands, high-low-close charts, candlestick plots, point and figure plots, and moving-average plots.

- “Analyzing and Computing Cash Flows” on page 2-16

Cash-flow evaluation and financial accounting functions compute interest rates, rates of return, payments associated with loans and annuities, future and present values, depreciation, and other standard accounting calculations associated with cash-flow streams.

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-20

Securities Industry Association (SIA) compliant fixed-income functions compute prices, yields, accrued interest, and sensitivities for securities such as bonds, zero-coupon bonds, and Treasury bills. They handle odd first and last periods in price/yield calculations, compute accrued interest and discount rates, and calculate convexity and duration. Another set of functions analyzes term structure of interest rates, including pricing bonds from yield curves and bootstrapping yield curves from market prices.

- “Pricing and Analyzing Equity Derivatives” on page 2-41

Derivatives analysis functions compute prices, yields, and sensitivities for derivative securities. They deal with both European and American options.

Black-Scholes functions work with European options. They compute delta, gamma, lambda, rho, theta, and vega, as well as values of call and put options.

Binomial functions work with American options, computing put and call prices.

- “Analyzing Portfolios” on page 3-2

Portfolio analysis functions provide basic utilities to compute variances and covariance of portfolios, find combinations to minimize variance, compute Markowitz efficient frontiers, and calculate combined rates of return.

- Modeling volatility in time series.

Generalized Autoregressive Conditional Heteroskedasticity (GARCH)

functions model the volatility of univariate economic time series. (Econometrics Toolbox™ software provides a more comprehensive and integrated computing environment. For information, see the Econometrics Toolbox documentation or the financial products Web page at <http://www.mathworks.com/products/finprod>.)

Handle and Convert Dates

In this section...
“Date Formats” on page 2-4
“Date Conversions” on page 2-5
“Current Date and Time” on page 2-8
“Determining Dates” on page 2-8

Date Formats

Since virtually all financial data is dated or derives from a time series, financial functions must have extensive date-handling capabilities. You most often work with date strings (14-Sep-1999) when dealing with dates. Financial Toolbox software works internally with *serial date numbers* (for example, 730377). A serial date number represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB software, serial date number 1 is January 1, 0000 A.D. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So 6:00 p.m. on 14-Sep-1999, in MATLAB, is date number 730377.75.

Note If you specify a two-digit year, MATLAB assumes that the year lies within the 100-year period centered about the current year. See the function `datenum` for specific information. MATLAB internal date handling and calculations generate no ambiguous values. However, whenever possible, programmers should use serial date numbers or date strings containing four-digit years.

Many toolbox functions that require dates accept either date strings or serial date numbers. If you are dealing with a few dates at the MATLAB command-line level, date strings are more convenient. If you are using toolbox functions on large numbers of dates, as in analyzing large portfolios or cash flows, performance improves if you use date numbers.

The Financial Toolbox software provides functions that convert date strings to serial date numbers, and vice versa.

Date Conversions

Functions that convert between date formats are

<code>datedisp</code>	Displays a numeric matrix with date entries formatted as date strings
<code>datenum</code>	Converts a date string to a serial date number
<code>datestr</code>	Converts a serial date number to a date string
<code>m2xdate</code>	Converts MATLAB serial date number to Excel serial date number
<code>x2mdate</code>	Converts Excel serial date number to MATLAB serial date number

Another function, `datevec`, converts a date number or date string to a date vector whose elements are [Year Month Day Hour Minute Second]. Date vectors are mostly an internal format for some MATLAB functions; you would not often use them in financial calculations.

Input Conversions

The `datenum` function is important for using Financial Toolbox software efficiently. `datenum` takes an input string in any of several formats, with 'dd-mmm-yyyy', 'mm/dd/yyyy' or 'dd-mmm-yyyy, hh:mm:ss.ss' most common. The input string can have up to six fields formed by letters and numbers separated by any other characters:

- The day field is an integer from 1 through 31.
- The month field is either an integer from 1 through 12 or an alphabetical string with at least three characters.
- The year field is a nonnegative integer: if only two numbers are specified, then the year is assumed to lie within the 100-year period centered about the current year; if the year is omitted, the current year is used as the default.
- The hours, minutes, and seconds fields are optional. They are integers separated by colons or followed by 'am' or 'pm'.

For example, if the current year is 1999, then these are all equivalent

```
'17-May-1999'
'17-May-99'
'17-may'
'May 17, 1999'
```

```
'5/17/99'  
'5/17'
```

and both of these represent the same time.

```
'17-May-1999, 18:30'  
'5/17/99/6:30 pm'
```

Note that the default format for numbers-only input follows the American convention. Thus 3/6 is March 6, not June 3.

With `datenum` you can convert dates into serial date format, store them in a matrix variable, then later pass the variable to a function. Alternatively, you can use `datenum` directly in a function input argument list.

For example, consider the function `bndprice` that computes the price of a bond given the yield-to-maturity. First set up variables for the yield-to-maturity, coupon rate, and the necessary dates.

```
Yield      = 0.07;  
CouponRate = 0.08;  
Settle     = datenum('17-May-2000');  
Maturity   = datenum('01-Oct-2000');
```

Then call the function with the variables

```
bndprice(Yield, CouponRate, Settle, Maturity)
```

Alternatively, convert date strings to serial date numbers directly in the function input argument list.

```
bndprice(0.07, 0.08, datenum('17-May-2000'),...  
datenum('01-Oct-2000'))
```

`bndprice` is an example of a function designed to detect the presence of date strings and make the conversion automatically. For these functions date strings may be passed directly.

```
bndprice(0.07, 0.08, '17-May-2000', '01-Oct-2000')
```

The decision to represent dates as either date strings or serial date numbers is often a matter of convenience. For example, when formatting data for visual display or for debugging date-handling code, it is often much easier to view dates as date strings because serial date numbers are difficult to interpret. Alternatively, serial date numbers

are just another type of numeric data, and can be placed in a matrix along with any other numeric data for convenient manipulation.

Remember that if you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill with spaces or zeros. See “Matrices of String Input” on page 1-18.

Output Conversions

The function `datestr` converts a serial date number to one of 19 different date string output formats showing date, time, or both. The default output for dates is a day-month-year string, for example, 24-Aug-2000. This function is quite useful for preparing output reports.

Format	Description
01-Mar-2000 15:45:17	day-month-year hour:minute:second
01-Mar-2000	day-month-year
03/01/00	month/day/year
Mar	month, three letters
M	month, single letter
3	month
03/01	month/day
1	day of month
Wed	day of week, three letters
W	day of week, single letter
2000	year, four numbers
99	year, two numbers
Mar01	month year
15:45:17	hour:minute:second
03:45:17 PM	hour:minute:second AM or PM
15:45	hour:minute
03:45 PM	hour:minute AM or PM
Q1-99	calendar quarter-year
Q1	calendar quarter

Current Date and Time

The functions `today` and `now` return serial date numbers for the current date, and the current date and time, respectively.

```
today
```

```
ans =  
    730693
```

```
now
```

```
ans =  
    730693.48
```

The MATLAB function `date` returns a string for today's date.

```
date
```

```
ans =  
26-Jul-2000
```

Determining Dates

The Financial Toolbox software provides many functions for determining specific dates, including functions which account for holidays and other nontrading days. For example, you schedule an accounting procedure for the last Friday of every month. The `lweekdate` function returns those dates for 2000; the 6 specifies Friday.

```
Fridates = lweekdate(6, 2000, 1:12);
```

```
Fridays = datestr(Fridates)
```

```
Fridays =  
28-Jan-2000  
25-Feb-2000  
31-Mar-2000  
28-Apr-2000  
26-May-2000  
30-Jun-2000  
28-Jul-2000
```

```

25-Aug-2000
29-Sep-2000
27-Oct-2000
24-Nov-2000
29-Dec-2000

```

Or your company closes on Martin Luther King Jr. Day, which is the third Monday in January. The `nweekdate` function determines those dates for 2001 through 2004.

```
MLKDates = nweekdate(3, 2, 2001:2004, 1);
```

```
MLKDays = datestr(MLKDates)
```

```
MLKDays =
```

```

15-Jan-2001
21-Jan-2002
20-Jan-2003
19-Jan-2004

```

Accounting for holidays and other nontrading days is important when examining financial dates. The Financial Toolbox software provides the `holidays` function, which contains holidays and special nontrading days for the New York Stock Exchange between 1950 and 2030, inclusive. In addition, you can use `nyseclosures` to evaluate all known or anticipated closures of the New York Stock Exchange from January 1, 1885 to December 31, 2050. `nyseclosures` returns a vector of serial date numbers corresponding to market closures between the dates `StartDate` and `EndDate`, inclusive.

In this example, you can use `holidays` to determine the standard holidays in the last half of 2000:

```
LHHDates = holidays('1-Jul-2000', '31-Dec-2000');
```

```
LHHDays = datestr(LHHDates)
```

```
LHHDays =
```

```

04-Jul-2000
04-Sep-2000
23-Nov-2000
25-Dec-2000

```

Now use the toolbox `busdate` function to determine the next business day after these holidays.

```
LHNextDates = busdate(LHHDates);  
LHNextDays = datestr(LHNextDates)  
LHNextDays =  
05-Jul-2000  
05-Sep-2000  
24-Nov-2000  
26-Dec-2000
```

The toolbox also provides the `cfdates` function to determine cash-flow dates for securities with periodic payments. This function accounts for the coupons per year, the day-count basis, and the end-of-month rule. For example, to determine the cash-flow dates for a security that pays four coupons per year on the last day of the month, on an actual/365 day-count basis, just enter the settlement date, the maturity date, and the parameters.

```
PayDates = cfdates('14-Mar-2000', '30-Nov-2001', 4, 3, 1);  
PayDays = datestr(PayDates)  
PayDays =  
31-May-2000  
31-Aug-2000  
30-Nov-2000  
28-Feb-2001  
31-May-2001  
31-Aug-2001  
30-Nov-2001
```

Format Currency

Financial Toolbox software provides several functions to format currency and chart financial data. The currency formatting functions are

<code>cur2frac</code>	Converts decimal currency values to fractional values
<code>cur2str</code>	Converts a value to Financial Toolbox bank format
<code>frac2cur</code>	Converts fractional currency values to decimal values

These examples show their use.

```
Dec = frac2cur('12.1', 8)
```

returns `Dec = 12.125`, which is the decimal equivalent of 12-1/8. The second input variable is the denominator of the fraction.

```
Str = cur2str(-8264, 2)
```

returns the string `($8264.00)`. For this toolbox function, the output format is a numerical format with dollar sign prefix, two decimal places, and negative numbers in parentheses; for example, `($123.45)` and `$6789.01`. The standard MATLAB bank format uses two decimal places, no dollar sign, and a minus sign for negative numbers; for example, `-123.45` and `6789.01`.

Charting Financial Data

In this section...

“Introduction” on page 2-12

“High-Low-Close Chart Example” on page 2-13

“Bollinger Chart Example” on page 2-14

Introduction

The following toolbox financial charting functions plot financial data and produce presentation-quality figures quickly and easily.

<code>bolling</code>	Bollinger band chart
<code>bollinger</code>	Time series Bollinger band
<code>candle</code>	Candlestick chart
<code>candle</code>	Time series candle plot
<code>pointfig</code>	Point and figure chart
<code>highlow</code>	High, low, open, close chart
<code>highlow</code>	Time series High-Low plot
<code>movavg</code>	Leading and lagging moving averages chart

These functions work with standard MATLAB functions that draw axes, control appearance, and add labels and titles. The toolbox also provides a comprehensive set of charting functions that work with financial time series objects, see “Chart Technical Indicators”.

Here are two plotting examples: a high-low-close chart of sample IBM[®] stock price data, and a Bollinger band chart of the same data. These examples load data from an external file (`ibm.dat`), then call the functions using subsets of the data. The MATLAB variable `ibm`, which is created by loading `ibm.dat`, is a six-column matrix where each row is a trading day's data and where columns 2, 3, and 4 contain the high, low, and closing prices, respectively.

Note The data in `ibm.dat` is fictional and for illustrative use only.

High-Low-Close Chart Example

First load the data and set up matrix dimensions. `load` and `size` are standard MATLAB functions.

```
load ibm.dat;  
[ro, co] = size(ibm);
```

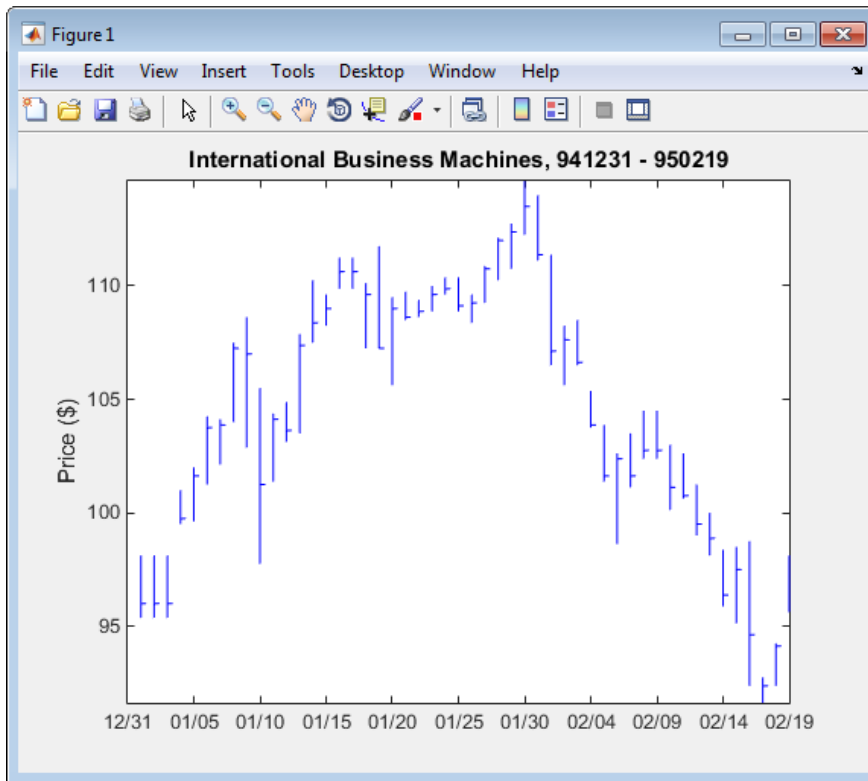
Open a figure window for the chart. Use the Financial Toolbox `highlow` function to plot high, low, and close prices for the last 50 trading days in the data file.

```
figure;  
highlow(ibm(ro-50:ro,2), ibm(ro-50:ro,3), ibm(ro-50:ro,4), [], 'b');
```

Add labels and title, and set axes with standard MATLAB functions. Use the Financial Toolbox `dateaxis` function to provide dates for the *x*-axis ticks.

```
xlabel('');  
ylabel('Price ($)');  
title('International Business Machines, 941231 - 950219');  
axis([0 50 -inf inf]);  
dateaxis('x',6,'31-Dec-1994')
```

MATLAB produces a figure like this. The plotted data and axes you see may differ. Viewed online, the high-low-close bars are blue.



Bollinger Chart Example

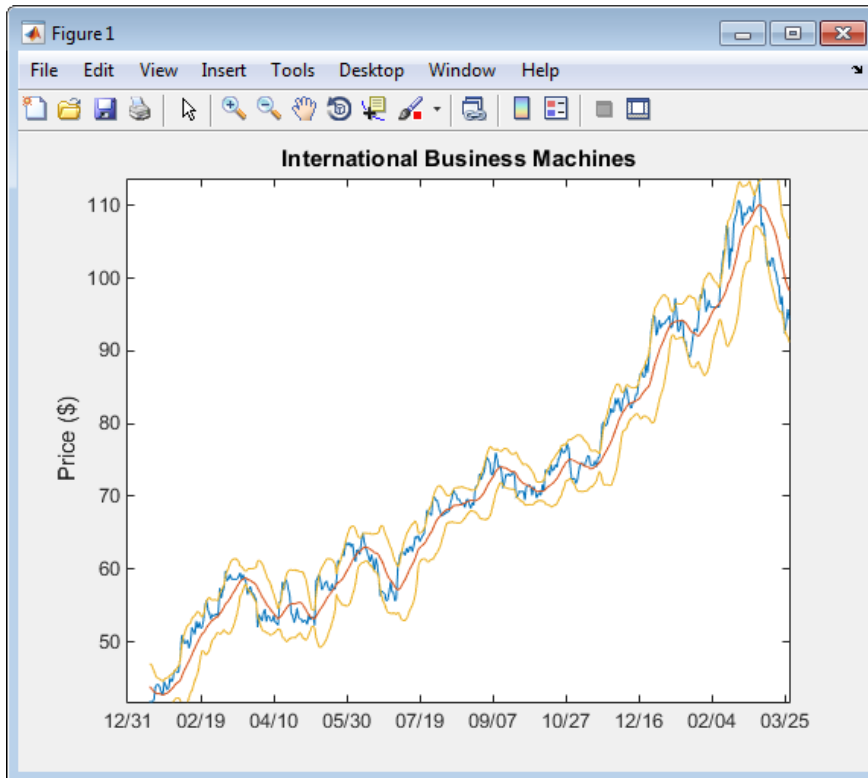
The `bolling` function in Financial Toolbox software produces a Bollinger band chart using all the closing prices in the same IBM stock price matrix. A Bollinger band chart plots actual data along with three other bands of data. The upper band is two standard deviations above a moving average; the lower band is two standard deviations below that moving average; and the middle band is the moving average itself. This example uses a 15-day moving average.

Assuming the previous IBM data is still loaded, execute the function.

```
bolling(ibm(:,4), 15, 0);
```

Specify the axes, labels, and titles. Again, use `dateaxis` to add the *x*-axis dates.


```
axis([0 ro min(ibm(:,4)) max(ibm(:,4))]);  
ylabel('Price ($)');  
title(['International Business Machines']);  
dateaxis('x', 6, '31-Dec-1994')
```



For help using MATLAB plotting functions, see “Create 2-D Graph and Customize Lines” in the MATLAB documentation. See the MATLAB documentation for details on the `axis`, `title`, `xlabel`, and `ylabel` functions.

Analyzing and Computing Cash Flows

In this section...

“Introduction” on page 2-16
 “Interest Rates/Rates of Return” on page 2-16
 “Present or Future Values” on page 2-17
 “Depreciation” on page 2-18
 “Annuities” on page 2-18

Introduction

Financial Toolbox cash-flow functions compute interest rates and rates of return, present or future values, depreciation streams, and annuities.

Some examples in this section use this income stream: an initial investment of \$20,000 followed by three annual return payments, a second investment of \$5,000, then four more returns. Investments are negative cash flows, return payments are positive cash flows.

```
Stream = [-20000, 2000, 2500, 3500, -5000, 6500, ...
          9500, 9500, 9500];
```

Interest Rates/Rates of Return

Several functions calculate interest rates involved with cash flows. To compute the internal rate of return of the cash stream, execute the toolbox function `irr`

```
ROR = irr(Stream)
```

which gives a rate of return of 11.72%.

Note that the internal rate of return of a cash flow may not have a unique value. Every time the sign changes in a cash flow, the equation defining `irr` can give up to two additional answers. An `irr` computation requires solving a polynomial equation, and the number of real roots of such an equation can depend on the number of sign changes in the coefficients. The equation for internal rate of return is

$$\frac{cf_1}{(1+r)} + \frac{cf_2}{(1+r)^2} + \dots + \frac{cf_n}{(1+r)^n} + Investment = 0,$$

where *Investment* is a (negative) initial cash outlay at time 0, cf_n is the cash flow in the n th period, and n is the number of periods. `irr` finds the rate r such that the present value of the cash flow equals the initial investment. If all of the cf_n s are positive there is only one solution. Every time there is a change of sign between coefficients, up to two additional real roots are possible.

Another toolbox rate function, `effrr`, calculates the effective rate of return given an annual interest rate (also known as nominal rate or annual percentage rate, APR) and number of compounding periods per year. To find the effective rate of a 9% APR compounded monthly, enter

```
Rate = effrr(0.09, 12)
```

The answer is 9.38%.

A companion function `nomrr` computes the nominal rate of return given the effective annual rate and the number of compounding periods.

Present or Future Values

The toolbox includes functions to compute the present or future value of cash flows at regular or irregular time intervals with equal or unequal payments: `fvfix`, `fvvar`, `pvfix`, and `pvvar`. The `-fix` functions assume equal cash flows at regular intervals, while the `-var` functions allow irregular cash flows at irregular periods.

Now compute the net present value of the sample income stream for which you computed the internal rate of return. This exercise also serves as a check on that calculation because the net present value of a cash stream at its internal rate of return should be zero. Enter

```
NPV = pvvar(Stream, ROR)
```

which returns an answer very close to zero. The answer usually is not *exactly* zero due to rounding errors and the computational precision of the computer.

Note Other toolbox functions behave similarly. The functions that compute a bond's yield, for example, often must solve a nonlinear equation. If you then use that yield to compute the net present value of the bond's income stream, it usually does not *exactly* equal the purchase price, but the difference is negligible for practical applications.

Depreciation

The toolbox includes functions to compute standard depreciation schedules: straight line, general declining-balance, fixed declining-balance, and sum of years' digits. Functions also compute a complete amortization schedule for an asset, and return the remaining depreciable value after a depreciation schedule has been applied.

This example depreciates an automobile worth \$15,000 over five years with a salvage value of \$1,500. It computes the general declining balance using two different depreciation rates: 50% (or 1.5), and 100% (or 2.0, also known as double declining balance). Enter

```
Decline1 = depreddb(15000, 1500, 5, 1.5)
Decline2 = depreddb(15000, 1500, 5, 2.0)
```

which returns

```
Decline1 =
    4500.00    3150.00    2205.00    1543.50    2101.50
Decline2 =
    6000.00    3600.00    2160.00    1296.00    444.00
```

These functions return the actual depreciation amount for the first four years and the remaining depreciable value as the entry for the fifth year.

Annuities

Several toolbox functions deal with annuities. This first example shows how to compute the interest rate associated with a series of loan payments when only the payment amounts and principal are known. For a loan whose original value was \$5000.00 and which was paid back monthly over four years at \$130.00/month

```
Rate = annurate(4*12, 130, 5000, 0, 0)
```

The function returns a rate of 0.0094 monthly, or about 11.28% annually.

The next example uses a present-value function to show how to compute the initial principal when the payment and rate are known. For a loan paid at \$300.00/month over four years at 11% annual interest

```
Principal = pvfix(0.11/12, 4*12, 300, 0, 0)
```

The function returns the original principal value of \$11,607.43.

The final example computes an amortization schedule for a loan or annuity. The original value was \$5000.00 and was paid back over 12 months at an annual rate of 9%.

```
[Prpmt, Intpmt, Balance, Payment] = ...
    amortize(0.09/12, 12, 5000, 0, 0);
```

This function returns vectors containing the amount of principal paid,

```
Prpmt = [399.76 402.76 405.78 408.82 411.89 414.97
         418.09 421.22 424.38 427.56 430.77 434.00]
```

the amount of interest paid,

```
Intpmt = [37.50 34.50 31.48 28.44 25.37 22.28
          19.17 16.03 12.88 9.69 6.49 3.26]
```

the remaining balance for each period of the loan,

```
Balance = [4600.24 4197.49 3791.71 3382.89 2971.01
           2556.03 2137.94 1716.72 1292.34 864.77
           434.00 0.00]
```

and a scalar for the monthly payment.

```
Payment = 437.26
```

Pricing and Computing Yields for Fixed-Income Securities

In this section...
“Introduction” on page 2-20
“Fixed-Income Terminology” on page 2-20
“Framework” on page 2-24
“Default Parameter Values” on page 2-25
“Coupon Date Calculations” on page 2-27
“Yield Conventions” on page 2-28
“Pricing Functions” on page 2-28
“Yield Functions” on page 2-29
“Fixed-Income Sensitivities” on page 2-30

Introduction

The Financial Toolbox product provides functions for computing accrued interest, price, yield, convexity, and duration of fixed-income securities. Various conventions exist for determining the details of these computations. The Financial Toolbox software supports conventions specified by the Securities Industry and Financial Markets Association (SIFMA), used in the US markets, the International Capital Market Association (ICMA), used mainly in the European markets, and the International Swaps and Derivatives Association (ISDA). Note that for historical reasons, SIFMA is referred to in Financial Toolbox documentation as SIA and ICMA is referred to as International Securities Market Association (ISMA).

Fixed-Income Terminology

Since terminology varies among texts on this subject, here are some basic definitions that apply to these Financial Toolbox functions. The Glossary contains additional definitions.

The *settlement date* of a bond is the date when money first changes hands; that is, when a buyer pays for a bond. It need not coincide with the *issue date*, which is the date a bond is first offered for sale.

The *first coupon date* and *last coupon date* are the dates when the first and last coupons are paid, respectively. Although bonds typically pay periodic annual or semiannual coupons, the length of the first and last coupon periods may differ from the standard

coupon period. The toolbox includes price and yield functions that handle these odd first and/or last periods.

Successive *quasi-coupon dates* determine the length of the standard coupon period for the fixed income security of interest, and do not necessarily coincide with actual coupon payment dates. The toolbox includes functions that calculate both actual and quasi-coupon dates for bonds with odd first and/or last periods.

Fixed-income securities can be purchased on dates that do not coincide with coupon payment dates. In this case, the bond owner is not entitled to the full value of the coupon for that period. When a bond is purchased between coupon dates, the buyer must compensate the seller for the pro-rata share of the coupon interest earned from the previous coupon payment date. This pro-rata share of the coupon payment is called *accrued interest*. The *purchase price*, the price actually paid for a bond, is the quoted market price plus accrued interest.

The *maturity date* of a bond is the date when the issuer returns the final face value, also known as the *redemption value* or *par value*, to the buyer. The *yield-to-maturity* of a bond is the nominal compound rate of return that equates the present value of all future cash flows (coupons and principal) to the current market price of the bond.

The *period* of a bond refers to the frequency with which the issuer of a bond makes coupon payments to the holder.

Period of a Bond

Period Value	Payment Schedule
0	No coupons (Zero coupon bond)
1	Annual
2	Semiannual
3	Tri-annual
4	Quarterly
6	Bi-monthly
12	Monthly

The *basis* of a bond refers to the basis or day-count convention for a bond. Basis is normally expressed as a fraction in which the numerator determines the number of days between two dates, and the denominator determines the number of days in the year. For example, the numerator of *actual/actual* means that when determining the number of

days between two dates, count the actual number of days; the denominator means that you use the actual number of days in the given year in any calculations (either 365 or 366 days depending on whether the given year is a leap year).

The day count convention determines how accrued interest is calculated and determines how cash flows for the bond are discounted, thereby effecting price and yield calculations. Furthermore, the SIA convention is to use the actual/actual day count convention for discounting cash flows in all cases.

Basis of a Bond

Basis Value	Meaning	Description
0 (default)	actual/actual	Actual days held over actual days in coupon period. Denominator is 365 in most years and 366 in a leap year.
1	30/360 (SIA)	Each month contains 30 days; a year contains 360 days. Payments are adjusted for bonds that pay coupons on the last day of February.
2	actual/360	Actual days held over 360.
3	actual/365	Actual days held over 365, even in leap years.
4	30/360 BMA (Bond Market Association)	Each month contains 30 days; a year contains 360 days. If the last date of the period is the last day of February, the month is extended to 30 days.
5	30/360 ISDA (International Swap Dealers Association)	Variant of 30/360 with slight differences for calculating number of days in a month.
6	30/360 European	Variant of 30/360 used primarily in Europe.
7	actual/365 Japanese	All years contain 365 days. Leap days are ignored.

Basis Value	Meaning	Description
8	actual/actual (ICMA)	Actual days held over actual days in coupon period. Denominator is 365 in most years and 366 in a leap year. This basis assumes an annual compounding period.
9	actual/360 (ICMA)	Actual days held over 360. This basis assumes an annual compounding period.
10	actual/365 (ICMA)	Actual days held over 365, even in leap years. This basis assumes an annual compounding period.
11	30/360E (ICMA)	The number of days in every month is set to 30. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360.
12	actual/365 (ISDA)	This day count fraction is equal to the sum of number of interest accrual days falling with a leap year divided by 366 and the number of interest accrual days not falling within a leap year divided by 365.
13	BUS/252	The number of business days between the previous coupon payment and the settlement date divided by 252. BUS/252 business days are non-weekend, non-holiday days. The

Basis Value	Meaning	Description
		holidays.m file defines holidays.

Note Although the concept of day count sounds deceptively simple, the actual calculation of day counts can be quite complex. You can find a good discussion of day counts and the formulas for calculating them in Chapter 5 of Stigum and Robinson, *Money Market and Bond Calculations* in Appendix A.

The *end-of-month rule* affects a bond's coupon payment structure. When the rule is in effect, a security that pays a coupon on the last actual day of a month will always pay coupons on the last day of the month. This means, for example, that a semiannual bond that pays a coupon on February 28 in nonleap years will pay coupons on August 31 in all years and on February 29 in leap years.

End-of-Month Rule

End-of-Month Rule Value	Meaning
1 (default)	Rule in effect.
0	Rule not in effect.

Framework

Although not all Financial Toolbox functions require the same input arguments, they all accept the following common set of input arguments.

Common Input Arguments

Input	Meaning
Settle	Settlement date
Maturity	Maturity date
Period	Coupon payment period
Basis	Day-count basis
EndMonthRule	End-of-month payment rule
IssueDate	Bond issue date

Input	Meaning
FirstCouponDate	First coupon payment date
LastCouponDate	Last coupon payment date

Of the common input arguments, only **Settle** and **Maturity** are required. All others are optional. They will be set to the default values if you do not explicitly set them. Note that, by default, the **FirstCouponDate** and **LastCouponDate** are nonapplicable. In other words, if you do not specify **FirstCouponDate** and **LastCouponDate**, the bond is assumed to have no odd first or last coupon periods. In this case, the bond is a standard bond with a coupon payment structure based solely on the maturity date.

Default Parameter Values

To illustrate the use of default values in Financial Toolbox functions, consider the **cfdates** function, which computes actual cash flow payment dates for a portfolio of fixed income securities regardless of whether the first and/or last coupon periods are normal, long, or short.

The complete calling syntax with the full input argument list is

```
CFlowDates = cfdates(Settle, Maturity, Period, Basis, ...
    EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

while the minimal calling syntax requires only settlement and maturity dates

```
CFlowDates = cfdates(Settle, Maturity)
```

Single Bond Example

As an example, suppose you have a bond with these characteristics

```
Settle           = '20-Sep-1999'
Maturity         = '15-Oct-2007'
Period          = 2
Basis            = 0
EndMonthRule    = 1
IssueDate       = NaN
FirstCouponDate = NaN
LastCouponDate  = NaN
```

Note that **Period**, **Basis**, and **EndMonthRule** are set to their default values, and **IssueDate**, **FirstCouponDate**, and **LastCouponDate** are set to NaN.

Formally, a NaN is an IEEE[®] arithmetic standard for *Not-a-Number* and is used to indicate the result of an undefined operation (for example, zero divided by zero). However, NaN is also a very convenient placeholder. In the SIA functions of Financial Toolbox software, NaN indicates the presence of a nonapplicable value. It tells the Financial Toolbox functions to ignore the input value and apply the default. Setting `IssueDate`, `FirstCouponDate`, and `LastCouponDate` to NaN in this example tells `cfdates` to assume that the bond has been issued before settlement and that no odd first or last coupon periods exist.

Having set these values, all these calls to `cfdates` produce the same result.

```
cfdates(Settle, Maturity)
cfdates(Settle, Maturity, Period)
cfdates(Settle, Maturity, Period, [])
cfdates(Settle, Maturity, [], Basis)
cfdates(Settle, Maturity, [], [])
cfdates(Settle, Maturity, Period, [], EndMonthRule)
cfdates(Settle, Maturity, Period, [], NaN)
cfdates(Settle, Maturity, Period, [], [], IssueDate)
cfdates(Settle, Maturity, Period, [], [], IssueDate, [], [])
cfdates(Settle, Maturity, Period, [], [], [], [], LastCouponDate)
cfdates(Settle, Maturity, Period, Basis, EndMonthRule, ...
IssueDate, FirstCouponDate, LastCouponDate)
```

Thus, leaving a particular input unspecified has the same effect as passing an empty matrix (`[]`) or passing a NaN – all three tell `cfdates` (and other Financial Toolbox functions) to use the default value for a particular input parameter.

Bond Portfolio Example

Since the previous example included only a single bond, there was no difference between passing an empty matrix or passing a NaN for an optional input argument. For a portfolio of bonds, however, using NaN as a placeholder is the only way to specify default acceptance for some bonds while explicitly setting nondefault values for the remaining bonds in the portfolio.

Now suppose you have a portfolio of two bonds.

```
Settle    = '20-Sep-1999'
Maturity  = ['15-Oct-2007'; '15-Oct-2010']
```

These calls to `cfdates` all set the coupon period to its default value (`Period = 2`) for both bonds.

```

cfdates(Settle, Maturity, 2)
cfdates(Settle, Maturity, [2 2])
cfdates(Settle, Maturity, [])
cfdates(Settle, Maturity, NaN)
cfdates(Settle, Maturity, [NaN NaN])
cfdates(Settle, Maturity)

```

The first two calls explicitly set `Period = 2`. Since `Maturity` is a 2-by-1 vector of maturity dates, `cfdates` knows you have a two-bond portfolio.

The first call specifies a single (that is, scalar) 2 for `Period`. Passing a scalar tells `cfdates` to apply the scalar-valued input to all bonds in the portfolio. This is an example of implicit scalar-expansion. Note that the settlement date has been implicit scalar-expanded as well.

The second call also applies the default coupon period by explicitly passing a two-element vector of 2's. The third call passes an empty matrix, which `cfdates` interprets as an invalid period, for which the default value will be used. The fourth call is similar, except that a NaN has been passed. The fifth call passes two NaN's, and has the same effect as the third. The last call passes the minimal input set.

Finally, consider the following calls to `cfdates` for the same two-bond portfolio.

```

cfdates(Settle, Maturity, [4 NaN])
cfdates(Settle, Maturity, [4 2])

```

The first call explicitly sets `Period = 4` for the first bond and implicitly sets the default `Period = 2` for the second bond. The second call has the same effect as the first but explicitly sets the periodicity for both bonds.

The optional input `Period` has been used for illustrative purpose only. The default-handling process illustrated in the examples applies to any of the optional input arguments.

Coupon Date Calculations

Calculating coupon dates, either actual or quasi dates, is notoriously complicated. Financial Toolbox software follows the SIA conventions in coupon date calculations.

The first step in finding the coupon dates associated with a bond is to determine the reference, or synchronization date (the *sync date*). Within the SIA framework, the order of precedence for determining the sync date is:

- 1 The first coupon date
- 2 The last coupon date
- 3 The maturity date

In other words, a Financial Toolbox function first examines the `FirstCouponDate` input. If `FirstCouponDate` is specified, coupon payment dates and quasi-coupon dates are computed with respect to `FirstCouponDate`; if `FirstCouponDate` is unspecified, empty (`[]`), or `NaN`, then the `LastCouponDate` is examined. If `LastCouponDate` is specified, coupon payment dates and quasi-coupon dates are computed with respect to `LastCouponDate`. If both `FirstCouponDate` and `LastCouponDate` are unspecified, empty (`[]`), or `NaN`, the `Maturity` (a required input argument) serves as the sync date.

Yield Conventions

There are two yield and time factor conventions that are used in the Financial Toolbox software – these are determined by the input `basis`. Specifically, bases 0 to 7 are assumed to have semiannual compounding, while bases 8 to 12 are assumed to have annual compounding regardless of the period of the bond's coupon payments (including zero-coupon bonds). In addition, any yield-related sensitivity (that is, duration and convexity), when quoted on a periodic basis, follows this same convention. (See `bndconvp`, `bndconvy`, `bnddurp`, `bnddury`, and `bndkrdur`.)

Pricing Functions

This example shows how easily you can compute the price of a bond with an odd first period using the function `bndprice`. Assume you have a bond with these characteristics:

```
Settle           = '11-Nov-1992';  
Maturity         = '01-Mar-2005';  
IssueDate       = '15-Oct-1992';  
FirstCouponDate = '01-Mar-1993';  
CouponRate      = 0.0785;  
Yield           = 0.0625;
```

Allow coupon payment period (`Period = 2`), day-count basis (`Basis = 0`), and end-of-month rule (`EndMonthRule = 1`) to assume the default values. Also, assume there is no odd last coupon date and that the face value of the bond is \$100. Calling the function

```
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, ...
```

Maturity, [], [], [], IssueDate, FirstCouponDate)

returns a price of \$113.60 and accrued interest of \$0.59.

Similar functions compute prices with regular payments, odd first and last periods, and prices of Treasury bills and discounted securities such as zero-coupon bonds.

Note `bndprice` and other functions use nonlinear formulas to compute the price of a security. For this reason, Financial Toolbox software uses Newton's method when solving for an independent variable within a formula. See any elementary numerical methods textbook for the mathematics underlying Newton's method.

Yield Functions

To illustrate toolbox yield functions, compute the yield of a bond that has odd first and last periods and settlement in the first period. First set up variables for settlement, maturity date, issue, first coupon, and a last coupon date.

```
Settle          = '12-Jan-2000';
Maturity        = '01-Oct-2001';
IssueDate       = '01-Jan-2000';
FirstCouponDate = '15-Jan-2000';
LastCouponDate  = '15-Apr-2000';
```

Assume a face value of \$100. Specify a purchase price of \$95.70, a coupon rate of 4%, quarterly coupon payments, and a 30/360 day-count convention (`Basis = 1`).

```
Price          = 95.7;
CouponRate     = 0.04;
Period         = 4;
Basis          = 1;
EndMonthRule   = 1;
```

Calling the function

```
Yield = bndyield(Price, CouponRate, Settle, Maturity, Period, ...
Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

returns

Yield = 0.0659 (6.60%).

Fixed-Income Sensitivities

Financial Toolbox software supports the following options for managing interest-rate risk for one or more bonds:

- `bnddurp` and `bnddury` support duration and convexity analysis based on market quotes and assume parallel shifts in the bond yield curve.
- `bndkrdur` supports key rate duration based on a market yield curve and can model nonparallel shifts in the bond yield curve.

Calculating Duration and Convexity for Bonds

The toolbox includes functions to perform sensitivity analysis such as convexity and the Macaulay and modified durations for fixed-income securities. The Macaulay duration of an income stream, such as a coupon bond, measures how long, on average, the owner waits before receiving a payment. It is the weighted average of the times payments are made, with the weights at time T equal to the present value of the money received at time T . The modified duration is the Macaulay duration discounted by the per-period interest rate; that is, divided by $(1 + \text{rate}/\text{frequency})$.

To illustrate, the following example computes the annualized Macaulay and modified durations, and the periodic Macaulay duration for a bond with settlement (12-Jan-2000) and maturity (01-Oct-2001) dates as above, a 5% coupon rate, and a 4.5% yield to maturity. For simplicity, any optional input arguments assume default values (that is, semiannual coupons, and day-count `basis = 0` (actual/actual), coupon payment structure synchronized to the maturity date, and end-of-month payment rule in effect).

```
CouponRate = 0.05;  
Yield = 0.045;
```

```
[ModDuration, YearDuration, PerDuration] = bnddury(Yield,...  
CouponRate, Settle, Maturity)
```

The durations are

```
ModDuration = 1.6107 (years)  
YearDuration = 1.6470 (years)  
PerDuration = 3.2940 (semiannual periods)
```

Note that the semiannual periodic Macaulay duration (`PerDuration`) is twice the annualized Macaulay duration (`YearDuration`).

Calculating Key Rate Durations for Bonds

Key rate duration enables you to evaluate the sensitivity and price of a bond to nonparallel changes in the spot or zero curve by decomposing the interest rate risk along the spot or zero curve. Key rate duration refers to the process of choosing a set of key rates and computing a duration for each rate. Specifically, for each key rate, while the other rates are held constant, the key rate is shifted up and down (and intermediate cash flow dates are interpolated), and then the present value of the security given the shifted curves is computed.

The calculation of `bndkrdur` supports:

$$krdur_i = \frac{(PV_{down} - PV_{up})}{(PV \times ShiftValue \times 2)}$$

Where *PV* is the current value of the instrument, *PV_{up}* and *PV_{down}* are the new values after the discount curve has been shocked, and *ShiftValue* is the change in interest rate. For example, if key rates of 3 months, 1, 2, 3, 5, 7, 10, 15, 20, 25, 30 years were chosen, then a 30-year bond might have corresponding key rate durations of:

3M	1Y	2Y	3Y	5Y	7Y	10Y	15Y	20Y	25Y	30Y
.01	.04	.09	.21	.4	.65	1.27	1.71	1.68	1.83	7.03

The key rate durations add up to approximately equal the duration of the bond.

For example, compute the key rate duration of the U.S. Treasury Bond with maturity date of August 15, 2028 and coupon rate of 5.5%.

```
Settle = datenum('18-Nov-2008');
CouponRate = 5.500/100;
Maturity = datenum('15-Aug-2028');
Price = 114.83;
```

For the `ZeroData` information on the current spot curve for this bond, refer to <http://www.treas.gov/offices/domestic-finance/debt-management/interest-rate/yield.shtml>:

```
ZeroDates = daysadd(Settle ,[30 90 180 360 360*2 360*3 360*5 ...
360*7 360*10 360*20 360*30]);
ZeroRates = ([0.06 0.12 0.81 1.08 1.22 1.53 2.32 2.92 3.68 4.42 4.20]/100)';
```

Compute the key rate duration for a specific set of rates (choose this based on the maturities of the available hedging instruments):

```
krd = bndkrdur([ZeroDates ZeroRates],CouponRate,Settle,Maturity,'keyrates',[2 5 10 20])
```

```
krd =
```

```
    0.2865    0.8729    2.6451    8.5778
```

Note, the sum of the key rate durations approximately equals the duration of the bond:

```
[sum(krd) bnddur(Price,CouponRate,Settle,Maturity)]
```

```
ans =
```

```
    12.3823    12.3919
```

Treasury Bills Defined

Treasury bills are short-term securities (issued with maturities of 1 year or less) sold by the United States Treasury. Sales of these securities are frequent, usually weekly. From time to time, the Treasury also offers longer duration securities called Treasury notes and Treasury bonds.

A Treasury bill is a discount security. The holder of the Treasury bill does not receive periodic interest payments. Instead, at the time of sale, a percentage discount is applied to the face value. At maturity, the holder redeems the bill for full face value.

The basis for Treasury bill interest calculation is actual/360. Under this system, interest accrues on the actual number of elapsed days between purchase and maturity, and each year contains 360 days.

Computing Treasury Bill Price and Yield

In this section...
“Introduction” on page 2-34
“Treasury Bill Repurchase Agreements” on page 2-34
“Treasury Bill Yields” on page 2-36

Introduction

Financial Toolbox software provides the following suite of functions for computing price and yield on Treasury bills.

Treasury Bill Functions

Function	Purpose
tbilldisc2yield	Convert discount rate to yield.
tbillprice	Price Treasury bill given its yield or discount rate.
tbillrepo	Break-even discount of repurchase agreement.
tbillyield	Yield and discount of Treasury bill given its price.
tbillyield2disc	Convert yield to discount rate.
tbillval01	The value of 1 basis point given the characteristics of the Treasury bill, as represented by its settlement and maturity dates. You can relate the basis point to discount, money-market, or bond-equivalent yield.

For all functions with yield in the computation, you can specify yield as money-market or bond-equivalent yield. The functions all assume a face value of \$100 for each Treasury bill.

Treasury Bill Repurchase Agreements

The following example shows how to compute the break-even discount rate. This is the rate that correctly prices the Treasury bill such that the profit from selling the bill equals 0.

```
Maturity = '26-Dec-2002';
```

```

InitialDiscount = 0.0161;
PurchaseDate = '26-Sep-2002';
SaleDate = '26-Oct-2002';
RepoRate = 0.0149;

BreakevenDiscount = tbillrepo(RepoRate, InitialDiscount, ...
PurchaseDate, SaleDate, Maturity)

BreakevenDiscount =

    0.0167

```

You can check the result of this computation by examining the cash flows in and out from the repurchase transaction. First compute the price of the Treasury bill on the purchase date (September 26).

```

PriceOnPurchaseDate = tbillprice(InitialDiscount, ...
PurchaseDate, Maturity, 3)

PriceOnPurchaseDate =

    99.5930

```

Next compute the interest due on the repurchase agreement.

```

RepoInterest = ...
RepoRate*PriceOnPurchaseDate*days360(PurchaseDate, SaleDate) / 360

RepoInterest =

    0.1237

```

RepoInterest for a 1.49% 30-day term repurchase agreement (30/360 basis) is 0.1237.

Finally, compute the price of the Treasury bill on the sale date (October 26).

```

PriceOnSaleDate = tbillprice(BreakevenDiscount, SaleDate, ...
Maturity, 3)

PriceOnSaleDate =

    99.7167

```

Examining the cash flows, observe that the break-even discount causes the sum of the price on the purchase date plus the accrued 30-day interest to be equal to the price on sale date. The next table shows the cash flows.

Cash Flows from Repurchase Agreement

Date	Cash Out Flow		Cash In Flow	
9/26/2002	Purchase T-bill	99.593	Repo money	99.593
10/26/2002	Payment of repo	99.593	Sell T-bill	99.7168
	Repo interest	0.1238		
	Total	199.3098		199.3098

Treasury Bill Yields

Using the same data as before, you can examine the money-market and bond-equivalent yields of the Treasury bill at the time of purchase and sale. The function `tbilldisc2yield` can perform both computations at one time.

```
Maturity = '26-Dec-2002';
InitialDiscount = 0.0161;
PurchaseDate = '26-Sep-2002';
SaleDate = '26-Oct-2002';
RepoRate = 0.0149;
BreakevenDiscount = tbillrepo(RepoRate, InitialDiscount, ...
PurchaseDate, SaleDate, Maturity)
```

```
[BEYield, MMYield] = ...
tbilldisc2yield([InitialDiscount; BreakevenDiscount], ...
[PurchaseDate; SaleDate], Maturity)
```

```
BreakevenDiscount =
```

```
    0.0167
```

```
BEYield =
```

```
    0.0164
    0.0170
```

```
MMYield =
```

```
    0.0162
    0.0168
```

For the short Treasury bill (fewer than 182 days to maturity), the money-market yield is $360/365$ of the bond-equivalent yield, as this example shows.

Term Structure of Interest Rates

In this section...

“Introduction” on page 2-38

“Deriving an Implied Zero Curve” on page 2-39

Introduction

The Financial Toolbox product contains several functions to derive and analyze interest rate curves, including data conversion and extrapolation, bootstrapping, and interest-rate curve conversion functions.

One of the first problems in analyzing the term structure of interest rates is dealing with market data reported in different formats. Treasury bills, for example, are quoted with bid and asked bank-discount rates. Treasury notes and bonds, on the other hand, are quoted with bid and asked prices based on \$100 face value. To examine the full spectrum of Treasury securities, analysts must convert data to a single format. Financial Toolbox functions ease this conversion. This brief example uses only one security each; analysts often use 30, 100, or more of each.

First, capture Treasury bill quotes in their reported format

```
%      Maturity      Days Bid   Ask   AskYield
TBill = [datenum('12/26/2000') 53   0.0503  0.0499  0.0510];
```

then capture Treasury bond quotes in their reported format

```
%      Coupon  Maturity      Bid   Ask   AskYield
TBond = [0.08875  datenum(2001,11,5) 103+4/32  103+6/32  0.0564];
```

and note that these quotes are based on a November 3, 2000 settlement date.

```
Settle = datenum('3-Nov-2000');
```

Next use the toolbox `tb12bond` function to convert the Treasury bill data to Treasury bond format.

```
TBTBond = tb12bond(TBill)
```

```
TBTBond =
    0    730846    99.26    99.27    0.05
```


(The second element of `TBTBond` is the serial date number for December 26, 2000.)

Now combine short-term (Treasury bill) with long-term (Treasury bond) data to set up the overall term structure.

```
TBondsAll = [TBTBond; TBond]
```

```
TBondsAll =
      0      730846      99.26      99.27      0.05
    0.09      731160     103.13     103.19      0.06
```

The Financial Toolbox software provides a second data-preparation function, `tr2bonds`, to convert the bond data into a form ready for the bootstrapping functions. `tr2bonds` generates a matrix of bond information sorted by maturity date, plus vectors of prices and yields.

```
[Bonds, Prices, Yields] = tr2bonds(TBondsAll);
```

Deriving an Implied Zero Curve

Using this market data, you can use one of the Financial Toolbox bootstrapping functions to derive an implied zero curve. Bootstrapping is a process whereby you begin with known data points and solve for unknown data points using an underlying arbitrage theory. Every coupon bond can be valued as a package of zero-coupon bonds which mimic its cash flow and risk characteristics. By mapping yields-to-maturity for each theoretical zero-coupon bond, to the dates spanning the investment horizon, you can create a theoretical zero-rate curve. The Financial Toolbox software provides two bootstrapping functions: `zbtprice` derives a zero curve from bond data and *prices*, and `zbtyield` derives a zero curve from bond data and *yields*. Using `zbtprice`

```
[ZeroRates, CurveDates] = zbtprice(Bonds, Prices, Settle)
```

```
ZeroRates =
```

```
    0.05
    0.06
```

```
CurveDates =
```

```
    730846
    731160
```

`CurveDates` gives the investment horizon.

```
datestr(CurveDates)
```

```
ans =
```

```
26-Dec-2000
```

```
05-Nov-2001
```

Additional Financial Toolbox functions construct discount, forward, and par yield curves from the zero curve, and vice versa.

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates,...  
Settle);  
[FwdRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, Settle);  
[PYldRates, CurveDates] = zero2pyld(ZeroRates, CurveDates,...  
Settle);
```

Pricing and Analyzing Equity Derivatives

In this section...

“Introduction” on page 2-41

“Sensitivity Measures” on page 2-41

“Analysis Models” on page 2-42

Introduction

These toolbox functions compute prices, sensitivities, and profits for portfolios of options or other equity derivatives. They use the Black-Scholes model for European options and the binomial model for American options. Such measures are useful for managing portfolios and for executing collars, hedges, and straddles.

Sensitivity Measures

There are six basic sensitivity measures associated with option pricing: delta, gamma, lambda, rho, theta, and vega — the “greeks.” The toolbox provides functions for calculating each sensitivity and for implied volatility.

Delta

Delta of a derivative security is the rate of change of its price relative to the price of the underlying asset. It is the first derivative of the curve that relates the price of the derivative to the price of the underlying security. When delta is large, the price of the derivative is sensitive to small changes in the price of the underlying security.

Gamma

Gamma of a derivative security is the rate of change of delta relative to the price of the underlying asset; that is, the second derivative of the option price relative to the security price. When gamma is small, the change in delta is small. This sensitivity measure is important for deciding how much to adjust a hedge position.

Lambda

Lambda, also known as the elasticity of an option, represents the percentage change in the price of an option relative to a 1% change in the price of the underlying security.

Rho

Rho is the rate of change in option price relative to the risk-free interest rate.

Theta

Theta is the rate of change in the price of a derivative security relative to time. Theta is usually very small or negative since the value of an option tends to drop as it approaches maturity.

Vega

Vega is the rate of change in the price of a derivative security relative to the volatility of the underlying security. When vega is large the security is sensitive to small changes in volatility. For example, options traders often must decide whether to buy an option to hedge against vega or gamma. The hedge selected usually depends upon how frequently one rebalances a hedge position and also upon the standard deviation of the price of the underlying asset (the volatility). If the standard deviation is changing rapidly, balancing against vega is usually preferable.

Implied Volatility

The implied volatility of an option is the standard deviation that makes an option price equal to the market price. It helps determine a market estimate for the future volatility of a stock and provides the input volatility (when needed) to the other Black-Scholes functions.

Analysis Models

Toolbox functions for analyzing equity derivatives use the Black-Scholes model for European options and the binomial model for American options. The Black-Scholes model makes several assumptions about the underlying securities and their behavior. The binomial model, on the other hand, makes far fewer assumptions about the processes underlying an option. For further explanation, see *Options, Futures, and Other Derivatives* by John Hull in Appendix A.

Black-Scholes Model

Using the Black-Scholes model entails several assumptions:

- The prices of the underlying asset follow an Ito process. (See Hull, page 222.)
- The option can be exercised only on its expiration date (European option).

- Short selling is permitted.
- There are no transaction costs.
- All securities are divisible.
- There is no riskless arbitrage.
- Trading is a continuous process.
- The risk-free interest rate is constant and remains the same for all maturities.

If any of these assumptions is untrue, Black-Scholes may not be an appropriate model.

To illustrate toolbox Black-Scholes functions, this example computes the call and put prices of a European option and its delta, gamma, lambda, and implied volatility. The asset price is \$100.00, the exercise price is \$95.00, the risk-free interest rate is 10%, the time to maturity is 0.25 years, the volatility is 0.50, and the dividend rate is 0. Simply executing the toolbox functions

```
[OptCall, OptPut] = blsprice(100, 95, 0.10, 0.25, 0.50, 0);
[CallVal, PutVal] = blsdelta(100, 95, 0.10, 0.25, 0.50, 0);
GammaVal = blsgamma(100, 95, 0.10, 0.25, 0.50, 0);
VegaVal = blsvega(100, 95, 0.10, 0.25, 0.50, 0);
[LamCall, LamPut] = blslambda(100, 95, 0.10, 0.25, 0.50, 0);
```

yields:

- The option call price `OptCall` = \$13.70
- The option put price `OptPut` = \$6.35
- delta for a call `CallVal` = 0.6665 and delta for a put `PutVal` = -0.3335
- gamma `GammaVal` = 0.0145
- vega `VegaVal` = 18.1843
- lambda for a call `LamCall` = 4.8664 and lambda for a put `LamPut` = -5.2528

Now as a computation check, find the implied volatility of the option using the call option price from `blsprice`.

```
Volatility = blsimpv(100, 95, 0.10, 0.25, OptCall);
```

The function returns an implied volatility of 0.500, the original `blsprice` input.

Binomial Model

The binomial model for pricing options or other equity derivatives assumes that the probability over time of each possible price follows a binomial distribution. The basic

assumption is that prices can move to only two values, one up and one down, over any short time period. Plotting the two values, and then the subsequent two values each, and then the subsequent two values each, and so on over time, is known as “building a binomial tree.” This model applies to American options, which can be exercised any time up to and including their expiration date.

This example prices an American call option using a binomial model. Again, the asset price is \$100.00, the exercise price is \$95.00, the risk-free interest rate is 10%, and the time to maturity is 0.25 years. It computes the tree in increments of 0.05 years, so there are $0.25/0.05 = 5$ periods in the example. The volatility is 0.50, this is a call (`flag = 1`), the dividend rate is 0, and it pays a dividend of \$5.00 after three periods (an ex-dividend date). Executing the toolbox function

```
[StockPrice, OptionPrice] = binprice(100, 95, 0.10, 0.25, ...
0.05, 0.50, 1, 0, 5.0, 3);
```

returns the tree of prices of the underlying asset

StockPrice =

100.00	111.27	123.87	137.96	148.69	166.28
0	89.97	100.05	111.32	118.90	132.96
0	0	81.00	90.02	95.07	106.32
0	0	0	72.98	76.02	85.02
0	0	0	0	60.79	67.98
0	0	0	0	0	54.36

and the tree of option values.

OptionPrice =

12.10	19.17	29.35	42.96	54.17	71.28
0	5.31	9.41	16.32	24.37	37.96
0	0	1.35	2.74	5.57	11.32
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

The output from the binomial function is a binary tree. Read the `StockPrice` matrix this way: column 1 shows the price for period 0, column 2 shows the up and down prices for period 1, column 3 shows the up-up, up-down, and down-down prices for period 2, and so on. Ignore the zeros. The `OptionPrice` matrix gives the associated option value for each node in the price tree. Ignore the zeros that correspond to a zero in the price tree.

Portfolio Analysis

- “Analyzing Portfolios” on page 3-2
- “Portfolio Optimization Functions” on page 3-3
- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion” on page 3-8
- “Constraint Specification” on page 3-12
- “Active Returns and Tracking Error Efficient Frontier” on page 3-19

Analyzing Portfolios

Portfolio managers concentrate their efforts on achieving the best possible trade-off between risk and return. For portfolios constructed from a fixed set of assets, the risk/return profile varies with the portfolio composition. Portfolios that maximize the return, given the risk, or, conversely, minimize the risk for the given return, are called *optimal*. Optimal portfolios define a line in the risk/return plane called the *efficient frontier*.

A portfolio may also have to meet additional requirements to be considered. Different investors have different levels of risk tolerance. Selecting the adequate portfolio for a particular investor is a difficult process. The portfolio manager can hedge the risk related to a particular portfolio along the efficient frontier with partial investment in risk-free assets. The definition of the capital allocation line, and finding where the final portfolio falls on this line, if at all, is a function of:

- The risk/return profile of each asset
- The risk-free rate
- The borrowing rate
- The degree of risk aversion characterizing an investor

Financial Toolbox software includes a set of portfolio optimization functions designed to find the portfolio that best meets investor requirements.

Portfolio Optimization Functions

The portfolio optimization functions assist portfolio managers in constructing portfolios that optimize risk and return.

Capital Allocation	Description
<code>portalloc</code>	Computes the optimal risky portfolio on the efficient frontier, based on the risk-free rate, the borrowing rate, and the investor's degree of risk aversion. Also generates the capital allocation line, which provides the optimal allocation of funds between the risky portfolio and the risk-free asset.
Efficient Frontier Computation	Description
<code>frontcon</code>	Computes portfolios along the efficient frontier for a given group of assets. The computation is based on sets of constraints representing the maximum and minimum weights for each asset, and the maximum and minimum total weight for specified groups of assets.
<code>frontier</code>	Computes portfolios along the efficient frontier for a given group of assets. Generates a surface of efficient frontiers showing how asset allocation influences risk and return over time.
<code>portopt</code>	Computes portfolios along the efficient frontier for a given group of assets. The computation is based on a set of user-specified linear constraints. Typically, these constraints are generated using the constraint specification functions described below.
Constraint Specification	Description
<code>portcons</code>	Generates the portfolio constraints matrix for a portfolio of asset investments using linear inequalities. The inequalities are of the type $A \cdot Wts \leq b$, where Wts is a row vector of weights.
<code>portvrisk</code>	Portfolio value at risk (VaR) returns the maximum potential loss in the value of a portfolio over one period of time, given the loss probability level <code>RiskThreshold</code> .
<code>pcalims</code>	Asset minimum and maximum allocation. Generates a constraint set to fix the minimum and maximum weight for each individual asset.

Constraint Specification	Description
pcgcomp	Group-to-group ratio constraint. Generates a constraint set specifying the maximum and minimum ratios between pairs of groups.
pcglims	Asset group minimum and maximum allocation. Generates a constraint set to fix the minimum and maximum total weight for each defined group of assets.
pcpval	Total portfolio value. Generates a constraint set to fix the total value of the portfolio.

Constraint Conversion	Description
abs2active	Transforms a constraint matrix expressed in absolute weight format to an equivalent matrix expressed in active weight format.
active2abs	Transforms a constraint matrix expressed in active weight format to an equivalent matrix expressed in absolute weight format.

Note: An alternative to using these portfolio optimization functions is to use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Portfolio Construction Examples

In this section...

“Introduction” on page 3-5

“Efficient Frontier Example” on page 3-5

Introduction

The efficient frontier computation functions require information about each asset in the portfolio. This data is entered into the function via two matrices: an expected return vector and a covariance matrix. The expected return vector contains the average expected return for each asset in the portfolio. The covariance matrix is a square matrix representing the interrelationships between pairs of assets. This information can be directly specified or can be estimated from an asset return time series with the function `ewstats`.

Efficient Frontier Example

This example computes the efficient frontier of portfolios consisting of three different assets using the function `frontcon`. To visualize the efficient frontier curve clearly, consider 10 different evenly spaced portfolios.

Assume that the expected return of the first asset is 10%, the second is 20%, and the third is 15%. The covariance is defined in the matrix `ExpCovariance`.

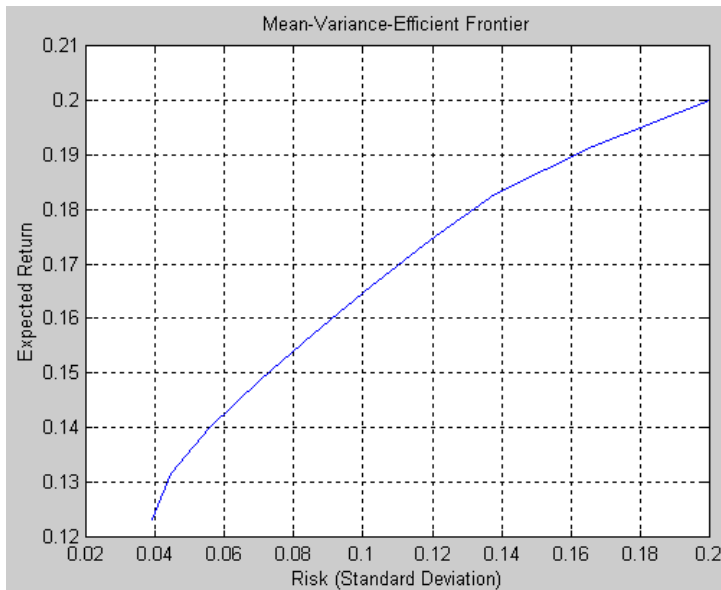
```
ExpReturn = [0.1 0.2 0.15];
```

```
ExpCovariance = [ 0.005  -0.010  0.004;
                 -0.010  0.040  -0.002;
                 0.004  -0.002  0.023];
```

```
NumPorts = 10;
```

Since there are no constraints, you can call `frontcon` directly with the data you already have. If you call `frontcon` without specifying any output arguments, you get a graph representing the efficient frontier curve.

```
frontcon (ExpReturn, ExpCovariance, NumPorts);
```



Calling `frontcon` while specifying the output arguments returns the corresponding vectors and arrays representing the risk, return, and weights for each of the 10 points computed along the efficient frontier.

```
[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn,...  
ExpCovariance, NumPorts)
```

```
PortRisk =  
    0.0392  
    0.0445  
    0.0559  
    0.0701  
    0.0858  
    0.1023  
    0.1192  
    0.1383  
    0.1661  
    0.2000
```

```
PortReturn =  
  
    0.1231  
    0.1316
```

0.1402
0.1487
0.1573
0.1658
0.1744
0.1829
0.1915
0.2000

PortWts =

0.7692	0.2308	0.0000
0.6667	0.2991	0.0342
0.5443	0.3478	0.1079
0.4220	0.3964	0.1816
0.2997	0.4450	0.2553
0.1774	0.4936	0.3290
0.0550	0.5422	0.4027
0	0.6581	0.3419
0	0.8291	0.1709
0	1.0000	0.0000

The output data is represented row-wise. Each portfolio's risk, rate of return, and associated weights are identified as corresponding rows in the vectors and matrix.

For example, you can see from these results that the second portfolio has a risk of 0.0445, an expected return of 13.16%, and allocations of about 67% in the first asset, 30% in the second, and 3% in the third.

Portfolio Selection and Risk Aversion

In this section...
“Introduction” on page 3-8
“Optimal Risky Portfolio” on page 3-9

Introduction

One of the factors to consider when selecting the optimal portfolio for a particular investor is degree of risk aversion. This level of aversion to risk can be characterized by defining the investor's indifference curve. This curve consists of the family of risk/return pairs defining the trade-off between the expected return and the risk. It establishes the increment in return that a particular investor will require in order to make an increment in risk worthwhile. Typical risk aversion coefficients range between 2.0 and 4.0, with the higher number representing lesser tolerance to risk. The equation used to represent risk aversion in Financial Toolbox software is

$$U = E(r) - 0.005 * A * sig^2$$

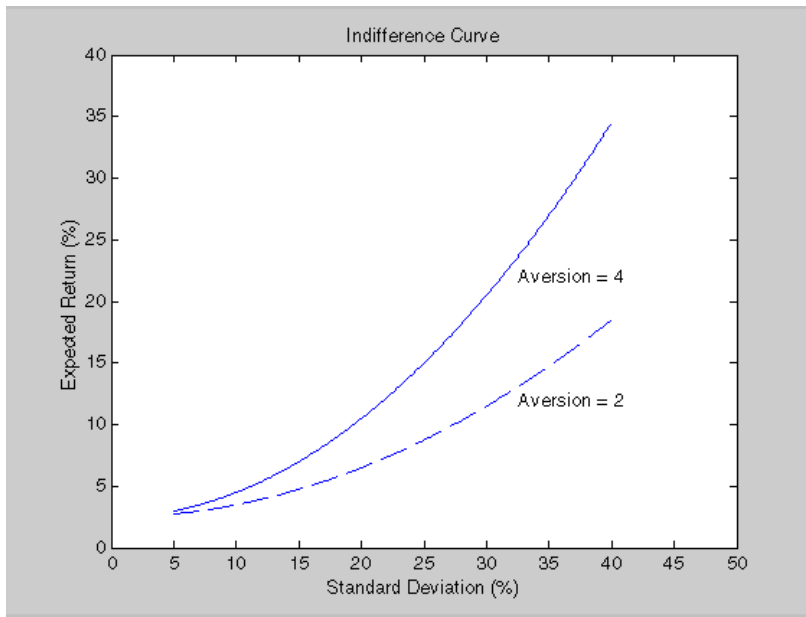
where:

U is the utility value.

E(r) is the expected return.

A is the index of investor's aversion.

sig is the standard deviation.



Optimal Risky Portfolio

This example computes the optimal risky portfolio on the efficient frontier based upon the risk-free rate, the borrowing rate, and the investor's degree of risk aversion. You do this with the function `portalloc`.

First generate the efficient frontier data using either `portopt` or `frontcon`. This example uses `portopt` and the same asset data from the previous example.

```
ExpReturn = [0.1 0.2 0.15];
```

```
ExpCovariance = [ 0.005  -0.010  0.004;
                 -0.010  0.040  -0.002;
                 0.004  -0.002  0.023];
```

This time consider 20 different points along the efficient frontier.

```
NumPorts = 20;
```

```
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn,...
ExpCovariance, NumPorts);
```

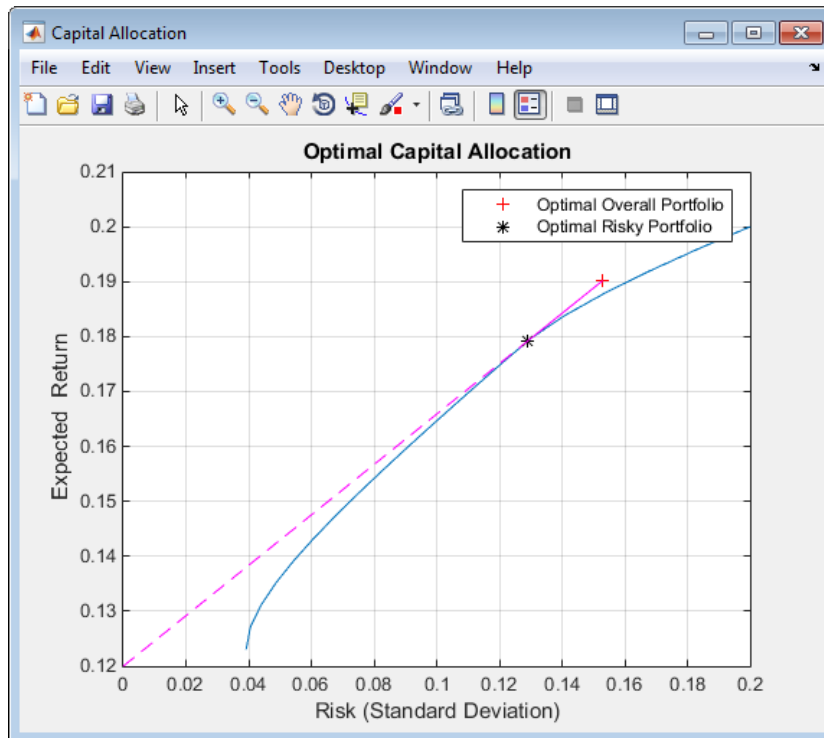
As with `frontcon`, calling `portopt` while specifying output arguments returns the corresponding vectors and arrays representing the risk, return, and weights for each of the portfolios along the efficient frontier. Use them as the first three input arguments to the function `portalloc`.

Now find the optimal risky portfolio and the optimal allocation of funds between the risky portfolio and the risk-free asset, using these values for the risk-free rate, borrowing rate and investor's degree of risk aversion.

```
RisklessRate = 0.08
BorrowRate   = 0.12
RiskAversion  = 3
```

Calling `portalloc` without specifying any output arguments gives a graph displaying the critical points.

```
portalloc (PortRisk, PortReturn, PortWts, RisklessRate, ...
BorrowRate, RiskAversion);
```



Calling `portalloc` while specifying the output arguments returns the variance (`RiskyRisk`), the expected return (`RiskyReturn`), and the weights (`RiskyWts`) allocated to the optimal risky portfolio. It also returns the fraction (`RiskyFraction`) of the complete portfolio allocated to the risky portfolio, and the variance (`OverallRisk`) and expected return (`OverallReturn`) of the optimal overall portfolio. The overall portfolio combines investments in the risk-free asset and in the risky portfolio. The actual proportion assigned to each of these two investments is determined by the degree of risk aversion characterizing the investor.

```
[RiskyRisk, RiskyReturn, RiskyWts, RiskyFraction, OverallRisk, ...  
OverallReturn] = portalloc (PortRisk, PortReturn, PortWts, ...  
RisklessRate, BorrowRate, RiskAversion)
```

```
RiskyRisk = 0.1288  
RiskyReturn = 0.1791  
RiskyWts = 0.0057 0.5879 0.4064  
RiskyFraction = 1.1869  
OverallRisk = 0.1529  
OverallReturn = 0.1902
```

The value of `RiskyFraction` exceeds 1 (100%), implying that the risk tolerance specified allows borrowing money to invest in the risky portfolio, and that no money will be invested in the risk-free asset. This borrowed capital is added to the original capital available for investment. In this example the customer will tolerate borrowing 18.69% of the original capital amount.

Constraint Specification

In this section...

“Example” on page 3-12

“Linear Constraint Equations” on page 3-14

“Specifying Additional Constraints” on page 3-17

Example

This example computes the efficient frontier of portfolios consisting of three different assets, INTC, XON, and RD, given a list of constraints. The expected returns for INTC, XON, and RD are respectively as follows:

```
ExpReturn = [0.1 0.2 0.15];
```

The covariance matrix is

```
ExpCovariance = [ 0.005  -0.010  0.004;
                  -0.010  0.040  -0.002;
                  0.004  -0.002  0.023];
```

- Constraint 1
 - Allow short selling up to 10% of the portfolio value in any asset, but limit the investment in any one asset to 110% of the portfolio value.
- Constraint 2
 - Consider two different sectors, technology and energy, with the following table indicating the sector each asset belongs to.

Asset	INTC	XON	RD
Sector	Technology	Energy	Energy

Constrain the investment in the Energy sector to 80% of the portfolio value, and the investment in the Technology sector to 70%.

To solve this problem, use `frontcon`, passing in a list of asset constraints. Consider eight different portfolios along the efficient frontier:

```
NumPorts = 8;
```

To introduce the asset bounds constraints specified in Constraint 1, create the matrix `AssetBounds`, where each column represents an asset. The upper row represents the lower bounds, and the lower row represents the upper bounds.

```
AssetBounds = [-0.10, -0.10, -0.10;
               1.10,  1.10,  1.10];
```

Constraint 2 needs to be entered in two parts, the first part defining the groups, and the second part defining the constraints for each group. Given the information above, you can build a matrix of 1s and 0s indicating whether a specific asset belongs to a group. Each column represents an asset, and each row represents a group. This example has two groups: the technology group, and the energy group. Create the matrix `Groups` as follows.

```
Groups = [0  1  1;
          1  0  0];
```

The `GroupBounds` matrix allows you to specify an upper and lower bound for each group. Each row in this matrix represents a group. The first column represents the minimum allocation, and the second column represents the maximum allocation to each group. Since the investment in the Energy sector is capped at 80% of the portfolio value, and the investment in the Technology sector is capped at 70%, create the `GroupBounds` matrix using this information.

```
GroupBounds = [0  0.80;
               0  0.70];
```

Now use `frontcon` to obtain the vectors and arrays representing the risk, return, and weights for each of the eight portfolios computed along the efficient frontier.

```
[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn,...
ExpCovariance, NumPorts, [], AssetBounds, Groups, GroupBounds)
```

```
PortRisk =
    0.0416
    0.0499
    0.0624
    0.0767
    0.0920
    0.1100
    0.1378
```

```
0.1716
PortReturn =
0.1279
0.1361
0.1442
0.1524
0.1605
0.1687
0.1768
0.1850
PortWts =
0.7000 0.2582 0.0418
0.6031 0.3244 0.0725
0.4864 0.3708 0.1428
0.3696 0.4172 0.2132
0.2529 0.4636 0.2835
0.2000 0.5738 0.2262
0.2000 0.7369 0.0631
0.2000 0.9000 -0.1000
```

The output data is represented row-wise, where each portfolio's risk, rate of return, and associated weight is identified as corresponding rows in the vectors and matrix.

Linear Constraint Equations

While `frontcon` allows you to enter a fixed set of constraints related to minimum and maximum values for groups and individual assets, you often need to specify a larger and more general set of constraints when finding the optimal risky portfolio. The function `portopt` addresses this need, by accepting an arbitrary set of constraints as an input matrix.

The auxiliary function `portcons` can be used to create the matrix of constraints, with each row representing an inequality. These inequalities are of the type $A * Wts' \leq b$, where A is a matrix, b is a vector, and Wts is a row vector of asset allocations. The number of columns of the matrix A , and the length of the vector Wts correspond to the number of assets. The number of rows of the matrix A , and the length of vector b

correspond to the number of constraints. This method allows you to specify any number of linear inequalities to the function `portopt`.

In actuality, `portcons` is an entry point to a set of functions that generate matrices for specific types of constraints. `portcons` allows you to specify all the constraints data at once, while the specific portfolio constraint functions allow you to build the constraints incrementally. These constraint functions are `pcpval`, `pcalims`, `pcglims`, and `pcgcomp`.

Consider an example to help understand how to specify constraints to `portopt` while bypassing the use of `portcons`. This example requires specifying the minimum and maximum investment in various groups.

Maximum and Minimum Group Exposure

Group	Minimum Exposure	Maximum Exposure
North America	0.30	0.75
Europe	0.10	0.55
Latin America	0.20	0.50
Asia	0.50	0.50

Note that the minimum and maximum exposure in Asia is the same. This means that you require a fixed exposure for this group.

Also assume that the portfolio consists of three different funds. The correspondence between funds and groups is shown in the table below.

Group Membership

Group	Fund 1	Fund 2	Fund 3
North America	X	X	
Europe			X
Latin America	X		
Asia		X	X

Using the information in these two tables, build a mathematical representation of the constraints represented. Assume that the vector of weights representing the exposure of each asset in a portfolio is called $Wts = [W1 \ W2 \ W3]$.

Specifically

1.	$W1 + W2$	\geq	0.30
2.	$W1 + W2$	\leq	0.75
3.	$W3$	\geq	0.10
4.	$W3$	\leq	0.55
5.	$W1$	\geq	0.20
6.	$W1$	\leq	0.50
7.	$W2 + W3$	$=$	0.50

Since you need to represent the information in the form $A \cdot W \leq b$, multiply equations 1, 3 and 5 by -1 . Also turn equation 7 into a set of two inequalities: $W2 + W3 \geq 0.50$ and $W2 + W3 \leq 0.50$. (The intersection of these two inequalities is the equality itself.) Thus

1.	$-W1 - W2$	\leq	-0.30
2.	$W1 + W2$	\leq	0.75
3.	$-W3$	\leq	-0.10
4.	$W3$	\leq	0.55
5.	$-W1$	\leq	-0.20
6.	$W1$	\leq	0.50
7.	$-W2 - W3$	\leq	-0.50
8.	$W2 + W3$	\leq	0.50

Bringing these equations into matrix notation gives

$$A = \begin{bmatrix} -1 & -1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & 1 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} -0.30 \\ 0.75 \\ \end{bmatrix}$$


```
-0.10;
 0.55;
-0.20;
 0.50;
-0.50;
 0.50]
```

Build the constraint matrix `ConSet` by concatenating the matrix `A` to the vector `b`.

```
ConSet = [A, b]
```

Specifying Additional Constraints

The example above defined a constraints matrix that specified a set of typical scenarios. It defined groups of assets, specified upper and lower bounds for total allocation in each of these groups, and it set the total allocation of one of the groups to a fixed value. Constraints like these are common occurrences. The function `portcons` was created to simplify the creation of the constraint matrix for these and other common portfolio requirements. `portcons` takes as input arguments a list of constraint-specifier strings, followed by the data necessary to build the constraint specified by the strings.

Assume that you need to add more constraints to the previous example. Specifically, add a constraint indicating that the sum of weights in any portfolio should be equal to 1, and another set of constraints (one per asset) indicating that the weight for each asset must be greater than 0. This translates into five more constraint rows: two for the new equality, and three indicating that each weight must be greater or equal to 0. The total number of inequalities in the example is now 13. Clearly, creating the constraint matrix can turn into a tedious task.

To create the new constraint matrix using `portcons`, use two separate constraint-specifier strings:

- `'Default'`, which indicates that each weight is greater than 0 and that the total sum of the weights adds to 1
- `'GroupLims'`, which defines the minimum and maximum allocation on each group

The only data requirement for the constraint-specifier string `'Default'` is `NumAssets` (the total number of assets). The constraint-specifier string `'GroupLims'` requires three different arguments: a `Groups` matrix indicating the assets that belong to each group, the `GroupMin` vector indicating the minimum bounds for each group, and the `GroupMax` vector indicating the maximum bounds for each group. Based on the table

Group Membership, build the Group matrix, with each row representing a group, and each column representing an asset.

```
Group = [1  1  0;  
         0  0  1;  
         1  0  0;  
         0  1  1]
```

The table Maximum and Minimum Group Exposure has the information to build GroupMin and GroupMax.

```
GroupMin = [0.30  0.10  0.20  0.50];  
GroupMax = [0.75  0.55  0.50  0.50];
```

Given that the number of assets is three, build the constraint matrix by calling portcons.

```
ConSet = portcons('Default', 3, 'GroupLims', Group, GroupMin,...  
GroupMax);
```

In most cases, portcons('Default') returns the minimal set of constraints required for calling portopt. If ConSet is not specified in the call to portopt, the function calls portcons passing 'Default' as its only specifier.

Now use portopt to obtain the vectors and arrays representing the risk, return, and weights for the portfolios computed along the efficient frontier.

```
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn,...  
ExpCovariance, [], [], ConSet)
```

```
PortRisk = 0.0586  
Port Return = 0.1375  
PortWts = 0.5 0.25 0.25
```

In this case, the constraints allow only one optimum portfolio.

Active Returns and Tracking Error Efficient Frontier

Suppose you want to identify an efficient set of portfolios that minimize the variance of the difference in returns with respect to a given target portfolio, subject to a given expected excess return. The mean and standard deviation of this excess return are often called the active return and active risk, respectively. Active risk is sometimes referred to as the tracking error. Since the objective is to track a given target portfolio as closely as possible, the resulting set of portfolios is sometimes referred to as the tracking error efficient frontier.

Specifically, assume that the target portfolio is expressed as an index weight vector, such that the index return series may be expressed as a linear combination of the available assets. This example illustrates how to construct a frontier that minimizes the active risk (tracking error) subject to attaining a given level of return. That is, it computes the tracking error efficient frontier.

One way to construct the tracking error efficient frontier is to explicitly form the target return series and subtract it from the return series of the individual assets. In this manner, you specify the expected mean and covariance of the active returns, and compute the efficient frontier subject to the usual portfolio constraints.

This example works directly with the mean and covariance of the absolute (unadjusted) returns but converts the constraints from the usual absolute weight format to active weight format.

Consider a portfolio of five assets with the following expected returns, standard deviations, and correlation matrix based on absolute weekly asset returns.

```

NumAssets      = 5;

ExpReturn      = [0.2074  0.1971  0.2669  0.1323  0.2535]/100;

Sigmas        = [2.6570  3.6297  3.9916  2.7145  2.6133]/100;

Correlations   = [1.0000  0.6092  0.6321  0.5833  0.7304
                  0.6092  1.0000  0.8504  0.8038  0.7176
                  0.6321  0.8504  1.0000  0.7723  0.7236
                  0.5833  0.8038  0.7723  1.0000  0.7225
                  0.7304  0.7176  0.7236  0.7225  1.0000];

```

Convert the correlations and standard deviations to a covariance matrix using `corr2cov`.

```
ExpCovariance = corr2cov(Sigmas, Correlations);
```

Next, assume that the target index portfolio is an equally-weighted portfolio formed from the five assets. Note that the sum of index weights equals 1, satisfying the standard full investment budget equality constraint.

```
Index = ones(NumAssets, 1)/NumAssets;
```

Generate an asset constraint matrix using `portcons`. The constraint matrix `AbsConSet` is expressed in absolute format (unadjusted for the index), and is formatted as $[A \ b]$, corresponding to constraints of the form $A*w \leq b$. Each row of `AbsConSet` corresponds to a constraint, and each column corresponds to an asset. Allow no short-selling and full investment in each asset (lower and upper bounds of each asset are 0 and 1, respectively). In particular, note that the first two rows correspond to the budget equality constraint; the remaining rows correspond to the upper/lower investment bounds.

```
AbsConSet = portcons('PortValue', 1, NumAssets, ...
'AssetLims', zeros(NumAssets,1), ones(NumAssets,1));
```

Now transform the absolute constraints to active constraints with `abs2active`.

```
ActiveConSet = abs2active(AbsConSet, Index);
```

An examination of the absolute and active constraint matrices reveals that they are differ only in the last column (the columns corresponding to the b in $A*w \leq b$).

```
[AbsConSet(:,end) ActiveConSet(:,end)]
```

```
ans =
```

```

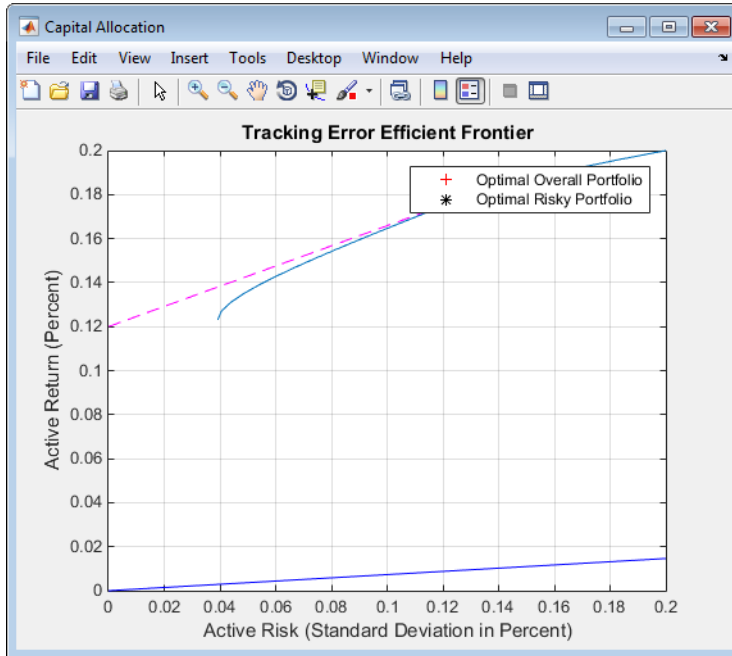
1.0000         0
-1.0000         0
1.0000    0.8000
1.0000    0.8000
1.0000    0.8000
1.0000    0.8000
1.0000    0.8000
1.0000    0.8000
         0    0.2000
         0    0.2000
         0    0.2000
         0    0.2000
         0    0.2000
         0    0.2000
```

In particular, note that the sum-to-one absolute budget constraint becomes a sum-to-zero active budget constraint. The general transformation is as follows:

$$b_{active} = b_{absolute} - A \times Index.$$

Now construct and plot the tracking error efficient frontier with 21 portfolios.

```
[ActiveRisk, ActiveReturn, ActiveWeights] = ...
portopt(ExpReturn,ExpCovariance, 21, [], ActiveConSet);
ActiveRisk = real(ActiveRisk);
plot(ActiveRisk*100, ActiveReturn*100, 'blue')
grid('on')
xlabel('Active Risk (Standard Deviation in Percent)')
ylabel('Active Return (Percent)')
title('Tracking Error Efficient Frontier')
```



Of particular interest is the lower-left portfolio along the frontier. This zero-risk/zero-return portfolio has a practical economic significance. It represents a full investment in the index portfolio itself. Note that each tracking error efficient portfolio (each row in the array `ActiveWeights`) satisfies the active budget constraint, and thus represents portfolio investment allocations with respect to the index portfolio. To convert these allocations to absolute investment allocations, add the index to each efficient portfolio.

```
AbsoluteWeights = ActiveWeights + repmat(Index', 21, 1);
```

Mean-Variance Portfolio Optimization Tools

- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow” on page 4-16
- “Portfolio Object” on page 4-17
- “Constructing the Portfolio Object” on page 4-26
- “Common Operations on the Portfolio Object” on page 4-34
- “Asset Returns and Moments of Asset Returns” on page 4-41
- “Working with Portfolio Constraints” on page 4-57
- “Validate the Portfolio Problem” on page 4-79
- “Estimate Efficient Portfolios” on page 4-84
- “Estimate Efficient Frontiers” on page 4-96
- “Postprocessing Results” on page 4-104
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

Portfolio Optimization Theory

In this section...
“Portfolio Optimization Problems” on page 4-2
“Portfolio Problem Specification” on page 4-2
“Return Proxy” on page 4-3
“Risk Proxy” on page 4-5
“Portfolio Set for Portfolio Optimization” on page 4-7
“Default Portfolio Problem” on page 4-14

Portfolio Optimization Problems

Portfolio optimization problems involve identifying portfolios that satisfy three criteria:

- Minimize a proxy for risk.
- Match or exceed a proxy for return.
- Satisfy basic feasibility requirements.

Portfolios are points from a feasible set of assets that constitute an asset universe. A portfolio specifies either holdings or weights in each individual asset in the asset universe. The convention is to specify portfolios in terms of weights, although the portfolio optimization tools work with holdings as well.

The set of feasible portfolios is necessarily a nonempty, closed, and bounded set. The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices. The proxy for return is a function that characterizes either the gross or net benefits associated with portfolio choices. The terms “risk” and “risk proxy” and “return” and “return proxy” are interchangeable. The fundamental insight of Markowitz (see “Portfolio Optimization” on page A-10) is that the goal of the portfolio choice problem is to seek minimum risk for a given level of return and to seek maximum return for a given level of risk. Portfolios satisfying these criteria are efficient portfolios and the graph of the risks and returns of these portfolios forms a curve called the efficient frontier.

Portfolio Problem Specification

To specify a portfolio optimization problem, you need the following:

- Proxy for portfolio return (μ)
- Proxy for portfolio risk (Σ)
- Set of feasible portfolios (X), called a portfolio set

Financial Toolbox has three objects to solve specific types of portfolio optimization problems:

- The Portfolio object (`Portfolio`) supports mean-variance portfolio optimization (see Markowitz [46], [47] at “Portfolio Optimization” on page A-10). This object has either gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set.
- The PortfolioCVaR object (`PortfolioCVaR`) implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-10), which is generally referred to as CVaR portfolio optimization. CVaR portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses conditional value-at-risk of portfolio returns as the risk proxy.
- The PortfolioMAD object (`PortfolioMAD`) implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] “Portfolio Optimization” on page A-10), which is generally referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

The proxy for portfolio return is a function $\mu : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the rewards associated with portfolio choices. In most cases, the proxy for portfolio return has two general forms, gross and net portfolio returns. Both portfolio return forms separate the risk-free rate r_0 so that the portfolio $x \in X$ contains only risky assets.

Regardless of the underlying distribution of asset returns, a collection of S asset returns y_1, \dots, y_S has a mean of asset returns

$$m = \frac{1}{S} \sum_{s=1}^S y_s,$$

and (sample) covariance of asset returns

$$C = \frac{1}{S-1} \sum_{s=1}^S (y_s - m)(y_s - m)^T.$$

These moments (or alternative estimators that characterize these moments) are used directly in mean-variance portfolio optimization to form proxies for portfolio risk and return.

Gross Portfolio Returns

The gross portfolio return for a portfolio $x \in X$ is

$$\mu(x) = r_0 + (m - r_0 \mathbf{1})^T x,$$

where:

r_0 is the risk-free rate (scalar).

m is the mean of asset returns (n vector).

If the portfolio weights sum to 1, the risk-free rate is irrelevant. The properties in the Portfolio object to specify gross portfolio returns are:

- `RiskFreeRate` for r_0
- `AssetMean` for m

Net Portfolio Returns

The net portfolio return for a portfolio $x \in X$ is

$$\mu(x) = r_0 + (m - r_0 \mathbf{1})^T x - b^T \max\{0, x - x_0\} - s^T \max\{0, x_0 - x\},$$

where:

r_0 is the risk-free rate (scalar).

m is the mean of asset returns (n vector).

b is the proportional cost to purchase assets (n vector).

s is the proportional cost to sell assets (n vector).

You can incorporate fixed transaction costs in this model also. Though in this case, it is necessary to incorporate prices into such costs. The properties in the Portfolio object to specify net portfolio returns are:

- RiskFreeRate for r_0
- AssetMean for m
- InitPort for x_0
- BuyCost for b
- SellCost for s

Risk Proxy

The proxy for portfolio risk is a function $\Sigma : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the risks associated with portfolio choices.

Variance

The variance of portfolio returns for a portfolio $x \in X$ is

$$\Sigma(x) = x^T C x$$

where C is the covariance of asset returns (n -by- n positive-semidefinite matrix).

The property in the Portfolio object to specify the variance of portfolio returns is AssetCovar for C .

Although the risk proxy in mean-variance portfolio optimization is the variance of portfolio returns, the square root, which is the standard deviation of portfolio returns, is often reported and displayed. Moreover, this quantity is often called the “risk” of the portfolio. For details, see Markowitz (“Portfolio Optimization” on page A-10).

Conditional Value-at-Risk

The conditional value-at-risk for a portfolio $x \in X$, which is also known as expected shortfall, is defined as

$$CVaR_\alpha(x) = \frac{1}{1-\alpha} \int_{f(x,y) \geq VaR_\alpha(x)} f(x,y)p(y)dy,$$

where:

α is the probability level such that $0 < \alpha < 1$.

$f(x,y)$ is the loss function for a portfolio x and asset return y .

$p(y)$ is the probability density function for asset return y .

VaR_α is the value-at-risk of portfolio x at probability level α .

The value-at-risk is defined as

$$VaR_\alpha(x) = \min\{\gamma : \Pr[f(x,Y) \leq \gamma] \geq \alpha\}.$$

An alternative formulation for CVaR has the form:

$$CVaR_\alpha(x) = VaR_\alpha(x) + \frac{1}{1-\alpha} \int_{R^n} \max\{0, (f(x,y) - VaR_\alpha(x))\} p(y) dy$$

The choice for the probability level α is typically 0.9 or 0.95. Choosing α implies that the value-at-risk $VaR_\alpha(x)$ for portfolio x is the portfolio return such that the probability of portfolio returns falling below this level is $(1 - \alpha)$. Given $VaR_\alpha(x)$ for a portfolio x , the conditional value-at-risk of the portfolio is the expected loss of portfolio returns above the value-at-risk return.

Note: Value-at-risk is a positive value for losses so that the probability level α indicates the probability that portfolio returns are below the negative of the value-at-risk.

The risk proxy for CVaR portfolio optimization is $CVaR_\alpha(x)$ for a given portfolio $x \in X$ and $\alpha \in (0,1)$. Note that the value-at-risk, or VaR, for a given probability level is estimated whenever CVaR is estimated.

In addition, keep in mind that VaR and CVaR are sample estimators for VaR and CVaR based on the given scenarios. Better scenario samples yield more reliable estimates of VaR and CVaR.

For more information, see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-10.

Mean Absolute-Deviation

The mean-absolute-deviation (MAD) for a portfolio $x \in X$ is defined as

$$\sum(x) = \frac{1}{S} \sum_{s=1}^S |(y_s - m)^T x|$$

where:

y_s are asset returns with scenarios $s = 1, \dots, S$ (S collection of n vectors).

$f(x, y)$ is the loss function for a portfolio x and asset return y .

m is the mean of asset returns (n vector).

such that

$$m = \frac{1}{S} \sum_{s=1}^S y_s$$

For more information, see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10.

Portfolio Set for Portfolio Optimization

The final element for a complete specification of a portfolio optimization problem is the set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset \mathcal{R}^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or “default” portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to 1 (using the budget constraint). The most general portfolio set handled by the portfolio optimization tools can have any of these constraints:

- Linear inequality constraints
- Linear equality constraints
- Bound constraints
- Budget constraints
- Group constraints
- Group ratio constraints
- Average turnover constraints
- One-way turnover constraints

Linear Inequality Constraints

Linear inequality constraints are general linear constraints that model relationships among portfolio weights that satisfy a system of inequalities. Linear inequality constraints take the form

$$A_I x \leq b_I$$

where:

x is the portfolio (n vector).

A_I is the linear inequality constraint matrix (n_I -by- n matrix).

b_I is the linear inequality constraint vector (n_I vector).

n is the number of assets in the universe and n_I is the number of constraints.

Portfolio object properties to specify linear inequality constraints are:

- `AInequality` for A_I
- `bInequality` for b_I
- `NumAssets` for n

The default is to ignore these constraints.

Linear Equality Constraints

Linear equality constraints are general linear constraints that model relationships among portfolio weights that satisfy a system of equalities. Linear equality constraints take the form

$$A_E x = b_E$$

where:

x is the portfolio (n vector).

A_E is the linear equality constraint matrix (n_E -by- n matrix).

b_E is the linear equality constraint vector (n_E vector).

n is the number of assets in the universe and n_E is the number of constraints.

Portfolio object properties to specify linear equality constraints are:

- `AEquality` for A_E
- `bEquality` for b_E
- `NumAssets` for n

The default is to ignore these constraints.

Bound Constraints

Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Since every portfolio set must be bounded, it is often a good practice, albeit not necessary, to set explicit bounds for the portfolio problem. To obtain explicit bounds for a given portfolio set, use the method `estimateBounds`. Bound constraints take the form

$$l_B \leq x \leq u_B$$

where:

x is the portfolio (n vector).

l_B is the lower-bound constraint (n vector).

u_B is the upper-bound constraint (n vector).

n is the number of assets in the universe.

Portfolio object properties to specify bound constraints are:

- `LowerBound` for l_B
- `UpperBound` for u_B
- `NumAssets` for n

The default is to ignore these constraints.

Note that the default portfolio optimization problem (see “Default Portfolio Problem” on page 4-14) has $l_B = 0$ with u_B set implicitly through a budget constraint.

Budget Constraints

Budget constraints are specialized linear constraints that confine the sum of portfolio weights to fall either above or below specific bounds. The constraints take the form

$$l_S \leq \mathbf{1}^T x \leq u_S$$

where:

x is the portfolio (n vector).

$\mathbf{1}$ is the vector of ones (n vector).

l_S is the lower-bound budget constraint (scalar).

u_S is the upper-bound budget constraint (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify budget constraints are:

- `LowerBudget` for l_S
- `UpperBudget` for u_S

- `NumAssets` for n

The default is to ignore this constraint.

The default portfolio optimization problem (see “Default Portfolio Problem” on page 4-14) has $l_S = u_S = 1$, which means that the portfolio weights sum to 1. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0 to 100% invested in cash. If cash is to be a portfolio choice, set `RiskFreeRate` (r_0) to a suitable value (see “Return Proxy” on page 4-3 and “Working with a Riskless Asset” on page 4-51).

Group Constraints

Group constraints are specialized linear constraints that enforce “membership” among groups of assets. The constraints take the form

$$l_G \leq Gx \leq u_G$$

where:

x is the portfolio (n vector).

l_G is the lower-bound group constraint (n_G vector).

u_G is the upper-bound group constraint (n_G vector).

G is the matrix of group membership indexes (n_G -by- n matrix).

Each row of G identifies which assets belong to a group associated with that row. Each row contains either 0s or 1s with 1 indicating that an asset is part of the group or 0 indicating that the asset is not part of the group.

Portfolio object properties to specify group constraints are:

- `GroupMatrix` for G
- `LowerGroup` for l_G
- `UpperGroup` for u_G
- `NumAssets` for n

The default is to ignore these constraints.

Group Ratio Constraints

Group ratio constraints are specialized linear constraints that enforce relationships among groups of assets. The constraints take the form

$$l_{Ri} \cdot (G_B x)_i \leq (G_A x)_i \leq u_{Ri} \cdot (G_B x)_i$$

for $i = 1, \dots, n_R$ where:

x is the portfolio (n vector).

l_R is the vector of lower-bound group ratio constraints (n_R vector).

u_R is the vector matrix of upper-bound group ratio constraints (n_R vector).

G_A is the matrix of base group membership indexes (n_R -by- n matrix).

G_B is the matrix of comparison group membership indexes (n_R -by- n matrix).

n is the number of assets in the universe and n_R is the number of constraints.

Each row of G_A and G_B identifies which assets belong to a base and comparison group associated with that row.

Each row contains either 0s or 1s with 1 indicating that an asset is part of the group or 0 indicating that the asset is not part of the group.

Portfolio object properties to specify group ratio constraints are:

- `GroupA` for G_A
- `GroupB` for G_B
- `LowerRatio` for l_R
- `UpperRatio` for u_R
- `NumAssets` for n

The default is to ignore these constraints.

Average Turnover Constraints

Turnover constraint is a linear absolute value constraint that ensures estimated optimal portfolios differ from an initial portfolio by no more than a specified amount. Although

portfolio turnover is defined in many ways, the turnover constraints implemented in Financial Toolbox computes portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

$$\frac{1}{2} \mathbf{1}^T |x - x_0| \leq \tau$$

where:

x is the portfolio (n vector).

$\mathbf{1}$ is the vector of ones (n vector).

x_0 is the initial portfolio (n vector).

τ is the upper bound for turnover (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

- `Turnover` for τ
- `InitPort` for x_0
- `NumAssets` for n

The default is to ignore this constraint.

One-Way Turnover Constraints

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the forms

$$\mathbf{1}^T \max\{0, x - x_0\} \leq \tau_B$$

$$\mathbf{1}^T \max\{0, x_0 - x\} \leq \tau_S$$

where:

x is the portfolio (n vector)

$\mathbf{1}$ is the vector of ones (n vector).

x_0 is the Initial portfolio (n vector).

τ_B is the upper bound for turnover constraint on purchases (scalar).

τ_S is the upper bound for turnover constraint on sales (scalar).

To specify one-way turnover constraints, use the following properties in the Portfolio or PortfolioCVaR object:

- `BuyTurnover` for τ_B
- `SellTurnover` for τ_S
- `InitPort` for x_0

The default is to ignore this constraint.

Note: The average turnover constraint (see “Average Turnover Constraints” on page 4-12) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

Default Portfolio Problem

The default portfolio optimization problem has a risk and return proxy associated with a given problem, and a portfolio set that specifies portfolio weights to be nonnegative and to sum to 1. The lower bound combined with the budget constraint is sufficient to ensure that the portfolio set is nonempty, closed, and bounded. The default portfolio optimization problem characterizes a long-only investor who is fully invested in a collection of assets.

- For mean-variance portfolio optimization, it is sufficient to set up the default problem. After setting up the problem, data in the form of a mean and covariance of asset returns are then used to solve portfolio optimization problems.
- For conditional value-at-risk portfolio optimization, the default problem requires the additional specification of a probability level that must be set explicitly. Generally, “typical” values for this level are 0.90 or 0.95. After setting up the problem, data in the form of scenarios of asset returns are then used to solve portfolio optimization problems.

- For MAD portfolio optimization, it is sufficient to set up the default problem. After setting up the problem, data in the form of scenarios of asset returns are then used to solve portfolio optimization problems.

See Also

AbstractPortfolio | Portfolio

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Object Workflow”

Portfolio Object Workflow

The Portfolio object workflow for creating and modeling a mean-variance portfolio is:

1 Create a Portfolio.

Create a `Portfolio` object for mean-variance portfolio optimization. For more information, see “Constructing the Portfolio Object”.

2 Estimate the mean and covariance for returns.

Evaluate the mean and covariance for portfolio asset returns, including assets with missing data and financial time series data. For more information, see “Asset Returns and Moments of Asset Returns”.

3 Specify the Portfolio Constraints.

Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, and turnover constraints. For more information, see “Working with Portfolio Constraints”.

4 Validate the Portfolio.

Identify errors for the portfolio specification. For more information, see “Validate the Portfolio Problem”.

5 Estimate the efficient portfolios and frontiers.

Analyze the efficient portfolios and efficient frontiers for a portfolio. For more information, see “Estimate Efficient Portfolios” and “Estimate Efficient Frontiers”.

6 Postprocess the results.

Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see “Postprocessing Results”.

For an example of this workflow, see “Asset Allocation” and “Portfolio Optimization Examples”.

Portfolio Object

In this section...

- “Portfolio Object Properties and Methods” on page 4-17
- “Working with Portfolio Objects” on page 4-21
- “Setting and Getting Properties” on page 4-22
- “Displaying Portfolio Objects” on page 4-22
- “Saving and Loading Portfolio Objects” on page 4-22
- “Estimating Efficient Portfolios and Frontiers” on page 4-23
- “Arrays of Portfolio Objects” on page 4-23
- “Subclassing Portfolio Objects” on page 4-24
- “Conventions for Representation of Data” on page 4-24

Portfolio Object Properties and Methods

The Portfolio object implements mean-variance portfolio optimization and is derived from the abstract class `AbstractPortfolio`. Every property and method of the Portfolio object is public, although some properties and methods are hidden. See Portfolio Object Properties and Portfolio Object Methods for the properties and methods of the Portfolio object. The Portfolio object is a value object where every instance of the object is a distinct version of the object. Since the Portfolio object is also a MATLAB object, it inherits the default methods associated with MATLAB objects.

Portfolio Object Properties

Property	Description	Characteristics
<code>AEquality</code>	Matrix for equality constraints	Matrix
<code>AInequality</code>	Matrix for inequality constraints	Matrix
<code>AssetCovar</code>	Covariance of asset returns	Symmetric positive-semidefinite matrix
<code>AssetList</code>	List of asset names or identifiers	Vector cell array of strings
<code>AssetMean</code>	Mean of asset returns	Vector
<code>bEquality</code>	Vector for equality constraints	Vector
<code>bInequality</code>	Vector for inequality constraints	Vector

Property	Description	Characteristics
BuyCost	Cost to purchase assets	Vector
BuyTurnover	Turnover constraint on purchases	Scalar
GroupA	Base group ratio constraint matrix	Boolean matrix or matrix
GroupB	Comparison group ratio constraint matrix	Boolean matrix or matrix
GroupMatrix	Group membership matrix	Boolean matrix or matrix
InitPort	Initial or current portfolio	Vector
LowerBound	Lower-bound constraint	Vector
LowerBudget	Lower-budget constraint	Scalar
LowerGroup	Lower-group constraint	Vector
LowerRatio	Lower-group ratio constraint ratio	Vector
Name	Name for instance of Portfolio object	String
NumAssets	Number of assets in universe	Scalar positive integer
RiskFreeRate	Period return of riskless asset	Scalar
SellCost	Cost to sell assets	Vector
SellTurnover	Turnover constraint on sales	Scalar
Turnover	Upper-bound portfolio turnover	Scalar
UpperBound	Upper-bound constraint	Vector
UpperBudget	Upper-budget constraint	Scalar
UpperGroup	Upper-group constraint	Vector
UpperRatio	Upper-group ratio constraint ratio	Vector

Portfolio Object Methods

Method	Description
addEquality	Add equality constraints for portfolio weights to existing constraints.
addGroupRatio	Add group ratio constraints for portfolio weights to existing constraints.

Method	Description
<code>addGroups</code>	Add group constraints for portfolio weights to existing constraints.
<code>addInequality</code>	Add inequality constraints for portfolio weights to existing constraints.
<code>checkFeasibility</code>	Determine if portfolios are members of the set of feasible portfolios.
<code>estimateAssetMoments</code>	Estimate mean and covariance of asset returns from price or return data.
<code>estimateBounds</code>	Determine if the set of feasible portfolios is nonempty and bounded.
<code>estimateFrontier</code>	Estimate portfolios on the entire efficient frontier.
<code>estimateFrontierByReturn</code>	Estimate portfolios on the efficient frontier with targeted returns or return proxies.
<code>estimateFrontierByRisk</code>	Estimate portfolios on the efficient frontier with targeted risks or risk proxies.
<code>estimateFrontierLimits</code>	Estimate portfolios at the extreme ends of the efficient frontier (minimum risk and maximum return).
<code>estimateMaxSharpeRatio</code>	Estimate efficient portfolio that maximizes the Sharpe ratio.
<code>estimatePortMoments</code>	Estimate mean and standard deviation of portfolio returns for specified portfolios.
<code>estimatePortReturn</code>	Estimate the return or return proxy for specified portfolios.
<code>estimatePortRisk</code>	Estimate the risk or risk proxy for specified portfolios.
<code>getAssetMoments</code>	Get mean and covariance of asset returns from object.
<code>getBounds</code>	Get lower and upper bounds from object.
<code>getBudget</code>	Get lower and upper budget constraints from object.

Method	Description
getCosts	Get purchase and sales proportional transaction costs from object.
getEquality	Get equality constraint matrix and vector from object.
getGroupRatio	Get base matrix, comparison matrix, and lower and upper bounds for group ratio constraints from object.
getGroups	Get group matrix and lower and upper bounds for group constraints from object.
getInequality	Get inequality constraint matrix and vector from object.
getOneWayTurnover	Get one-way portfolio turnover constraints.
plotFrontier	Plot efficient frontier and optionally obtain risks and returns for portfolios on the efficient frontier.
setAssetList	Set up a list of asset names and symbols to be associated with assets in universe.
setAssetMoments	Set up mean and covariance of asset returns.
setBounds	Set up lower and upper bounds for portfolio weights.
setBudget	Set up lower and upper budget constraints for portfolio weights.
setCosts	Set up purchase and sale proportional transaction costs for assets in universe.
setDefaultConstraints	Set up default constraints for portfolio weights (nonnegative weights that must sum to 1).
setEquality	Set up equality constraints for portfolio weights.
setGroupRatio	Set up group ratio constraints for portfolio weights.
setGroups	Set up group constraints for portfolio weights.

Method	Description
<code>setInequality</code>	Set up inequality constraints for portfolio weights.
<code>setInitPort</code>	Set up initial portfolio weights.
<code>setOneWayTurnover</code>	Set up one-way portfolio turnover constraints.
<code>setOptions</code>	Set up hidden control properties in object (not implemented).
<code>setSolver</code>	Set up solver to estimate efficient portfolios.
<code>setTurnover</code>	Set up average turnover constraints for portfolio weights.

Working with Portfolio Objects

The Portfolio object and its methods are an interface for mean-variance portfolio optimization. Consequently, almost everything you do with the Portfolio object can be done using the methods. The basic workflow is:

- 1 Design your portfolio problem.
- 2 Use the portfolio constructor (`Portfolio`) to create the Portfolio object or use the various set methods to set up your portfolio problem.
- 3 Use estimate methods to solve your portfolio problem.

In addition, methods are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a Portfolio object, you can save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of mean-variance portfolio optimization, means that you have either data or moments for asset returns and a collection of constraints on your portfolios, use the portfolio constructor to set the properties for the Portfolio object. The portfolio constructor lets you create an object from scratch or update an existing object. Since the Portfolio object is a value object, it is easy to create a basic object, then use methods to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see “Constructing the Portfolio Object” on page 4-26.

Setting and Getting Properties

You can set properties of a Portfolio object using either the constructor (`Portfolio`) or various set methods.

Note: Although you can also set properties directly, it is not recommended since error-checking is not performed when you set a property directly.

The portfolio constructor supports setting properties with name-value pair arguments such that each argument name is a property and each value is the value to assign to that property. For example, to set the `AssetMean` and `AssetCovar` properties in an existing Portfolio object `p` with the values `m` and `C`, use the syntax:

```
p = Portfolio(p, 'AssetMean', m, 'AssetCovar', C);
```

In addition to the portfolio constructor, which lets you set individual properties one at a time, groups of properties are set in a Portfolio object with various “set” and “add” methods. For example, to set up an average turnover constraint, use the `setTurnover` method to specify the bound on portfolio average turnover and the initial portfolio. To get individual properties from a Portfolio object, obtain properties directly or use an assortment of “get” methods that obtain groups of properties from a Portfolio object. The Portfolio object constructor and set methods have several useful features:

- The constructor and set methods try to determine the dimensions of your problem with either explicit or implicit inputs.
- The constructor and set methods try to resolve ambiguities with default choices.
- The constructor and set methods perform scalar expansion on arrays when possible.
- The methods try to diagnose and warn about problems.

Displaying Portfolio Objects

The Portfolio object uses the default display method provided by MATLAB, where `display` and `disp` display a Portfolio object and its properties with or without the object variable name.

Saving and Loading Portfolio Objects

Save and load Portfolio objects using the MATLAB `save` and `load` commands.

Estimating Efficient Portfolios and Frontiers

Estimating efficient portfolios and efficient frontiers is the primary purpose of the portfolio optimization tools. A collection of “estimate” and “plot” methods provide ways to explore the efficient frontier. The “estimate” methods obtain either efficient portfolios or risk and return proxies to form efficient frontiers. At the portfolio level, a collection of methods estimates efficient portfolios on the efficient frontier with methods to obtain efficient portfolios:

- At the endpoints of the efficient frontier
- That attain targeted values for return proxies
- That attain targeted values for risk proxies
- Along the entire efficient frontier

These methods also provide purchases and sales needed to shift from an initial or current portfolio to each efficient portfolio. At the efficient frontier level, a collection of methods plot the efficient frontier and estimate either risk or return proxies for efficient portfolios on the efficient frontier. You can use the resultant efficient portfolios or risk and return proxies in subsequent analyses.

Arrays of Portfolio Objects

Although all methods associated with a Portfolio object are designed to work on a scalar Portfolio object, the array capabilities of MATLAB enables you to set up and work with arrays of Portfolio objects. The easiest way to do this is with the `repmat` function. For example, to create a 3-by-2 array of Portfolio objects:

```
p = repmat(Portfolio, 3, 2);
disp(p)
```

After setting up an array of Portfolio objects, you can work on individual Portfolio objects in the array by indexing. For example:

```
p(i,j) = Portfolio(p(i,j), ... );
```

This example calls the Portfolio object constructor for the (i,j) element of a matrix of Portfolio objects in the variable p.

If you set up an array of Portfolio objects, you can access properties of a particular Portfolio object in the array by indexing so that you can set the lower and upper bounds lb and ub for the (i,j,k) element of a 3-D array of Portfolio objects with

```
p(i,j,k) = setBounds(p(i,j,k),lb, ub);
```

and, once set, you can access these bounds with

```
[lb, ub] = getBounds(p(i,j,k));
```

Portfolio object methods work on only one Portfolio object at a time.

Subclassing Portfolio Objects

You can subclass the Portfolio object to override existing methods or to add new properties or methods. To do so, create a derived class from the `Portfolio` class. This gives you all the properties and methods of the `Portfolio` class along with any new features that you choose to add to your subclassed object. The `Portfolio` class is derived from an abstract class called `AbstractPortfolio`. Because of this, you can also create a derived class from `AbstractPortfolio` that implements an entirely different form of portfolio optimization using properties and methods of the `AbstractPortfolio` class.

Conventions for Representation of Data

The portfolio optimization tools follow these conventions regarding the representation of different quantities associated with portfolio optimization:

- Asset returns or prices are in matrix form with samples for a given asset going down the rows and assets going across the columns. In the case of prices, the earliest dates must be at the top of the matrix, with increasing dates going down.
- The mean and covariance of asset returns are stored in a vector and a matrix and the tools have no requirement that the mean must be either a column or row vector.
- Portfolios are in vector or matrix form with weights for a given portfolio going down the rows and distinct portfolios going across the columns.
- Constraints on portfolios are formed in such a way that a portfolio is a column vector.
- Portfolio risks and returns are either scalars or column vectors (for multiple portfolio risks and returns).

See Also

`AbstractPortfolio` | `Portfolio`

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57

- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Constructing the Portfolio Object

In this section...
“Syntax” on page 4-26
“Portfolio Problem Sufficiency” on page 4-27
“Constructor Examples” on page 4-27

To create a fully specified mean-variance portfolio optimization problem, instantiate the Portfolio object using the portfolio constructor. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Syntax

Use the portfolio constructor `Portfolio` to create an instance of an object of the `Portfolio` class. You can use this constructor in several ways. To set up a portfolio optimization problem in a Portfolio object, the simplest syntax is:

```
p = Portfolio;
```

This syntax creates a Portfolio object, `p`, such that all object properties are empty.

The constructor also accepts collections of argument name-value pair arguments for properties and their values. The constructor accepts inputs for public properties (see Portfolio Object Properties) with the general syntax:

```
p = Portfolio('property1', value1, 'property2', value2, ... );
```

If a Portfolio object already exists, the syntax permits the first (and only the first argument) of the portfolio constructor to be an existing object with subsequent argument name-value pair arguments for properties to be added or modified. For example, given an existing Portfolio object in `p`, the general syntax is:

```
p = Portfolio(p, 'property1', value1, 'property2', value2, ... );
```

Input argument names are not case sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 4-31). The `Portfolio` constructor tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process

to formulate a problem. In addition, a Portfolio object is a value object so that, given portfolio `p`, the following code creates two objects, `p` and `q`, that are distinct:

```
q = Portfolio(p, ...)
```

Portfolio Problem Sufficiency

A mean-variance portfolio optimization is completely specified with the Portfolio object if these two conditions are met:

- The moments of asset returns must be specified such that the property `AssetMean` contains a valid finite mean vector of asset returns and the property `AssetCovar` contains a valid symmetric positive-semidefinite matrix for the covariance of asset returns.

The first condition is satisfied by setting the properties associated with the moments of asset returns.

- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded.

The second condition is satisfied by an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed, and several methods, such as `estimateBounds`, provide ways to ensure that your problem is properly formulated.

Although the general sufficiency conditions for mean-variance portfolio optimization go beyond these two conditions, the Portfolio object implemented in Financial Toolbox implicitly handles all these additional conditions. For more information on the Markowitz model for mean-variance portfolio optimization, see “Portfolio Optimization” on page A-10.

Constructor Examples

If you create a Portfolio object, `p`, with no input arguments, you can display it using `disp`:

```
p = Portfolio;  
disp(p); Portfolio
```

Portfolio with properties:

```
    BuyCost: []
    SellCost: []
RiskFreeRate: []
    AssetMean: []
    AssetCovar: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    Name: []
    NumAssets: []
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: []
    UpperBound: []
    LowerBudget: []
    UpperBudget: []
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
    GroupA: []
    GroupB: []
    LowerRatio: []
    UpperRatio: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the portfolio constructor. The custom set methods offer additional ways to set and modify collections of properties in the Portfolio object.

Using the Constructor for a Single-Step Setup

You can use the constructor to directly set up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C`:

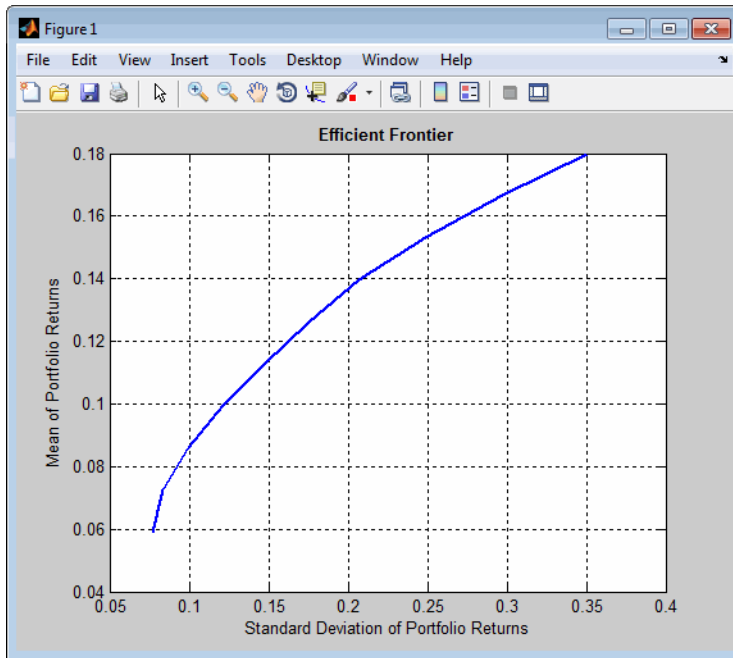
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio('assetmean', m, 'assetcovar', C, ...
            'lowerbudget', 1, 'upperbudget', 1, 'lowerbound', 0);
```

Note that the `LowerBound` property value undergoes scalar expansion since `AssetMean` and `AssetCovar` provide the dimensions of the problem.

You can use dot notation with the method `plotFrontier`.

```
plotFrontier(p);
```



Using the Constructor with a Sequence of Steps

An alternative way to accomplish the same task of setting up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C` (which also illustrates that argument names are not case sensitive):

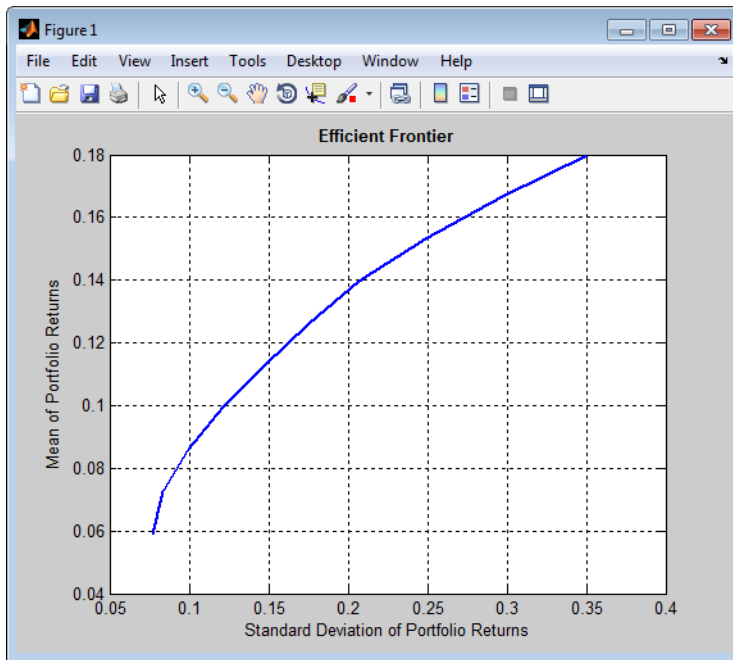
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```

p = Portfolio;
p = Portfolio(p, 'assetmean', m, 'assetcovar', C);
p = Portfolio(p, 'lowerbudget', 1, 'upperbudget', 1);
p = Portfolio(p, 'lowerbound', 0);

plotFrontier(p);

```



This way works because the calls to the constructor are in this particular order. In this case, the call to initialize `AssetMean` and `AssetCovar` provides the dimensions for the problem. If you were to do this step last, you would have to explicitly dimension the `LowerBound` property as follows:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;

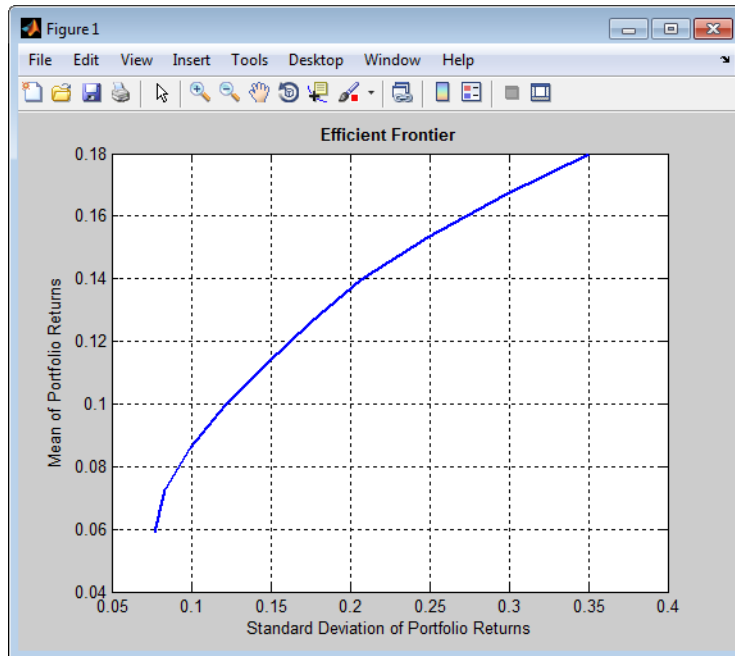
```

```

p = Portfolio(p, 'LowerBound', zeros(size(m)));
p = Portfolio(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = Portfolio(p, 'AssetMean', m, 'AssetCovar', C);

plotFrontier(p);

```



If you did not specify the size of `LowerBound` but, instead, input a scalar argument, the constructor assumes that you are defining a single-asset problem and produces an error at the call to set asset moments with four assets.

Shortcuts for Property Names

The portfolio constructor has shorter argument names that replace longer argument names associated with specific properties of the Portfolio object. For example, rather than enter `'assetcovar'`, the constructor accepts the case-insensitive name `'covar'` to set the `AssetCovar` property in a Portfolio object. Every shorter argument name corresponds with a single property in the Portfolio constructor. The one exception is the alternative argument name `'budget'`, which signifies both the `LowerBudget` and `UpperBudget` properties. When `'budget'` is used, then the `LowerBudget` and `UpperBudget` properties are set to the same value to form an equality budget constraint.

Shortcut Argument Name	Equivalent Argument / Property Name
ae	AEquality
ai	AInequality
covar	AssetCovar
assetnames or assets	AssetList
mean	AssetMean
be	bEquality
bi	bInequality
group	GroupMatrix
lb	LowerBound
n or num	NumAssets
rfr	RiskFreeRate
ub	UpperBound
budget	UpperBudget and LowerBudget

For example, this call to the constructor uses these shortcuts for properties and is equivalent to the previous examples:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio('mean', m, 'covar', C, 'budget', 1, 'lb', 0);
plotFrontier(p);
```

Direct Setting of Portfolio Object Properties

Although not recommended, you can set properties directly, however no error-checking is done on your inputs:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;  
p.NumAssets = numel(m);  
p.AssetMean = m;  
p.AssetCovar = C;  
p.LowerBudget = 1;  
p.UpperBudget = 1;  
p.LowerBound = zeros(size(m));  
  
plotFrontier(p);
```

See Also

Portfolio | estimateBounds

Related Examples

- “Common Operations on the Portfolio Object” on page 4-34
- “Working with Portfolio Constraints” on page 4-57
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Common Operations on the Portfolio Object

In this section...

“Naming a Portfolio Object” on page 4-34

“Configuring the Assets in the Asset Universe” on page 4-34

“Setting Up a List of Asset Identifiers” on page 4-35

“Truncating and Padding Asset Lists” on page 4-36

“Setting Up an Initial or Current Portfolio” on page 4-37

Naming a Portfolio Object

To name a Portfolio object, use the `Name` property. `Name` is informational and has no effect on any portfolio calculations. If the `Name` property is nonempty, `Name` is the title for the efficient frontier plot generated by `plotFrontier`. For example, if you set up an asset allocation fund, you could name the Portfolio object Asset Allocation Fund:

```
p = Portfolio('Name', 'Asset Allocation Fund');  
disp(p.Name);
```

```
Asset Allocation Fund
```

Configuring the Assets in the Asset Universe

The fundamental quantity in the Portfolio object is the number of assets in the asset universe. This quantity is maintained in the `NumAssets` property. Although you can set this property directly, it is usually derived from other properties such as the mean of asset returns and the initial portfolio. In some instances, the number of assets may need to be set directly. This example shows how to set up a Portfolio object that has four assets:

```
p = Portfolio('NumAssets', 4);  
disp(p.NumAssets);
```

```
4
```

After setting the `NumAssets` property, you cannot modify it (unless no other properties are set that depend on `NumAssets`). The only way to change the number of assets in an existing Portfolio object with a known number of assets is to create a new Portfolio object.

Setting Up a List of Asset Identifiers

When working with portfolios, you must specify a universe of assets. Although you can perform a complete analysis without naming the assets in your universe, it is helpful to have an identifier associated with each asset as you create and work with portfolios. You can create a list of asset identifiers as a cell vector of strings in the property `AssetList`. You can set up the list using the next two methods.

Setting Up Asset Lists Using the Constructor

Suppose you have a Portfolio object, `p`, with assets with symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR'. You can create a list of these asset symbols in the object using the constructor:

```
p = Portfolio('assetlist', { 'AA', 'BA', 'CAT', 'DD', 'ETR' });
disp(p.AssetList);
```

```
'AA'    'BA'    'CAT'    'DD'    'ETR'
```

Notice that the property `AssetList` is maintained as a cell array that contains strings, and that it is necessary to pass a cell array into the constructor to set `AssetList`. In addition, notice that the property `NumAssets` is set to 5 based on the number of symbols used to create the asset list:

```
disp(p.NumAssets);
```

```
5
```

Setting Up Asset Lists Using the `setAssetList` Method

You can also specify a list of assets using the method `setAssetList`. Given the list of asset symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR', you can use `setAssetList` with:

```
p = Portfolio;
p = setAssetList(p, { 'AA', 'BA', 'CAT', 'DD', 'ETR' });
disp(p.AssetList);
```

```
'AA'    'BA'    'CAT'    'DD'    'ETR'
```

`setAssetList` also enables you enter symbols directly as a comma-separated list without creating a cell array of strings. For example, given the list of assets symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR', use `setAssetList`:

```
p = Portfolio;
p = setAssetList(p, 'AA', 'BA', 'CAT', 'DD', 'ETR');
```

```
disp(p.AssetList);  
  
'AA'      'BA'      'CAT'     'DD'      'ETR'
```

`setAssetList` has many additional features to create lists of asset identifiers. If you use `setAssetList` with just a Portfolio object, it creates a default asset list according to the name specified in the hidden public property `defaultforAssetList` (which is `'Asset'` by default). The number of asset names created depends on the number of assets in the property `NumAssets`. If `NumAssets` is not set, then `NumAssets` is assumed to be 1.

For example, if a Portfolio object `p` is created with `NumAssets = 5`, then this code fragment shows the default naming behavior:

```
p = Portfolio('numassets',5);  
p = setAssetList(p);  
disp(p.AssetList);  
  
'Asset1'   'Asset2'   'Asset3'   'Asset4'   'Asset5'
```

Suppose that your assets are, for example, ETFs and you change the hidden property `defaultforAssetList` to `'ETF'`, you can then create a default list for ETFs:

```
p = Portfolio('numassets',5);  
p.defaultforAssetList = 'ETF';  
p = setAssetList(p);  
disp(p.AssetList);  
  
'ETF1'     'ETF2'     'ETF3'     'ETF4'     'ETF5'
```

Truncating and Padding Asset Lists

If the `NumAssets` property is already set and you pass in too many or too few identifiers, the portfolio constructor, and the `setAssetList` method truncate or pad the list with numbered default asset names that use the name specified in the hidden public property `defaultforAssetList`. If the list is truncated or padded, a warning message indicates the discrepancy. For example, assume you have a Portfolio object with five ETFs and you only know the first three CUSIPs `'921937835'`, `'922908769'`, and `'922042775'`. Use this syntax to create an asset list that pads the remaining asset identifiers with numbered `'UnknownCUSIP'` placeholders:

```
p = Portfolio('numassets',5);  
p.defaultforAssetList = 'UnknownCUSIP';  
p = setAssetList(p,'921937835', '922908769', '922042775');  
disp(p.AssetList);
```

Warning: Input list of assets has 2 too few identifiers. Padding with numbered assets.

```
> In Portfolio.setAssetList at 121
'921937835' '922908769' '922042775' 'UnknownCUSIP4' 'UnknownCUSIP5'
```

Alternatively, suppose you have too many identifiers and need only the first four assets. This example illustrates truncation of the asset list using the portfolio constructor:

```
p = Portfolio('numassets',4);
p = Portfolio(p, 'assetlist', { 'AGG', 'EEM', 'MDY', 'SPY', 'VEU' });
disp(p.AssetList);
```

Warning: AssetList has 1 too many identifiers. Using first 4 assets.

```
> In Portfolio.checkarguments at 434
In Portfolio.Portfolio>Portfolio.Portfolio at 171
'AGG' 'EEM' 'MDY' 'SPY'
```

The hidden public property `uppercaseAssetList` is a Boolean flag to specify whether to convert asset names to uppercase letters. The default value for `uppercaseAssetList` is `false`. This example shows how to use the `uppercaseAssetList` flag to force identifiers to be uppercase letters:

```
p = Portfolio;
p.uppercaseAssetList = true;
p = setAssetList(p,{ 'aa', 'ba', 'cat', 'dd', 'etr' });
disp(p.AssetList);
```

```
'AA' 'BA' 'CAT' 'DD' 'ETR'
```

Setting Up an Initial or Current Portfolio

In many applications, creating a new optimal portfolio requires comparing the new portfolio with an initial or current portfolio to form lists of purchases and sales. The Portfolio object property `InitPort` lets you identify an initial or current portfolio. The initial portfolio also plays an essential role if you have either transaction costs or turnover constraints. The initial portfolio need not be feasible within the constraints of the problem. This can happen if the weights in a portfolio have shifted such that some constraints become violated. To check if your initial portfolio is feasible, use the method `checkFeasibility` described in “Validating Portfolios” on page 4-81. Suppose you have an initial portfolio in `x0`, then use the Portfolio object constructor to set up an initial portfolio:

```
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = Portfolio('InitPort', x0);
disp(p.InitPort);
```

```
0.3000
```

```
0.2000
0.2000
0
```

As with all array properties, you can set `InitPort` with scalar expansion. This is helpful to set up an equally weighted initial portfolio of, for example, 10 assets:

```
p = Portfolio('NumAssets', 10, 'InitPort', 1/10);
disp(p.InitPort);
```

```
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
```

To clear an initial portfolio from your `Portfolio` object, use either the constructor or the `setInitPort` method with an empty input for the `InitPort` property. If transaction costs or turnover constraints are set, it is not possible to clear the `InitPort` property in this way. In this case, to clear `InitPort`, first clear the dependent properties and then clear the `InitPort` property.

The `InitPort` property can also be set with `setInitPort` which lets you specify the number of assets if you want to use scalar expansion. For example, given an initial portfolio in `x0`, use `setInitPort` to set the `InitPort` property:

```
p = Portfolio;
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = setInitPort(p, x0);
disp(p.InitPort);
```

```
0.3000
0.2000
0.2000
0
```

To create an equally weighted portfolio of four assets, use `setInitPort`:

```
p = Portfolio;
p = setInitPort(p, 1/4, 4);
```

```
disp(p.InitPort);

0.2500
0.2500
0.2500
0.2500
```

Portfolio object methods that work with either transaction costs or turnover constraints also depend on the `InitPort` property. Consequently, the set methods for transaction costs or turnover constraints permit the assignment of a value for the `InitPort` property as part of their implementation. For details, see “Working with Average Turnover Constraints” on page 4-73, “Working with One-Way Turnover Constraints” on page 4-75, and “Working with Transaction Costs” on page 4-52 for details. If either transaction costs or turnover constraints are used, then the `InitPort` property must have a nonempty value. Absent a specific value assigned through the constructor or various set methods, the Portfolio object sets `InitPort` to 0 and warns if `BuyCost`, `SellCost`, or `Turnover` properties are set. The following example illustrates what happens if an average turnover constraint is specified with an initial portfolio:

```
p = Portfolio('Turnover', 0.3, 'InitPort', [ 0.3; 0.2; 0.2; 0.0 ]);
disp(p.InitPort);

0.3000
0.2000
0.2000
0
```

In contrast, this example shows what happens if an average turnover constraint is specified without an initial portfolio:

```
p = Portfolio('Turnover', 0.3);
disp(p.InitPort);

Warning: InitPort and NumAssets are empty and either transaction costs or turnover constraints specified.
Will set NumAssets = 1 and InitPort = 0.
> In Portfolio.checkarguments at 367
   In Portfolio.Portfolio>Portfolio.Portfolio at 171
   0
```

See Also

Portfolio | checkFeasibility | estimateBounds | setAssetList | setInitPort

Related Examples

- “Working with Portfolio Constraints” on page 4-57

- “Asset Returns and Moments of Asset Returns” on page 4-41
- “Validate the Portfolio Problem” on page 4-79
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Asset Returns and Moments of Asset Returns

In this section...

“Assignment Using the Portfolio Constructor” on page 4-41
 “Assignment Using the setAssetMoments Method” on page 4-43
 “Scalar Expansion of Arguments” on page 4-43
 “Estimating Asset Moments from Prices or Returns” on page 4-45
 “Estimating Asset Moments with Missing Data” on page 4-48
 “Estimating Asset Moments from Time Series Data” on page 4-50
 “Working with a Riskless Asset” on page 4-51
 “Working with Transaction Costs” on page 4-52

Since mean-variance portfolio optimization problems require estimates for the mean and covariance of asset returns, the Portfolio object has several ways to set and get the properties `AssetMean` (for the mean) and `AssetCovar` (for the covariance). In addition, the return for a riskless asset is kept in the property `RiskFreeRate` so that all assets in `AssetMean` and `AssetCovar` are risky assets. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Assignment Using the Portfolio Constructor

Suppose you have a mean and covariance of asset returns in variables `m` and `C`. The properties for the moments of asset returns are set using the constructor:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
p = Portfolio('AssetMean', m, 'AssetCovar', C);
disp(p.NumAssets);
disp(p.AssetMean);
disp(p.AssetCovar);

```

```
0.0042
0.0083
0.0100
0.0150

0.0005    0.0003    0.0002         0
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

Notice that the `Portfolio` object determines the number of assets in `NumAssets` from the moments. The portfolio constructor enables separate initialization of the moments, for example:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = Portfolio(p, 'AssetMean', m);
p = Portfolio(p, 'AssetCovar', C);
[assetmean, assetcovar] = p.getAssetMoments
```

```
assetmean =
```

```
0.0042
0.0083
0.0100
0.0150
```

```
assetcovar =
```

```
0.0005    0.0003    0.0002         0
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

The `getAssetMoments` method lets you get the values for `AssetMean` and `AssetCovar` properties at the same time.

Assignment Using the `setAssetMoments` Method

You can also set asset moment properties using the `setAssetMoments` method. For example, given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = setAssetMoments(p, m, C);
[assetmean, assetcovar] = getAssetMoments(p)
```

```
assetmean =
```

```
0.0042
0.0083
0.0100
0.0150
```

```
assetcovar =
```

```
0.0005    0.0003    0.0002         0
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

Scalar Expansion of Arguments

Both the constructor `Portfolio` and the `setAssetMoments` method perform scalar expansion on arguments for the moments of asset returns. When using the constructor, the number of assets must be already specified in the variable `NumAssets`. If `NumAssets` has not already been set, a scalar argument is interpreted as a scalar with `NumAssets` set to 1. `setAssetMoments` provides an additional optional argument to specify the number of assets so that scalar expansion works with the correct number of assets. In addition, if either a scalar or vector is input for the covariance of asset returns, a diagonal matrix is formed such that a scalar expands along the diagonal and a vector

becomes the diagonal. This example demonstrates scalar expansion for four jointly independent assets with a common mean 0.1 and common variance 0.03:

```
p = Portfolio;  
p = setAssetMoments(p, 0.1, 0.03, 4);  
[assetmean, assetcovar] = getAssetMoments(p)
```

```
assetmean =
```

```
0.1000  
0.1000  
0.1000  
0.1000
```

```
assetcovar =
```

```
0.0300    0    0    0  
0    0.0300    0    0  
0    0    0.0300    0  
0    0    0    0.0300
```

If at least one argument is properly dimensioned, you don't need to include the additional `NumAssets` argument. This example illustrates a constant-diagonal covariance matrix and a mean of asset returns for four assets:

```
p = Portfolio;  
p = setAssetMoments(p, [ 0.05; 0.06; 0.04; 0.03 ], 0.03);  
[assetmean, assetcovar] = getAssetMoments(p)
```

```
assetmean =
```

```
0.0500  
0.0600  
0.0400  
0.0300
```

```
assetcovar =
```

```
0.0300    0    0    0  
0    0.0300    0    0  
0    0    0.0300    0  
0    0    0    0.0300
```

In addition, scalar expansion works with the portfolio constructor if `NumAssets` is known, or is deduced from the inputs.

Estimating Asset Moments from Prices or Returns

Another way to set the moments of asset returns is to use the method `estimateAssetMoments` which accepts either prices or returns and estimates the mean and covariance of asset returns. Either prices or returns are stored as matrices with samples going down the rows and assets going across the columns. In addition, prices or returns can be stored in a financial time series (`fints`) object (see “Estimating Asset Moments from Time Series Data” on page 4-50). To illustrate using `estimateAssetMoments`, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with `portsim`. The default behavior of `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by `portsim` in the variable `X`, a price series is created in the variable `Y`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
X = portsim(m', C, 120);
Y = ret2tick(X);
```

Note: Portfolio optimization requires that you use total returns and not just price returns. Consequently, "returns" should be total returns and "prices" should be total return prices.

Given asset returns and prices in variables `X` and `Y` from above, this sequence of examples demonstrates equivalent ways to estimate asset moments for the Portfolio object. A Portfolio object is created in `p` with the moments of asset returns set directly in the constructor, and a second Portfolio object is created in `q` to obtain the mean and covariance of asset returns from asset return data in `X` using `estimateAssetMoments`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
```

```

C = C/12;

X = portsim(m', C, 120);
p = Portfolio('mean', m, 'covar', C);
q = Portfolio;
q = estimateAssetMoments(q, X);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0     0.0010    0.0028    0.0102

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =

    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
    0.0000    0.0010    0.0028    0.0102

```

Notice how either approach has the same moments. The default behavior of `estimateAssetMoments` is to work with asset returns. If, instead, you have asset prices in the variable `Y`, `estimateAssetMoments` accepts an argument name `'DataFormat'` with a corresponding value set to `'prices'` to indicate that the input to the method is in the form of asset prices and not returns (the default value for the `'DataFormat'` argument is `'returns'`). This example compares direct assignment of moments in the

Portfolio object `p` with estimated moments from asset price data in `Y` in the Portfolio object `q`:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
Y = ret2tick(X);

p = Portfolio('mean',m,'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Y, 'dataformat', 'prices');

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =

```

```

0.0005    0.0003    0.0002    0.0000
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0048    0.0028
0.0000    0.0010    0.0028    0.0102

```

Estimating Asset Moments with Missing Data

Often when working with multiple assets, you have missing data indicated by NaN values in your return or price data. Although “Multivariate Normal Regression” on page 9-2 goes into detail about regression with missing data, the method `estimateAssetMoments` has an argument name `'MissingData'` that indicates with a Boolean value whether to use the missing data capabilities of Financial Toolbox software. The default value for `'MissingData'` is `false` which removes all samples with NaN values. If, however, `'MissingData'` is set to `true`, `estimateAssetMoments` uses the ECM algorithm to estimate asset moments. This example illustrates how this works on price data with missing values:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
Y = ret2tick(X);
Y(1:20,1) = NaN;
Y(1:12,4) = NaN;

p = Portfolio('mean',m,'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Y, 'dataformat', 'prices');

r = Portfolio;
r = estimateAssetMoments(r, Y, 'dataformat', 'prices', 'missingdata', true);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)
[rassetmean, rassetcovar] = getAssetMoments(r)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

```

```
passetcovar =
```

```
    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

```
qassetmean =
```

```
    0.0045
    0.0082
    0.0101
    0.0091
```

```
qassetcovar =
```

```
    0.0006    0.0003    0.0001   -0.0000
    0.0003    0.0023    0.0017    0.0011
    0.0001    0.0017    0.0048    0.0029
   -0.0000    0.0011    0.0029    0.0112
```

```
rassetmean =
```

```
    0.0045
    0.0083
    0.0100
    0.0113
```

```
rassetcovar =
```

```
    0.0008    0.0005    0.0001   -0.0001
    0.0005    0.0032    0.0022    0.0015
    0.0001    0.0022    0.0063    0.0040
   -0.0001    0.0015    0.0040    0.0144
```

The Portfolio object **p** contains raw moments, the object **q** contains estimated moments in which NaN values are discarded, and the object **r** contains raw moments that accommodate missing values. Each time you run this example, you will get different estimates for the moments in **q** and **r**, and these will also differ from the moments in **p**.

Estimating Asset Moments from Time Series Data

The `estimateAssetMoments` method also accepts asset returns or prices stored in financial time series (`fints`) objects. `estimateAssetMoments` implicitly works with matrices of data or data in a `fints` object using the same rules for whether the data are returns or prices.

To illustrate, use `fints` to create a `fints` objects `Xfts` that contains asset returns generated with `portsim` (see “Estimating Asset Moments from Prices or Returns” on page 4-45) and add series labels:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);

d = (datenum('31-jan-2001'):datenum('31-dec-2010'))';
Xfts = fints(d, zeros(numel(d),4), {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'});
Xfts = tomonthly(Xfts);

Xfts.Bonds = X(:,1);
Xfts.LargeCap = X(:,2);
Xfts.SmallCap = X(:,3);
Xfts.Emerging = X(:,4);

p = Portfolio('mean',m, 'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Xfts);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```



```

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =

    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
    0.0000    0.0010    0.0028    0.0102

```

As you can see, the moments match. The argument name-value inputs `'DataFormat'` to handle return or price data and `'MissingData'` to ignore or use samples with missing values also work for `fints` data. In addition, `estimateAssetMoments` also extracts asset names or identifiers from a `fints` object with the argument name `'GetAssetList'` set to `true` (its default value is `false`). If the `'GetAssetList'` value is `true`, the identifiers are used to set the `AssetList` property of the object. Thus, repeating the formation of the Portfolio object `q` from the previous example with the `'GetAssetList'` flag set to `true` extracts the series labels from the `fints` object:

```

q = estimateAssetMoments(q, Xfts, 'getassetlist', true);
disp(q.AssetList)

```

```
'Bonds'      'LargeCap'    'SmallCap'    'Emerging'
```

Note if you set the `'GetAssetList'` flag set to `true` and your input data is in a matrix, `estimateAssetMoments` uses the default labeling scheme from `setAssetList` described in “Setting Up a List of Asset Identifiers” on page 4-35.

Working with a Riskless Asset

You can specify a riskless asset with the mean and covariance of asset returns in the `AssetMean` and `AssetCovar` properties such that the riskless asset has variance of 0 and is completely uncorrelated with all other assets. In this case, the Portfolio object uses a separate `RiskFreeRate` property that stores the rate of return of a riskless asset. Thus, you can separate your universe into a riskless asset and a collection of risky assets. For example, assume that your riskless asset has a return in the scalar variable `r0`, then the property for the `RiskFreeRate` is set using the constructor:

```

r0 = 0.01/12;
m = [ 0.05; 0.1; 0.12; 0.18 ];

```

```
C = [ 0.0064 0.00408 0.00192 0;  
      0.00408 0.0289 0.0204 0.0119;  
      0.00192 0.0204 0.0576 0.0336;  
      0 0.0119 0.0336 0.1225 ];  
  
p = Portfolio('RiskFreeRate', r0, 'AssetMean', m, 'AssetCovar', C);  
disp(p.RiskFreeRate);  
  
8.3333e-004
```

Note: If your problem has a budget constraint such that your portfolio weights must sum to 1, then the riskless asset is irrelevant.

Working with Transaction Costs

The difference between net and gross portfolio returns is transaction costs. The net portfolio return proxy has distinct proportional costs to purchase and to sell assets which are maintained in the Portfolio object properties `BuyCost` and `SellCost`. Transaction costs are in units of total return and, as such, are proportional to the price of an asset so that they enter the model for net portfolio returns in return form. For example, suppose you have a stock currently priced \$40 and your usual transaction costs are 5 cents per share. Then the transaction cost for the stock is $0.05/40 = 0.00125$ (as defined in “Net Portfolio Returns” on page 4-4). Costs are entered as positive values and credits are entered as negative values.

Setting Transaction Costs Using the Constructor

To set up transaction costs, you must specify an initial or current portfolio in the `InitPort` property. If the initial portfolio is not set when you set up the transaction cost properties, `InitPort` is 0. The properties for transaction costs can be set using the constructor `Portfolio`. For example, assume that purchase and sale transaction costs are in the variables `bc` and `sc` and an initial portfolio is in the variable `x0`, then transaction costs are set:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];  
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];  
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];  
p = Portfolio('BuyCost', bc, 'SellCost', sc, 'InitPort', x0);  
disp(p.NumAssets);  
disp(p.BuyCost);  
disp(p.SellCost);  
disp(p.InitPort);
```

```

5
0.0013
0.0013
0.0013
0.0013
0.0013

0.0013
0.0070
0.0013
0.0013
0.0024

0.4000
0.2000
0.2000
0.1000
0.1000

```

Setting Transaction Costs Using the `setCosts` Method

You can also set the properties for transaction costs using `setCosts`. Assume that you have the same costs and initial portfolio as in the previous example. Given a `Portfolio` object `p` with an initial portfolio already set, use `setCosts` to set up transaction costs:

```

bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = Portfolio('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);

```

```

5
0.0013
0.0013
0.0013
0.0013
0.0013

```

```
0.0013
0.0070
0.0013
0.0013
0.0024

0.4000
0.2000
0.2000
0.1000
0.1000
```

You can also set up the initial portfolio's `InitPort` value as an optional argument to `setCosts` so that the following is an equivalent way to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = Portfolio;
p = setCosts(p, bc, sc, x0);
```

```
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5

0.0013
0.0013
0.0013
0.0013
0.0013

0.0013
0.0070
0.0013
0.0013
0.0024

0.4000
0.2000
0.2000
```

```
0.1000
0.1000
```

Setting Transaction Costs with Scalar Expansion

Both the constructor `Portfolio` and `setCosts` method implement scalar expansion on the arguments for transaction costs and the initial portfolio. If the `NumAssets` property is already set in the Portfolio object, scalar arguments for these properties are expanded to have the same value across all dimensions. In addition, `setCosts` lets you specify `NumAssets` as an optional final argument. For example, assume that you have an initial portfolio `x0` and you want to set common transaction costs on all assets in your universe. You can set these costs in any of these equivalent ways:

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio('InitPort', x0, 'BuyCost', 0.002, 'SellCost', 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio('InitPort', x0);
p = setCosts(p, 0.002, 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio;
p = setCosts(p, 0.002, 0.002, x0);
```

To clear costs from your Portfolio object, use either the constructor or `setCosts` with empty inputs for the properties to be cleared. For example, you can clear sales costs from the Portfolio object `p` in the previous example:

```
p = Portfolio(p, 'SellCost', []);
```

See Also

`Portfolio` | `estimateAssetMoments` | `getAssetMoments` | `setAssetMoments` | `setCosts`

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Validate the Portfolio Problem” on page 4-79

- “Estimate Efficient Portfolios” on page 4-84
- “Estimate Efficient Frontiers” on page 4-96
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Working with Portfolio Constraints

In this section...

“Setting Default Constraints for Portfolio Weights” on page 4-57

“Working with Bound Constraints” on page 4-60

“Working with Budget Constraints” on page 4-62

“Working with Group Constraints” on page 4-63

“Working with Group Ratio Constraints” on page 4-66

“Working with Linear Equality Constraints” on page 4-69

“Working with Linear Inequality Constraints” on page 4-71

“Working with Average Turnover Constraints” on page 4-73

“Working with One-Way Turnover Constraints” on page 4-75

The final element for a complete specification of a portfolio optimization problem is the set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset \mathcal{R}^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or “default” portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to 1 (using the budget constraint). For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Setting Default Constraints for Portfolio Weights

The “default” portfolio problem has two constraints on portfolio weights:

- Portfolio weights must be nonnegative.
- Portfolio weights must sum to 1.

Implicitly, these constraints imply that portfolio weights are no greater than 1, although this is a superfluous constraint to impose on the problem.

Setting Default Constraints Using the Constructor

Given a portfolio optimization problem with `NumAssets = 20` assets, use the `Portfolio` constructor to set up a default problem and explicitly set bounds and budget constraints:

```
p = Portfolio('NumAssets', 20, 'LowerBound', 0, 'Budget', 1);  
disp(p);
```

Portfolio with properties:

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []  
AssetMean: []  
AssetCovar: []  
Turnover: []  
BuyTurnover: []  
SellTurnover: []  
Name: []  
NumAssets: 20  
AssetList: []  
InitPort: []  
AInequality: []  
bInequality: []  
AEquality: []  
bEquality: []  
LowerBound: [20x1 double]  
UpperBound: []  
LowerBudget: 1  
UpperBudget: 1  
GroupMatrix: []  
LowerGroup: []  
UpperGroup: []  
GroupA: []  
GroupB: []  
LowerRatio: []  
UpperRatio: []
```

Setting Default Constraints Using the `setDefaultConstraints` Method

An alternative approach is to use the `setDefaultConstraints` method. If the number of assets is already known in a `Portfolio` object, use `setDefaultConstraints` with no arguments to set up the necessary bound and budget constraints. Suppose you have 20 assets to set up the portfolio set for a default problem:


```
p = Portfolio('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);
```

Portfolio with properties:

```
    BuyCost: []
    SellCost: []
RiskFreeRate: []
    AssetMean: []
    AssetCovar: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
        Name: []
    NumAssets: 20
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: [20x1 double]
    UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
        GroupA: []
        GroupB: []
    LowerRatio: []
    UpperRatio: []
```

If the number of assets is unknown, `setDefaultConstraints` accepts `NumAssets` as an optional argument to form a portfolio set for a default problem. Suppose you have 20 assets:

```
p = Portfolio;
p = setDefaultConstraints(p, 20);
disp(p);
```

Portfolio with properties:

```
    BuyCost: []
```

```
SellCost: []
RiskFreeRate: []
AssetMean: []
AssetCovar: []
Turnover: []
BuyTurnover: []
SellTurnover: []
Name: []
NumAssets: 20
AssetList: []
InitPort: []
AInequality: []
bInequality: []
AEquality: []
bEquality: []
LowerBound: [20x1 double]
UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
LowerGroup: []
UpperGroup: []
GroupA: []
GroupB: []
LowerRatio: []
UpperRatio: []
```

Working with Bound Constraints

Bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “Bound Constraints” on page 4-9). Although every portfolio set must be bounded, it is not necessary to specify a portfolio set with explicit bound constraints. For example, you can create a portfolio set with an implicit upper bound constraint or a portfolio set with just average turnover constraints. The bound constraints have properties `LowerBound` for the lower-bound constraint and `UpperBound` for the upper-bound constraint. Set default values for these constraints using the `setDefaultConstraints` method (see “Setting Default Constraints for Portfolio Weights” on page 4-57).

Setting Bounds Using the Constructor

The properties for bound constraints are set through the constructor `Portfolio`. Suppose you have a balanced fund with stocks that can range from 50 to 75% of

your portfolio and bonds that can range from 25 to 50% of your portfolio. The bound constraints for a balanced fund are set with:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = Portfolio('LowerBound', lb, 'UpperBound', ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
2
```

```
0.5000
0.2500
```

```
0.7500
0.5000
```

To continue with this example, you must set up a budget constraint. For details, see “Working with Budget Constraints” on page 4-62.

Setting Bounds Using the `setBounds` Method

You can also set the properties for bound constraints using `setBounds`. Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. Given a `Portfolio` object `p`, use `setBounds` to set the bound constraints:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = Portfolio;
p = setBounds(p, lb, ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
2
```

```
0.5000
0.2500
```

```
0.7500
0.5000
```

Setting Bounds Using the Constructor or `setBounds` Method

Both the constructor `Portfolio` and `setBounds` method implement scalar expansion on either the `LowerBound` or `UpperBound` properties. If the `NumAssets` property is already set in the `Portfolio` object, scalar arguments for either property expand to have the same value across all dimensions. In addition, `setBounds` lets you specify `NumAssets` as an optional argument. Suppose you have a universe of 500 assets and you want to set common bound constraints on all assets in your universe. Specifically, you are a long-only investor and want to hold no more than 5% of your portfolio in any single asset. You can set these bound constraints in any of these equivalent ways:

```
p = Portfolio('NumAssets', 500, 'LowerBound', 0, 'UpperBound', 0.05);
```

or

```
p = Portfolio('NumAssets', 500);  
p = setBounds(p, 0, 0.05);
```

or

```
p = Portfolio;  
p = setBounds(p, 0, 0.05, 500);
```

To clear bound constraints from your `Portfolio` object, use either the `Portfolio` constructor or `setBounds` with empty inputs for the properties to be cleared. For example, to clear the upper-bound constraint from the `Portfolio` object `p` in the previous example:

```
p = Portfolio(p, 'UpperBound', []);
```

Working with Budget Constraints

The budget constraint is an optional linear constraint that maintains upper and lower bounds on the sum of portfolio weights (see “Budget Constraints” on page 4-10). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to 1 using `setDefaultConstraints` (see “Setting Default Constraints for Portfolio Weights” on page 4-57).

Setting Budget Constraints Using the Constructor

The properties for the budget constraint can also be set using the constructor `Portfolio`. Suppose you have an asset universe with many risky assets and a riskless asset and you want to ensure that your portfolio never holds more than 1% cash, that is, you want to ensure that you are 99 to 100% invested in risky assets. The budget constraint for this portfolio can be set with:

```
p = Portfolio('LowerBudget', 0.99, 'UpperBudget', 1);
disp(p.LowerBudget);
disp(p.UpperBudget);

0.9900

1
```

Setting Budget Constraints Using the `setBudget` Method

You can also set the properties for a budget constraint using `setBudget`. Suppose you have a fund that permits up to 10% leverage which means that your portfolio can be from 100 to 110% invested in risky assets. Given a `Portfolio` object `p`, use `setBudget` to set the budget constraints:

```
p = Portfolio;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);
disp(p.UpperBudget);

1
```

```
1.1000
```

If you were to continue with this example, then set the `RiskFreeRate` property to the borrowing rate to finance possible leveraged positions. For details on the `RiskFreeRate` property, see “Working with a Riskless Asset” on page 4-51. To clear either bound for the budget constraint from your `Portfolio` object, use either the constructor `Portfolio` or `setBudget` with empty inputs for the properties to be cleared. For example, clear the upper-budget constraint from the `Portfolio` object `p` in the previous example with:

```
p = Portfolio(p, 'UpperBudget', []);
```

Working with Group Constraints

Group constraints are optional linear constraints that group assets together and enforce bounds on the group weights (see “Group Constraints” on page 4-11). Although the

constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with 1 or 0) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the Constructor

The properties for group constraints are set through the constructor `Portfolio`. Suppose you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio, then you can set group constraints:

```
G = [ 1 1 1 0 0 ];  
p = Portfolio('GroupMatrix', G, 'UpperGroup', 0.3);  
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.UpperGroup);
```

5

```
1    1    1    0    0
```

0.3000

The group matrix `G` can also be a logical matrix so that the following code achieves the same result.

```
G = [ true true true false false ];  
p = Portfolio('GroupMatrix', G, 'UpperGroup', 0.3);  
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.UpperGroup);
```

5

```
1    1    1    0    0
```

0.3000

Setting Group Constraints Using the `setGroups` and `addGroups` Methods

You can also set the properties for group constraints using `setGroups`. Suppose you have a portfolio of five assets and want to ensure that the first three assets constitute

no more than 30% of your portfolio. Given a Portfolio object `p`, use `setGroups` to set the group constraints:

```
G = [ true true true false false ];
p = Portfolio;
p = setGroups(p, G, [], 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

5

```
1    1    1    0    0
```

0.3000

In this example, you would set the `LowerGroup` property to be empty (`[]`).

Suppose you want to add another group constraint to make odd-numbered assets constitute at least 20% of your portfolio. Set up an augmented group matrix and introduce infinite bounds for unconstrained group bounds or use the `addGroups` method to build up group constraints. For this example, create another group matrix for the second group constraint:

```
p = Portfolio;
G = [ true true true false false ]; % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ]; % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

5

```
1    1    1    0    0
1    0    1    0    1
```

-Inf
0.2000

0.3000
Inf

`addGroups` determines which bounds are unbounded so you only need to focus on the constraints that you want to set.

The constructor `Portfolio` and `setGroups` and `addGroups` implement scalar expansion on either the `LowerGroup` or `UpperGroup` properties based on the dimension of the group matrix in the property `GroupMatrix`. Suppose you have a universe of 30 assets with 6 asset classes such that assets 1-5, assets 6-12, assets 13-18, assets 19-22, assets 23-27, and assets 28-30 constitute each of your asset classes and you want each asset class to fall from 0 to 25% of your portfolio. Let the following group matrix define your groups and scalar expansion define the common bounds on each group:

```
p = Portfolio;
G = blkdiag(true(1,5), true(1,7), true(1,6), true(1,4), true(1,5), true(1,3));
p = setGroups(p, G, 0, 0.25);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

30

Columns 1 through 16

1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Columns 17 through 30

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0

0
0
0
0
0
0

0.2500
0.2500
0.2500
0.2500
0.2500
0.2500

Working with Group Ratio Constraints

Group ratio constraints are optional linear constraints that maintain bounds on proportional relationships among groups of assets (see “Group Ratio Constraints” on page 4-12). Although the constraints are implemented as general constraints, the usual

convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with 1 or 0) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group to a comparison group fall within specified bounds. Group ratio constraints have properties:

- `GroupA` for the base membership matrix
- `GroupB` for the comparison membership matrix
- `LowerRatio` for the lower-bound constraint on the ratio of groups
- `UpperRatio` for the upper-bound constraint on the ratio of groups

Setting Group Ratio Constraints Using the Constructor

The properties for group ratio constraints are set using constructor `Portfolio`. For example, assume you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ]; % financial companies
GB = [ 0 0 0 1 1 1 ]; % nonfinancial companies
p = Portfolio('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

6

```
1    1    1    0    0    0
```

```
0    0    0    1    1    1
```

0.5000

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false true true true true ]; % nonfinancial companies
p = Portfolio('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```

6
1   1   1   0   0   0
0   0   0   1   1   1
0.5000

```

Setting Group Ratio Constraints Using the `setGroupRatio` and `addGroupRatio` Methods

You can also set the properties for group ratio constraints using `setGroupRatio`. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). Given a Portfolio object `p`, use `setGroupRatio` to set the group constraints:

```

GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);

```

```

6
1   1   1   0   0   0
0   0   0   1   1   1
0.5000

```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose you want to add another group ratio constraint to ensure that the weights in odd-numbered assets constitute at least 20% of the weights in nonfinancial assets your portfolio. You can set up augmented group ratio matrices and introduce infinite bounds for unconstrained group ratio bounds, or you can use the `addGroupRatio` method to build up group ratio constraints. For this example, create another group matrix for the second group constraint:

```

p = Portfolio;
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

```

```

GA = [ true false true false true false ]; % odd-numbered companies
GB = [ false false false true true true ]; % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);

6

1      1      1      0      0      0
1      0      1      0      1      0

0      0      0      1      1      1
0      0      0      1      1      1

-Inf
0.2000

0.5000
Inf

```

Notice that `addGroupRatio` determines which bounds are unbounded so you only need to focus on the constraints you want to set.

The constructor `Portfolio`, `setGroupRatio`, and `addGroupRatio` implement scalar expansion on either the `LowerRatio` or `UpperRatio` properties based on the dimension of the group matrices in `GroupA` and `GroupB` properties.

Working with Linear Equality Constraints

Linear equality constraints are optional linear constraints that impose systems of equalities on portfolio weights (see “Linear Equality Constraints” on page 4-9). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `bEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the Constructor

The properties for linear equality constraints are set using the constructor `Portfolio`. Suppose you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. To set this constraint:

```
A = [ 1 1 1 0 0 ];
```

```
b = 0.5;
p = Portfolio('AEquality', A, 'bEquality', b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);
```

5

```
1    1    1    0    0
```

0.5000

Setting Linear Equality Constraints Using the `setEquality` and `addEquality` Methods

You can also set the properties for linear equality constraints using `setEquality`. Suppose you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. Given a `Portfolio` object `p`, use `setEquality` to set the linear equality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio;
p = setEquality(p, A, b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);
```

5

```
1    1    1    0    0
```

0.5000

Suppose you want to add another linear equality constraint to ensure that the last three assets also constitute 50% of your portfolio. You can set up an augmented system of linear equalities or use `addEquality` to build up linear equality constraints. For this example, create another system of equalities:

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % first equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % second equality constraint
```

```

b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000

```

The constructor `Portfolio` and `setEquality` and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

Working with Linear Inequality Constraints

Linear inequality constraints are optional linear constraints that impose systems of inequalities on portfolio weights (see “Linear Inequality Constraints” on page 4-8). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the Constructor

The properties for linear inequality constraints are set using the constructor `Portfolio`. Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. To set up these constraints:

```

A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio('AInequality', A, 'bInequality', b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);

5

1     1     1     0     0

```

0.5000

Setting Linear Inequality Constraints Using the `setInequality` and `addInequality` Methods

You can also set the properties for linear inequality constraints using `setInequality`. Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 50% of your portfolio. Given a `Portfolio` object `p`, use `setInequality` to set the linear inequality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio;
p = setInequality(p, A, b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

5

```
1    1    1    0    0
```

0.5000

Suppose you want to add another linear inequality constraint to ensure that the last three assets constitute at least 50% of your portfolio. You can set up an augmented system of linear inequalities or use the `addInequality` method to build up linear inequality constraints. For this example, create another system of inequalities:

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

5

```
1    1    1    0    0
```

```

0      0      -1      -1      -1

0.5000
-0.5000

```

The constructor `Portfolio` and `setInequality` and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

Working with Average Turnover Constraints

The turnover constraint is an optional linear absolute value constraint (see “Average Turnover Constraints” on page 4-12) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the portfolio constructor `Portfolio` or the `setTurnover` method. The turnover constraint depends on an initial or current portfolio, which is assumed to be zero if not set when the turnover constraint is set. The turnover constraint has properties `Turnover`, for the upper bound on average turnover, and `InitPort`, for the portfolio against which turnover is computed.

Setting Average Turnover Constraints Using the Constructor

The properties for the turnover constraints are set using the constructor `Portfolio`. Suppose you have an initial portfolio of 10 assets in a variable `x0` and you want to ensure that average turnover is no more than 30%. To set this turnover constraint:

```

x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('Turnover', 0.3, 'InitPort', x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);

10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100

```

```
0.0800
0.1000
```

Note if the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 4-37).

Setting Average Turnover Constraints Using the `setTurnover` Method

You can also set properties for portfolio turnover using `setTurnover` to specify both the upper bound for average turnover and an initial portfolio. Suppose you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that average turnover is no more than 30%. Given a `Portfolio` object `p`, use `setTurnover` to set the turnover constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('InitPort', x0);
p = setTurnover(p, 0.3);
```

```
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
```

```
0.3000
```

```
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

```
or
```

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio;
p = setTurnover(p, 0.3, x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
```



```

0.3000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000

```

`setTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the Portfolio object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setTurnover` lets you specify `NumAssets` as an optional argument. To clear turnover from your Portfolio object, use the constructor `Portfolio` or `setTurnover` with empty inputs for the properties to be cleared.

Working with One-Way Turnover Constraints

One-way turnover constraints are optional constraints (see “One-Way Turnover Constraints” on page 4-13) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the portfolio constructor `Portfolio` or the `setOneWayTurnover` method. One-way turnover constraints depend upon an initial or current portfolio, which is assumed to be zero if not set when the turnover constraints are set. One-way turnover constraints have properties `BuyTurnover`, for the upper bound on net purchases, `SellTurnover`, for the upper bound on net sales, and `InitPort`, for the portfolio against which turnover is computed.

Setting One-Way Turnover Constraints Using the Constructor

The Properties for the one-way turnover constraints are set using the constructor `Portfolio`. Suppose you have an initial portfolio with 10 assets in a variable `x0` and you want to ensure that turnover on purchases is no more than 30% and turnover on sales is no more than 20% of the initial portfolio. To set these turnover constraints:

```

x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('BuyTurnover', 0.3, 'SellTurnover', 0.2, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyTurnover);

```

```
disp(p.SellTurnover);
disp(p.InitPort);

10

0.3000

0.2000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

If the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 4-37).

Setting Turnover Constraints Using the `setOneWayTurnover` Method

You can also set properties for portfolio turnover using `setOneWayTurnover` to specify the upper bounds for turnover on purchases (`BuyTurnover`) and sales (`SellTurnover`) and an initial portfolio. Suppose you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that turnover on purchases is no more than 30% and that turnover on sales is no more than 20% of the initial portfolio. Given a `Portfolio` object `p`, use `setOneWayTurnover` to set the turnover constraints with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);

disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);

10

0.3000
```

```

0.2000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000

```

OR

```

x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio;
p = setOneWayTurnover(p, 0.3, 0.2, x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);

```

```

10
0.3000
0.2000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000

```

`setOneWayTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the `Portfolio` object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setOneWayTurnover` lets you specify `NumAssets` as an optional argument. To remove one-way turnover from your `Portfolio` object, use the constructor `Portfolio` or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality`
| `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` |
`setTurnover`

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Validate the Portfolio Problem” on page 4-79
- “Estimate Efficient Portfolios” on page 4-84
- “Estimate Efficient Frontiers” on page 4-96
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Validate the Portfolio Problem

In this section...

“Validating a Portfolio Set” on page 4-79

“Validating Portfolios” on page 4-81

In some cases, you may want to validate either your inputs to, or outputs from, a portfolio optimization problem. Although most error checking that occurs during the problem setup phase catches most difficulties with a portfolio optimization problem, the processes to validate portfolio sets and portfolios are time consuming and are best done offline. Consequently, the portfolio optimization tools have specialized methods to validate portfolio sets and portfolios. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Validating a Portfolio Set

Since it is necessary and sufficient that your portfolio set must be a nonempty, closed, and bounded set to have a valid portfolio optimization problem, the method `estimateBounds` lets you examine your portfolio set to determine if it is nonempty and, if nonempty, whether it is bounded. Suppose you have the following portfolio set which is an empty set because the initial portfolio at 0 is too far from a portfolio that satisfies the budget and turnover constraint:

```
p = Portfolio('NumAssets', 3, 'Budget', 1);
p = setTurnover(p, 0.3, 0);
```

If a portfolio set is empty, `estimateBounds` returns NaN bounds and sets the `isbounded` flag to []:

```
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
NaN
NaN
NaN
```

```
ub =
```

```
NaN
```

```
NaN
NaN

isbounded =

[]
```

Suppose you create an unbounded portfolio set as follows:

```
p = Portfolio('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

-Inf
-Inf

ub =

1.0e-008 *

-0.3712
    Inf

isbounded =

0
```

In this case, `estimateBounds` returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Finally, suppose you created a portfolio set that is both nonempty and bounded. `estimateBounds` not only validates the set, but also obtains tighter bounds which is useful if you are concerned with the actual range of portfolio choices for individual assets in your portfolio set:

```
p = Portfolio;
p = setBudget(p, 1,1);
p = setBounds(p, [-0.1; 0.2; 0.3; 0.2 ], [ 0.5; 0.3; 0.9; 0.8 ]);

[lb, ub, isbounded] = estimateBounds(p)

lb =

-0.1000
0.2000
```

```

0.3000
0.2000

ub =

0.3000
0.3000
0.7000
0.6000

isbounded =

1

```

In this example, all but the second asset have tighter upper bounds than the input upper bound implies.

Validating Portfolios

Given a portfolio set specified in a Portfolio object, you often want to check if specific portfolios are feasible with respect to the portfolio set. This can occur with, for example, initial portfolios and with portfolios obtained from other procedures. The `checkFeasibility` method determines whether a collection of portfolios is feasible. Suppose you perform the following portfolio optimization and want to determine if the resultant efficient portfolios are feasible relative to a modified problem.

First, set up a problem in the Portfolio object `p`, estimate efficient portfolios in `pwgt`, and then confirm that these portfolios are feasible relative to the initial problem:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)

ans =

```

```
1 1 1 1 1 1 1 1 1 1
```

Next, set up a different portfolio problem that starts with the initial problem with an additional a turnover constraint and an equally weighted initial portfolio:

```
q = setTurnover(p, 0.3, 0.25);  
checkFeasibility(q, pwgt)
```

```
ans =
```

```
0 0 0 1 1 0 0 0 0 0
```

In this case, only two of the ten efficient portfolios from the initial problem are feasible relative to the new problem in Portfolio object `q`. Solving the second problem using `checkFeasibility` demonstrates that the efficient portfolio for Portfolio object `q` is feasible relative to the initial problem:

```
qwgt = estimateFrontier(q);  
checkFeasibility(p, qwgt)
```

```
ans =
```

```
1 1 1 1 1 1 1 1 1 1
```

See Also

[Portfolio](#) | [checkFeasibility](#) | [estimateBounds](#)

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Estimate Efficient Portfolios” on page 4-84
- “Estimate Efficient Frontiers” on page 4-96
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2

- “Portfolio Object Workflow”

Estimate Efficient Portfolios

In this section...

“Obtaining Portfolios Along the Entire Efficient Frontier” on page 4-84

“Obtaining Endpoints of the Efficient Frontier” on page 4-85

“Obtaining Efficient Portfolios for Target Returns” on page 4-87

“Obtaining Efficient Portfolios for Target Risks” on page 4-90

“Efficient Portfolio that Maximizes Sharpe Ratio” on page 4-92

“Choosing and Controlling the Solver” on page 4-94

There are two ways to look at a portfolio optimization problem that depends on what you are trying to do. One goal is to estimate efficient portfolios and the other is to estimate efficient frontiers. This section focuses on the former goal and “Estimate Efficient Frontiers” on page 4-96 focuses on the latter goal. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Obtaining Portfolios Along the Entire Efficient Frontier

The most basic way to obtain optimal portfolios is to obtain points over the entire range of the efficient frontier. Given a portfolio optimization problem in a Portfolio object, the `estimateFrontier` method computes efficient portfolios spaced evenly according to the return proxy from the minimum to maximum return efficient portfolios. The number of portfolios estimated is controlled by the hidden property `defaultNumPorts` which is set to 10. A different value for the number of portfolios estimated is specified as input to `estimateFrontier`. This example shows the default number of efficient portfolios over the entire range of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);

disp(pwgt);
```

0.8891	0.7215	0.5540	0.3865	0.2190	0.0515	0	0	0	0
0.0369	0.1289	0.2209	0.3129	0.4049	0.4969	0.4049	0.2314	0.0579	0
0.0404	0.0567	0.0730	0.0893	0.1056	0.1219	0.1320	0.1394	0.1468	0

```
0.0336 0.0929 0.1521 0.2113 0.2705 0.3297 0.4630 0.6292 0.7953 1.0000
```

If you want only four portfolios in the previous example:

```
pwgt = estimateFrontier(p, 4);
disp(pwgt);
```

```
0.8891 0.3865 0 0
0.0369 0.3129 0.4049 0
0.0404 0.0893 0.1320 0
0.0336 0.2113 0.4630 1.0000
```

Starting from the initial portfolio, `estimateFrontier` also returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);
```

```
display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =
```

```
0.8891 0.7215 0.5540 0.3865 0.2190 0.0515 0 0 0 0
0.0369 0.1289 0.2209 0.3129 0.4049 0.4969 0.4049 0.2314 0.0579 0
0.0404 0.0567 0.0730 0.0893 0.1056 0.1219 0.1320 0.1394 0.1468 0
0.0336 0.0929 0.1521 0.2113 0.2705 0.3297 0.4630 0.6292 0.7953 1.0000
```

```
pbuy =
```

```
0.5891 0.4215 0.2540 0.0865 0 0 0 0 0 0
0 0 0 0.0129 0.1049 0.1969 0.1049 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0.0521 0.1113 0.1705 0.2297 0.3630 0.5292 0.6953 0.9000
```

```
psell =
```

```
0 0 0 0 0.0810 0.2485 0.3000 0.3000 0.3000 0.3000
0.2631 0.1711 0.0791 0 0 0 0 0.0686 0.2421 0.3000
0.1596 0.1433 0.1270 0.1107 0.0944 0.0781 0.0680 0.0606 0.0532 0.2000
0.0664 0.0071 0 0 0 0 0 0 0 0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Endpoints of the Efficient Frontier

In many cases, you might be interested in the endpoint portfolios for the efficient frontier. Suppose you want to determine the range of returns from minimum to

maximum to refine a search for a portfolio with a specific target return. Use the `estimateFrontierLimits` method to obtain the endpoint portfolios:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
```

```
disp(pwgt);
```

```
    0.8891         0
    0.0369         0
    0.0404         0
    0.0336    1.0000
```

The `estimatePortMoments` method shows the range of risks and returns for efficient portfolios:

```
[prsk, pret] = estimatePortMoments(p, pwgt);
disp([prsk, pret]);
```

```
    0.0769    0.0590
    0.3500    0.1800
```

Starting from an initial portfolio, `estimateFrontierLimits` also returns purchases and sales to get from the initial portfolio to the endpoint portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
```

```

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierLimits(p);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

    0.8891    0
    0.0369    0
    0.0404    0
    0.0336    1.0000

pbuy =

    0.5891    0
    0         0
    0         0
    0         0.9000

psell =

    0         0.3000
    0.2631    0.3000
    0.1596    0.2000
    0.0664    0

```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Efficient Portfolios for Target Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` method accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. For example, assume that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 6%, 9%, and 12%:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;

```

```

0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);

display(pwgt);

pwgt =

    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022

```

In some cases, you can request a return for which no efficient portfolio exists. Based on the previous example, suppose you want a portfolio with a 5% return (which is the return of the first asset). A portfolio that is fully invested in the first asset, however, is inefficient. `estimateFrontierByReturn` warns if your target returns are outside the range of efficient portfolio returns and replaces it with the endpoint portfolio of the efficient frontier closest to your target return:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByReturn(p, [0.05, 0.09, 0.12]);

display(pwgt);

Warning: One or more target return values are outside the feasible range [ 0.0590468, 0.18 ].
Will return portfolios associated with endpoints of the range for these values.
> In Portfolio.estimateFrontierByReturn at 70

pwgt =

    0.8891    0.5032    0.1293
    0.0369    0.2488    0.4541
    0.0404    0.0780    0.1143
    0.0336    0.1700    0.3022

```

The best way to avoid this situation is to bracket your target portfolio returns with `estimateFrontierLimits` and `estimatePortReturn` (see “Obtaining Endpoints of the Efficient Frontier” on page 4-85 and “Obtaining Portfolio Risks and Returns” on page 4-96).

```

pret = estimatePortReturn(p, p.estimateFrontierLimits);
display(pret);
pret =
    0.0590
    0.1800

```

This result indicates that efficient portfolios have returns that range between 5.9% and 18%.

If you have an initial portfolio, `estimateFrontierByReturn` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, to obtain purchases and sales with target returns of 6%, 9%, and 12%:

```

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);
display(pwgt);
display(pbuy);
display(psell);
pwgt =
    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022
pbuy =
    0.5772    0.2032         0
         0         0    0.1541
         0         0         0
         0    0.0700    0.2022
psell =
         0         0    0.1707
    0.2566    0.0512         0
    0.1584    0.1220    0.0857
    0.0622         0         0

```

If you do not have an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Efficient Portfolios for Target Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` method accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Suppose you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);

pwgt =

    0.3984    0.2659    0.1416
    0.3064    0.3791    0.4474
    0.0882    0.1010    0.1131
    0.2071    0.2540    0.2979
```

In some cases, you can request a risk for which no efficient portfolio exists. Based on the previous example, suppose you want a portfolio with 7% risk (individual assets in this universe have risks ranging from 8 to 35%). It turns out that a portfolio with 7% risk cannot be formed with these four assets. `estimateFrontierByRisk` warns if your target risks are outside the range of efficient portfolio risks and replaces it with the endpoint of the efficient frontier closest to your target risk:

```
pwgt = estimateFrontierByRisk(p, 0.07)

Warning: One or more target risk values are outside the feasible range [ 0.0769288, 0.35 ].
Will return portfolios associated with endpoints of the range for these values.
> In Portfolio.estimateFrontierByRisk at 82

pwgt =

    0.8891
    0.0369
    0.0404
    0.0336
```


The best way to avoid this situation is to bracket your target portfolio risks with `estimateFrontierLimits` and `estimatePortRisk` (see “Obtaining Endpoints of the Efficient Frontier” on page 4-85 and “Obtaining Portfolio Risks and Returns” on page 4-96).

```
prsk = estimatePortRisk(p, p.estimateFrontierLimits);

display(prsk);

prsk =

    0.0769
    0.3500
```

This result indicates that efficient portfolios have risks that range from 7.7 to 35%.

Starting with an initial portfolio, `estimateFrontierByRisk` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the example with target risks of 12%, 14%, and 16%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

    0.3984    0.2659    0.1416
    0.3064    0.3791    0.4474
    0.0882    0.1010    0.1131
    0.2071    0.2540    0.2979

pbuy =

    0.0984         0         0
    0.0064    0.0791    0.1474
         0         0         0
    0.1071    0.1540    0.1979

psell =
```

```

      0      0.0341      0.1584
      0      0      0
0.1118      0.0990      0.0869
      0      0      0
    
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Efficient Portfolio that Maximizes Sharpe Ratio

The Sharpe ratio is defined as the ratio

$$\frac{\mu(x) - r_0}{\sqrt{\Sigma(x)}}$$

where $x \in R^n$ and r_0 is the risk-free rate (μ and Σ proxies for portfolio return and risk). For more information, see “Portfolio Optimization Theory” on page 4-2.

Portfolios that maximize the Sharpe ratio are portfolios on the efficient frontier that satisfy a number of theoretical conditions in finance. For example, such portfolios are called tangency portfolios since the tangent line from the risk-free rate to the efficient frontier touches the efficient frontier at portfolios that maximize the Sharpe ratio.

To obtain efficient portfolios that maximizes the Sharpe ratio, the `estimateMaxSharpeRatio` method accepts a `Portfolio` object and obtains efficient portfolios that maximize the Sharpe Ratio.

Suppose you have a universe with four risky assets and a riskless asset and you want to obtain a portfolio that maximizes the Sharpe ratio, where, in this example, r_0 is the return for the riskless asset.

```

r0 = 0.03;
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio('RiskFreeRate', r0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
    
```

```
pwgt = estimateMaxSharpeRatio(p);
```

```
display(pwgt);
```

```
pwgt =
    0.4251
    0.2917
    0.0856
    0.1977
```

If you start with an initial portfolio, `estimateMaxSharpeRatio` also returns purchases and sales to get from your initial portfolio to the portfolio that maximizes the Sharpe ratio. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the previous example:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
```

```
p = setInitPort(p, pwgt0);
```

```
[pwgt, pbuy, psell] = estimateMaxSharpeRatio(p);
```

```
display(pwgt);
```

```
display(pbuy);
```

```
display(psell);
```

```
pwgt =
    0.4251
    0.2917
    0.0856
    0.1977
```

```
pbuy =
```

```
    0.1251
         0
         0
    0.0977
```

```
psell =
```

```
         0
    0.0083
    0.1144
```

0

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Choosing and Controlling the Solver

The default solver for mean-variance portfolio optimization is `lcprog`, which implements a linear complementarity programming (LCP) algorithm. Although `lcprog` works for most problems, you can adjust arguments to control the algorithm. Alternatively, the mean-variance portfolio optimization tools let you use any of the variations of `quadprog` from Optimization Toolbox™ software. Unlike Optimization Toolbox which uses the `trust-region-reflective` algorithm as the default algorithm for `quadprog`, the portfolio optimization tools use the `interior-point-convex` algorithm. For details about `quadprog` and quadratic programming algorithms and options, see “Quadratic Programming Algorithms”.

To modify either `lcprog` or to specify `quadprog` as your solver, use the `setSolver` method to set the hidden properties `solverType` and `solverOptions` that specify and control the solver. Since the solver properties are hidden, you cannot set these using the portfolio constructor `Portfolio`. The default solver is `lcprog` so you do not need to use `setSolver` to specify this solver. To use `quadprog`, you must set up the `interior-point-convex` version of `quadprog` using:

```
p = Portfolio;  
p = setSolver(p, 'quadprog');  
display(p.solverType);
```

`quadprog`

and you can switch back to `lcprog` with:

```
p = setSolver(p, 'lcprog');  
display(p.solverType);
```

`lcprog`

In both cases, `setSolver` sets up default options associated with either solver. If you want to specify additional options associated with a given solver, `setSolver` accepts these options with argument name-value pair arguments in the function call. For example, if you intend to use `quadprog` and want to use the `active-set` algorithm, call `setSolver` with:

```
p = setSolver(p, 'quadprog', 'Algorithm', 'active-set');  
display(p.solverOptions.Algorithm);
```

```
active-set
```

In addition, if you want to specify any of the options for `quadprog` that are normally set through `optimoptions`, `setSolver` accepts an `optimoptions` object as the second argument. For example, you can start with the default options for `quadprog` set by `setSolver` and then change the algorithm to `'trust-region-reflective'` with no displayed output:

```
p = Portfolio;
options = optimoptions('quadprog', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'quadprog', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Estimate Efficient Frontiers” on page 4-96
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Estimate Efficient Frontiers

In this section...

“Obtaining Portfolio Risks and Returns” on page 4-96

“Plotting the Efficient Frontier” on page 4-98

Whereas “Estimate Efficient Portfolios” on page 4-84 focused on estimation of efficient portfolios, this section focuses on the estimation of efficient frontiers. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Obtaining Portfolio Risks and Returns

Given any portfolio and, in particular, efficient portfolios, the methods `estimatePortReturn`, `estimatePortRisk`, and `estimatePortMoments` provide estimates for the return (or return proxy), risk (or the risk proxy), and, in the case of mean-variance portfolio optimization, the moments of expected portfolio returns. Each method has the same input syntax but with different combinations of outputs. Suppose you have this following portfolio optimization problem that gave you a collection of portfolios along the efficient frontier in `pwgt`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = Portfolio('AssetMean', m, 'AssetCovar', C, 'InitPort', pwgt0);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
```

Given `pwgt0` and `pwgt`, use the portfolio risk and return estimation methods to obtain risks and returns for your initial portfolio and the portfolios on the efficient frontier:

```
[prsk0, pret0] = estimatePortMoments(p, pwgt0);
[prsk, pret] = estimatePortMoments(p, pwgt);
```

or

```
prsk0 = estimatePortRisk(p, pwgt0);
pret0 = estimatePortReturn(p, pwgt0);
prsk = estimatePortRisk(p, pwgt);
pret = estimatePortReturn(p, pwgt);
In either case, you obtain these risks and returns:
```

```
display(prsk0);  
display(pret0);  
display(prsk);  
display(pret);
```

```
prsk0 =  
    0.1103
```

```
pret0 =  
    0.0870
```

```
prsk =  
    0.0769  
    0.0831  
    0.0994  
    0.1217  
    0.1474  
    0.1750  
    0.2068  
    0.2487  
    0.2968  
    0.3500
```

```
pret =  
    0.0590  
    0.0725  
    0.0859  
    0.0994  
    0.1128  
    0.1262  
    0.1397  
    0.1531  
    0.1666  
    0.1800
```

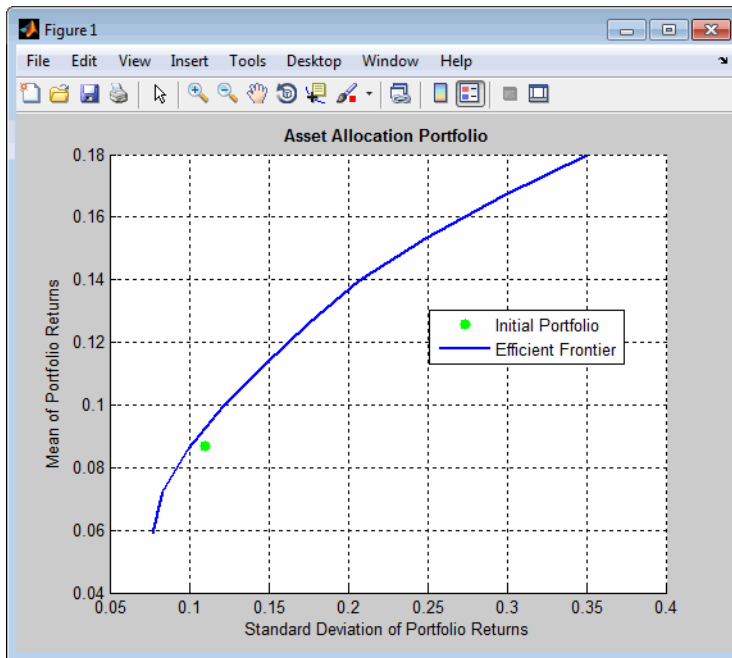
Note the returns and risks are at the periodicity of the moments of asset returns so that, if you have values for `AssetMean` and `AssetCovar` in terms of monthly returns, the estimates for portfolio risk and return are in terms of monthly returns as well. In addition, the estimate for portfolio risk in the mean-variance case is the standard deviation of portfolio returns, not the variance of portfolio returns.

Plotting the Efficient Frontier

The `plotFrontier` method creates a plot of the efficient frontier for a given portfolio optimization problem. This method accepts several types of inputs and generates a plot with an optional possibility to output the estimates for portfolio risks and returns along the efficient frontier. `plotFrontier` has four different ways that it can be used. In addition to a plot of the efficient frontier, if you have an initial portfolio in the `InitPort` property, `plotFrontier` also displays the return versus risk of the initial portfolio on the same plot. If you have a well-posed portfolio optimization problem set up in a `Portfolio` object and you use `plotFrontier`, you will get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is currently 10 and is maintained in the hidden property `defaultNumPorts`). This example illustrates a typical use of `plotFrontier` to create a new plot:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
plotFrontier(p);
```

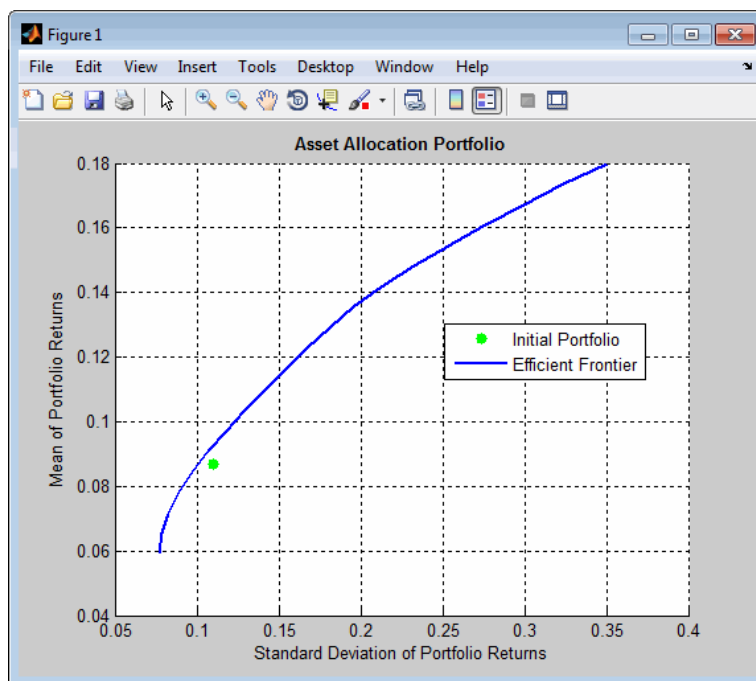
The `Name` property appears as the title of the efficient frontier plot if you set it in the `Portfolio` object. Without an explicit name, the title on the plot would be “Efficient Frontier.” If you want to obtain a specific number of portfolios along the efficient frontier, use `plotFrontier` with the number of portfolios that you want. Suppose you have the `Portfolio` object from the previous example and you want to plot 20 portfolios along the efficient frontier and to obtain 20 risk and return values for each portfolio:

```
[prsk, pret] = plotFrontier(p, 20);
display([pret, prsk]);
```

```
ans =
```

```
0.0590    0.0769
0.0654    0.0784
0.0718    0.0825
0.0781    0.0890
0.0845    0.0973
0.0909    0.1071
0.0972    0.1179
0.1036    0.1296
```

0.1100	0.1418
0.1163	0.1545
0.1227	0.1676
0.1291	0.1810
0.1354	0.1955
0.1418	0.2128
0.1482	0.2323
0.1545	0.2535
0.1609	0.2760
0.1673	0.2995
0.1736	0.3239
0.1800	0.3500



Plotting Existing Efficient Portfolios

If you already have efficient portfolios from any of the "estimateFrontier" methods (see "Estimate Efficient Portfolios" on page 4-84), pass them into `plotFrontier` directly to plot the efficient frontier:

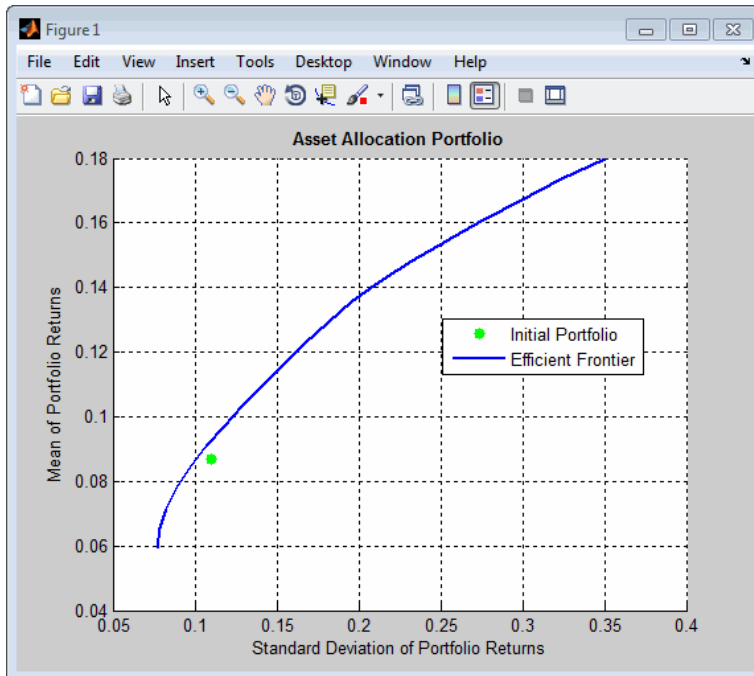
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
```

```

C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p, 20);
plotFrontier(p, pwgt);

```



Plotting Existing Efficient Portfolio Risks and Returns

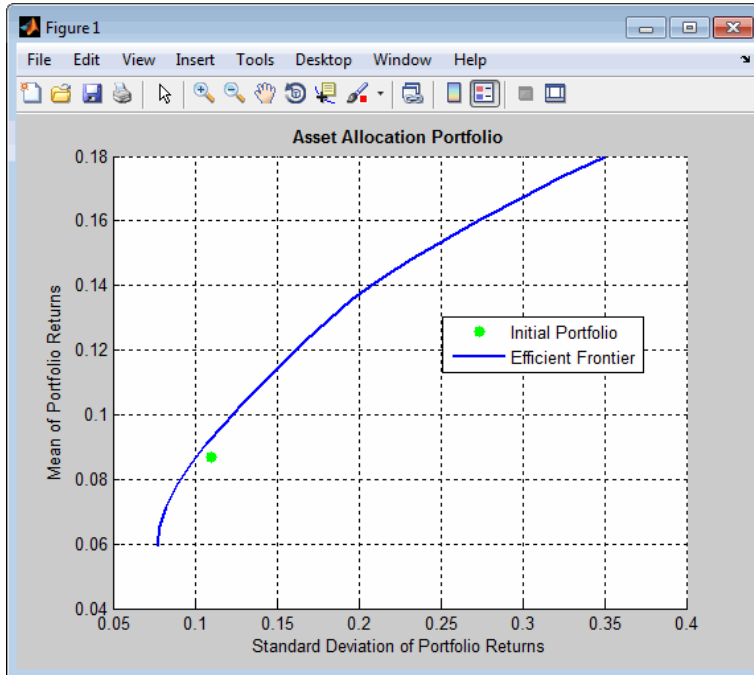
If you already have efficient portfolio risks and returns, you can use the interface to `plotFrontier` to pass them into `plotFrontier` to obtain a plot of the efficient frontier:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

```

```
p = Portfolio('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);  
p = setAssetMoments(p, m, C);  
p = setDefaultConstraints(p);  
[prsk, pret] = estimatePortMoments(p, p.estimateFrontier(20));  
plotFrontier(p, prsk, pret);
```



See Also

[Portfolio](#) | [estimatePortMoments](#) | [estimatePortReturn](#) | [plotFrontier](#)

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Estimate Efficient Portfolios” on page 4-84
- “Postprocessing Results” on page 4-104
- “Asset Allocation” on page 4-109

- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Postprocessing Results

In this section...

“Setting Up Tradable Portfolios” on page 4-104

“Troubleshooting Portfolio Optimization Results” on page 4-105

After obtaining efficient portfolios or estimates for expected portfolio risks and returns, use your results to set up trades to move toward an efficient portfolio. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Setting Up Tradable Portfolios

Suppose you set up a portfolio optimization problem and obtained portfolios on the efficient frontier. Use the `dataset` object from Statistics Toolbox™ to form a blotter that lists your portfolios with the names for each asset. For example, suppose you want to obtain five portfolios along the efficient frontier. You can set up a blotter with weights multiplied by 100 to view the allocations for each portfolio:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('InitPort', pwgt0);
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([100*pwgt],pnames,'obsnames',p.AssetList);
display(Blotter);

Blotter =
```

	Port1	Port2	Port3	Port4	Port5
Bonds	88.906	51.216	13.525	0	0
Large-Cap Equities	3.6875	24.387	45.086	27.479	0
Small-Cap Equities	4.0425	7.7088	11.375	13.759	0
Emerging Equities	3.364	16.689	30.014	58.762	100

This result indicates that you would invest primarily in bonds at the minimum-risk/minimum-return end of the efficient frontier (Port1), and that you would invest completely in emerging equity at the maximum-risk/maximum-return end of the efficient frontier (Port5). You can also select a particular efficient portfolio, for example, suppose you want a portfolio with 15% risk and you add purchase and sale weights outputs obtained from the “estimateFrontier” methods to set up a trade blotter:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('InitPort', pwgt0);
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, 0.15);

Blotter = dataset([100*[pwgt0, pwgt, pbuy, psell]}, ...
                 {'Initial', 'Weight', 'Purchases', 'Sales'}], 'obsnames', p.AssetList);

display(Blotter);

Blotter =
```

	Initial	Weight	Purchases	Sales
Bonds	30	20.299	0	9.7007
Large-Cap Equities	30	41.366	11.366	0
Small-Cap Equities	20	10.716	0	9.2838
Emerging Equities	10	27.619	17.619	0

If you have prices for each asset (in this example, they can be ETFs), add them to your blotter and then use the tools of the `dataset` object to obtain shares and shares to be traded. For an example, see “Asset Allocation” on page 4-109.

Troubleshooting Portfolio Optimization Results

Portfolio Object Destroyed When Modifying

If a Portfolio object is destroyed when modifying, remember to pass an existing object into the constructor if you want to modify it, otherwise it creates a new object. See “Constructing the Portfolio Object” on page 4-26 for details.

Optimization Fails with “Bad Pivot” Message

If the optimization fails with a “bad pivot” message from `lcprog`, try a larger value for `tolpiv` which is a tolerance for pivot selection in the `lcprog` algorithm (try `1.0e-7`,

for example) or try the `interior-point-convex` version of `quadprog`. For details, see “Choosing and Controlling the Solver” on page 4-94, the help header for `lcprog`, and the `quadprog` documentation.

Speed of Optimization

Although it is difficult to characterize when one algorithm is faster than the other, the default solver, `lcprog` is generally faster for smaller problems and the `quadprog` solver is generally faster for larger problems. If one solver seems to take too much time, try the other solver. To change solvers, use `setSolver`.

Matrix Incompatibility and "Non-Conformable" Errors

If you get matrix incompatibility or "non-conformable" errors, the representation of data in the tools follows a specific set of basic rules described in “Conventions for Representation of Data” on page 4-24.

Missing Data Estimation Fails

If asset return data has missing or NaN values, the method `estimateAssetMoments` with the `'missingdata'` flag set to `true` may fail with either too many iterations or a singular covariance. To correct this problem, consider this:

- If you have asset return data with no missing or NaN values, you can compute a covariance matrix that may be singular without difficulties. If you have missing or NaN values in your data, the supported missing data feature requires that your covariance matrix must be positive-definite, i.e., nonsingular.
- `estimateAssetMoments` uses default settings for the missing data estimation procedure that might not be appropriate for all problems.

In either case, you might want to estimate the moments of asset returns separately with either the ECM estimation functions such as `ecmmle` or with your own methods.

`mv_optim_transform` Errors

If you obtain optimization errors such as:

```
Error using mv_optim_transform (line 233)
Portfolio set appears to be either empty or unbounded. Check constraints.
```

```
Error in Portfolio/estimateFrontier (line 63)
[A, b, f0, f, H, g, lb] = mv_optim_transform(obj);
or
```



```
Error using mv_optim_transform (line 238)
Cannot obtain finite lower bounds for specified portfolio set.
```

```
Error in Portfolio/estimateFrontier (line 63)
[A, b, f0, f, H, g, lb] = mv_optim_transform(obj);
```

Since the portfolio optimization tools require a bounded portfolio set, these errors (and similar errors) can occur if your portfolio set is either empty and, if nonempty, unbounded. Specifically, the portfolio optimization algorithm requires that your portfolio set have at least a finite lower bound. The best way to deal with these problems is to use the validation methods in “Validate the Portfolio Problem” on page 4-79. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

If you obtain efficient portfolios that do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between 0 and 1 and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

See Also

Portfolio | `checkFeasibility` | `estimateAssetMoments`

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Estimate Efficient Portfolios” on page 4-84

- “Estimate Efficient Frontiers” on page 4-96
- “Asset Allocation” on page 4-109
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Asset Allocation

In this section...

“Defining the Portfolio Problem” on page 4-109

“Simulating Asset Prices” on page 4-110

“Setting Up the Portfolio Object” on page 4-111

“Validating the Portfolio Problem” on page 4-113

“Plotting the Efficient Frontier” on page 4-113

“Evaluating Gross vs. Net Portfolio Returns” on page 4-114

“Analyzing Descriptive Properties of the Portfolio Structures” on page 4-115

“Obtaining a Portfolio at the Specified Return Level on the Efficient Frontier” on page 4-116

“Obtaining a Portfolio at the Specified Risk Levels on the Efficient Frontier” on page 4-116

“Displaying the Final Results” on page 4-118

The following example sets up a basic asset allocation problem to use mean-variance portfolio optimization to estimate efficient portfolios. Suppose you want to manage an asset allocation fund with four asset classes: bonds, large-cap equities, small-cap equities, and emerging equities. The fund is long-only with no borrowing or leverage, should have no more than 85% of the portfolio in equities, and no more than 35% of the portfolio in emerging equities.

The cost to trade the first three assets is 10 basis points annualized and the cost to trade emerging equities is four times higher. Finally, you want to ensure that average turnover is no more than 15%. To solve this problem, you will set up a basic mean-variance portfolio optimization problem and then slowly introduce the various constraints on the problem to get to a solution.

For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Defining the Portfolio Problem

To set up the portfolio optimization problem, start with basic definitions of known quantities associated with the structure of this problem. Each asset class is assumed to have a tradeable asset with a real-time price. Such assets can be, for example, exchange-

traded funds (ETFs). The initial portfolio with holdings in each asset that has a total of \$7.5 million along with an additional cash position of \$60,000. These basic quantities and the costs to trade are set up in the following variables with asset names in the cell array `Asset`, current prices in the vector `Price`, current portfolio holdings in the vector `Holding`, and transaction costs in the vector `UnitCost`.

```
Asset = { 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities' };
Price = [ 52.4; 122.7; 35.2; 46.9 ];
Holding = [ 42938; 24449; 42612; 15991 ];
UnitCost = [ 0.001; 0.001; 0.001; 0.004 ];
```

To analyze this portfolio, you can set up a blotter in a `dataset` object to help track prices, holdings, weights, and so forth. In particular, you can compute the initial portfolio weights and maintain them in a new blotter field called `InitPort`.

```
Blotter = dataset({Price, 'Price'}, {Holding, 'InitHolding'}, 'obsnames', Asset);
Wealth = sum(Blotter.Price .* Blotter.InitHolding);
Blotter.InitPort = (1/Wealth)*(Blotter.Price .* Blotter.InitHolding);
Blotter.UnitCost = UnitCost;
disp(Blotter);
```

	Price	InitHolding	InitPort	UnitCost
Bonds	52.4	42938	0.3	0.001
Large-Cap Equities	122.7	24449	0.4	0.001
Small-Cap Equities	35.2	42612	0.2	0.001
Emerging Equities	46.9	15991	0.1	0.004

Simulating Asset Prices

Since this is a hypothetical example, to simulate asset prices from a given mean and covariance of annual asset total returns for the asset classes, `portsim` is used to create asset returns with the desired mean and covariance. Specifically, `portsim` is used to simulate 5 years of monthly total returns. The mean and covariance of annual asset total returns are maintained in the variables `AssetMean` and `AssetCovar`. The simulated asset total return prices (which are compounded total returns) are maintained in the variable `Y`. All initial asset total return prices are normalized to 1 in this example.

```
AssetMean = [ 0.05; 0.1; 0.12; 0.18 ];
AssetCovar = [ 0.0064 0.00408 0.00192 0;
               0.00408 0.0289 0.0204 0.0119;
               0.00192 0.0204 0.0576 0.0336;
               0 0.0119 0.0336 0.1225 ];

X = portsim(AssetMean'/12, AssetCovar/12, 60); % monthly total returns for 5 years (60 months)
[Y, T] = ret2tick(X, [], 1/12); % form total return prices
```

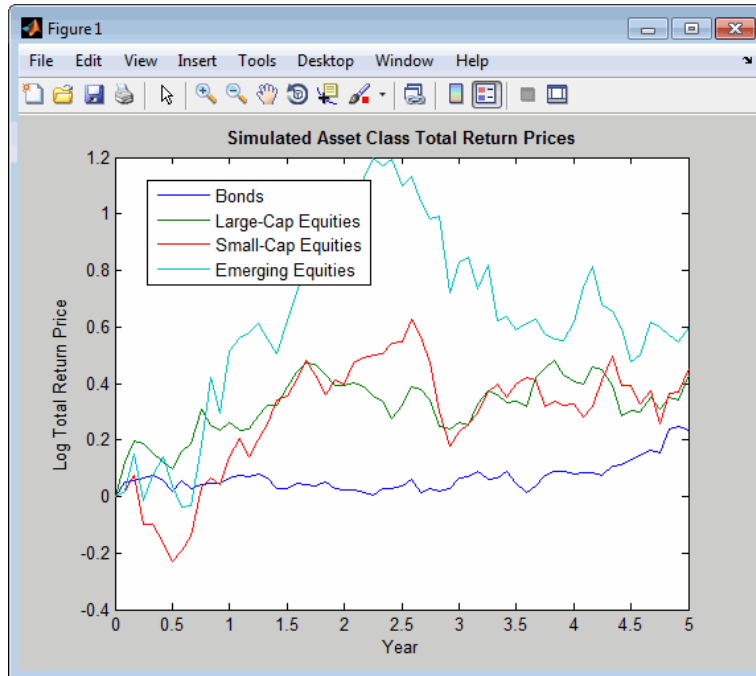
This plot shows the log of the simulated total return prices:

```
plot(T, log(Y));
```

```

title('\bfSimulated Asset Class Total Return Prices');
xlabel('Year');
ylabel('Log Total Return Price');
legend(Asset, 'Location', 'best');

```



If working with actual historical asset prices, income, and corporate actions data, you would compute total returns for your assets by other means.

Setting Up the Portfolio Object

To explore portfolios on the efficient frontier, set up a Portfolio object using these specifications:

- Portfolio weights are nonnegative and sum to 1.
- Equity allocation is no more than 85% of the portfolio.
- Emerging equity is no more than 35% of the portfolio.

These specifications are incorporated into the Portfolio object p in the following sequence of methods that starts with the portfolio constructor:

```
p = Portfolio('Name', 'Asset Allocation Portfolio', ...  
            'AssetList', Asset, 'InitPort', Blotter.InitPort);
```

The specification of the initial portfolio from `Blotter` gives the number of assets in your universe so you do not need to specify the `NumAssets` property directly. Next, set up default constraints (long-only with a budget constraint). In addition, set up the group constraint that imposes an upper bound on equities in the portfolio (equities are identified in the group matrix with 1s) and the upper bound constraint on emerging equities.

```
p = setDefaultConstraints(p);  
p = setGroups(p, [ 0, 1, 1, 1 ], [], 0.85);  
p = addGroups(p, [ 0, 0, 0, 1 ], [], 0.35);
```

Although you could have set the upper bound on emerging equities using the `setBounds` method, notice how you used the `addGroups` method to set up this constraint.

Finally, to have a fully specified mean-variance portfolio optimization problem, you must specify the mean and covariance of asset returns. Since starting with these moments in the variables `AssetMean` and `AssetCovar`, you could use the method `setAssetMoments` to enter these variables into your `Portfolio` object in the following way (remember that you are assuming that your raw data are monthly returns which is why you divide your annual input moments by 12 to get monthly returns).

```
p = setAssetMoments(p, AssetMean/12, AssetCovar/12);
```

To make things more interesting, however, you can use the total return prices and use the method `estimateAssetMoments` with a specification that your data in `Y` are prices, and not returns, to estimate asset return moments for your `Portfolio` object.

```
p = estimateAssetMoments(p, Y, 'DataFormat', 'Prices');
```

Although the returns in your `Portfolio` object are in units of monthly returns, and since subsequent costs are annualized, it is convenient to specify them as annualized total returns with this direct transformation of the `AssetMean` and `AssetCovar` properties of your object:

```
p.AssetMean = 12*p.AssetMean;  
p.AssetCovar = 12*p.AssetCovar;
```

Now, the `Portfolio` object is ready:

```
display(p);  
p =
```

```

Portfolio
Properties:
  BuyCost: []
  SellCost: []
  RiskFreeRate: []
  AssetMean: [4x1 double]
  AssetCovar: [4x4 double]
  Turnover: []
  Name: 'Asset Allocation Portfolio'
  NumAssets: 4
  AssetList: {'Bonds' 'Large-Cap Equities' 'Small-Cap Equities' 'Emerging Equities'}
  InitPort: [4x1 double]
  AInequality: []
  bInequality: []
  AEquality: []
  bEquality: []
  LowerBound: [4x1 double]
  UpperBound: []
  LowerBudget: 1
  UpperBudget: 1
  GroupMatrix: [2x4 double]
  LowerGroup: []
  UpperGroup: [2x1 double]
  GroupA: []
  GroupB: []
  LowerRatio: []
  UpperRatio: []

```

Validating the Portfolio Problem

An important step in portfolio optimization is to validate that the portfolio problem is feasible and the main test is to ensure that the set of portfolios is nonempty and bounded. Use the `estimateBounds` method to determine the bounds for the portfolio set:

```
[lb, ub] = estimateBounds(p);
display([lb, ub]);
```

```
ans =
```

```

0.1500    1.0000
0.0000    0.8500
0.0000    0.8500
0.0000    0.3500

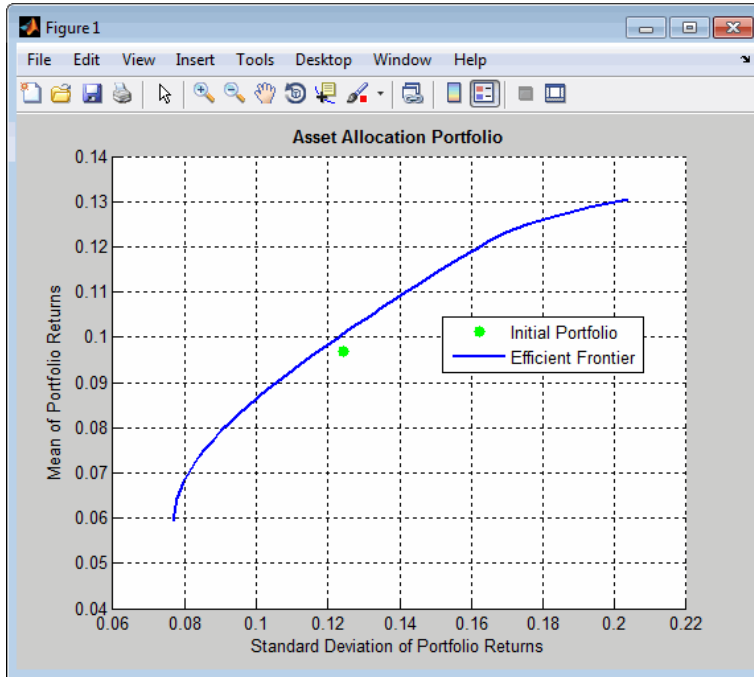
```

Since both `lb` and `ub` are finite, the set is bounded.

Plotting the Efficient Frontier

Given the constructed Portfolio object, use the method `plotFrontier` to view the efficient frontier. Instead of using the default of 10 portfolios along the frontier, you can display the frontier with 40 portfolios:

```
plotFrontier(p, 40);
```



Notice gross efficient portfolio returns fall between approximately 6% and 16% per years.

Evaluating Gross vs. Net Portfolio Returns

The Portfolio object `p` does not include transaction costs so that the portfolio optimization problem specified in `p` uses gross portfolio return as the return proxy. To handle net returns, create a second Portfolio object `q` that includes transaction costs:

```
q = setCosts(p, UnitCost, UnitCost);
display(q);
```

```
q =
Portfolio
Properties:
  BuyCost: [4x1 double]
  SellCost: [4x1 double]
  RiskFreeRate: []
  AssetMean: [4x1 double]
  AssetCovar: [4x4 double]
```



```

Turnover: []
  Name: 'Asset Allocation Portfolio'
NumAssets: 4
AssetList: {'Bonds' 'Large-Cap Equities' 'Small-Cap Equities' 'Emerging Equities'}
  InitPort: [4x1 double]
AInequality: []
bInequality: []
  AEquality: []
  bEquality: []
LowerBound: [4x1 double]
UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: [2x4 double]
  LowerGroup: []
  UpperGroup: [2x1 double]
    GroupA: []
    GroupB: []
LowerRatio: []
UpperRatio: []

```

Analyzing Descriptive Properties of the Portfolio Structures

To be more concrete about the ranges of efficient portfolio returns and risks, use the method `estimateFrontierLimits` to obtain portfolios at the endpoints of the efficient frontier. Given these portfolios, compute their moments using `estimatePortMoments`. The following code generates a table that lists the risk and return of the initial portfolio as well as the gross and net moments of portfolio returns for the portfolios at the endpoints of the efficient frontier:

```

[prsk0, pret0] = estimatePortMoments(p, p.InitPort);

pret = estimatePortReturn(p, p.estimateFrontierLimits);
qret = estimatePortReturn(q, q.estimateFrontierLimits);

fprintf('Annualized Portfolio Returns ... \n');
fprintf('          %6s      %6s \n', 'Gross', 'Net');
fprintf('Initial Portfolio Return      %6.2f %%   %6.2f %% \n', 100*pret0, 100*pret0);
fprintf('Minimum Efficient Portfolio Return %6.2f %%   %6.2f %% \n', 100*pret(1), 100*qret(1));
fprintf('Maximum Efficient Portfolio Return %6.2f %%   %6.2f %% \n', 100*pret(2), 100*qret(2));

```

```

Annualized Portfolio Returns ...
Initial Portfolio Return      9.70 %   9.70 %
Minimum Efficient Portfolio Return  5.90 %   5.77 %
Maximum Efficient Portfolio Return 13.05 %  12.86 %

```

This result shows that the cost to trade ranges from 14 to 19 basis points to get from the current portfolio to the efficient portfolios at the endpoints of the efficient frontier (these costs are the difference between gross and net portfolio returns.) In addition, notice that the maximum efficient portfolio return (13%) is less than the maximum asset return (18%) due to the constraints on equity allocations.

Obtaining a Portfolio at the Specified Return Level on the Efficient Frontier

A common approach to select efficient portfolios is to pick a portfolio that has a desired fraction of the range of expected portfolio returns. To obtain the portfolio that is 30% of the range from the minimum to maximum return on the efficient frontier, obtain the range of net returns in `qret` using the Portfolio object `q` and interpolate to obtain a 30% level with `interp1` to obtain a portfolio `qwgt`:

```
Level = 0.3;

qret = estimatePortReturn(q, q.estimateFrontierLimits);
qwgt = estimateFrontierByReturn(q, interp1([0, 1], qret, Level));
[qrsk, qret] = estimatePortMoments(q, qwgt);

fprintf('Portfolio at %g%% return level on efficient frontier ...\n', 100*Level);
fprintf('%10s %10s\n', 'Return', 'Risk');
fprintf('%10.2f %10.2f\n', 100*qret, 100*qrsk);

display(qwgt);

Portfolio at 30% return level on efficient frontier ...
      Return      Risk
      7.90      9.09

qwgt =

    0.6252
    0.1856
    0.0695
    0.1198
```

The target portfolio that is 30% of the range from minimum to maximum net returns has a return of 7.9% and a risk of 9.1%.

Obtaining a Portfolio at the Specified Risk Levels on the Efficient Frontier

Although you could accept this result, suppose you want to target values for portfolio risk. Specifically, suppose you have a conservative target risk of 10%, a moderate target risk of 15%, and an aggressive target risk of 20% and you want to obtain portfolios that satisfy each risk target. Use the `estimateFrontierByRisk` method to obtain targeted risks specified in the variable `TargetRisk`. The resultant three efficient portfolios are obtained in `qwgt`:

```
TargetRisk = [ 0.10; 0.15; 0.20 ];
qwgt = estimateFrontierByRisk(q, TargetRisk);
display(qwgt);

qwgt =
```

```

0.5407    0.2020    0.1500
0.2332    0.4000    0.0318
0.0788    0.1280    0.4682
0.1474    0.2700    0.3500

```

Use `estimatePortRisk` to compute the portfolio risks for the three portfolios to confirm that the target risks have been attained:

```
display(estimatePortRisk(q, qwgt));
```

```
ans =
```

```

0.1000
0.1500
0.2000

```

Suppose you want to shift from the current portfolio to the moderate portfolio. You can estimate the purchases and sales to get to this portfolio:

```
[qwgt, qbuy, qsell] = estimateFrontierByRisk(q, 0.15);
```

If you average the purchases and sales for this portfolio, you can see that the average turnover is 17%, which is greater than the target of 15%:

```
disp(sum(qbuy + qsell)/2)
```

```
0.1700
```

Since you also want to ensure that average turnover is no more than 15%, you can add the average turnover constraint to the Portfolio object:

```
q = setTurnover(q, 0.15);
[qwgt, qbuy, qsell] = estimateFrontierByRisk(q, 0.15);
```

You can enter the estimated efficient portfolio with purchases and sales into the Blotter:

```
qbuy(abs(qbuy) < 1.0e-5) = 0;
qsell(abs(qsell) < 1.0e-5) = 0; % zero out near 0 trade weights
```

```
Blotter.Port = qwgt;
Blotter.Buy = qbuy;
Blotter.Sell = qsell;
```

```
display(Blotter);
```

```
Blotter =
```

	Price	InitHolding	InitPort	UnitCost	Port	Buy	Sell
Bonds	52.4	42938	0.3	0.001	0.18787	0	0.11213

Large-Cap Equities	122.7	24449	0.4	0.001	0.4	0	0
Small-Cap Equities	35.2	42612	0.2	0.001	0.16213	0	0.037871
Emerging Equities	46.9	15991	0.1	0.004	0.25	0.15	0

The **Buy** and **Sell** elements of the **Blotter** are changes in portfolio weights that must be converted into changes in portfolio holdings to determine the trades. Since you are working with net portfolio returns, you must first compute the cost to trade from your initial portfolio to the new portfolio. This can be accomplished as follows:

```
TotalCost = Wealth * sum(Blotter.UnitCost .* (Blotter.Buy + Blotter.Sell))
```

```
TotalCost =
```

```
5.6248e+003
```

The cost to trade is \$5,625, so that, in general, you would have to adjust your initial wealth accordingly before setting up your new portfolio weights. However, to keep the analysis simple, note that you have sufficient cash (\$60,0000) set aside to pay the trading costs and that you will not touch the cash position to build up any positions in your portfolio. Thus, you can populate your blotter with the new portfolio holdings and the trades to get to the new portfolio without making any changes in your total invested wealth.

First, compute portfolio holding:

```
Blotter.Holding = Wealth * (Blotter.Port ./ Blotter.Price);
```

Next, compute number of shares to **Buy** and **Sell** in your **Blotter**:

```
Blotter.BuyShare = Wealth * (Blotter.Buy ./ Blotter.Price);  
Blotter.SellShare = Wealth * (Blotter.Sell ./ Blotter.Price);
```

Notice how you used an add-hoc truncation rule to obtain unit numbers of shares to buy and sell.

Finally, clean up the blotter by removing the unit costs and the buy and sell portfolio weights:

```
Blotter.Buy = [];  
Blotter.Sell = [];  
Blotter.UnitCost = [];
```

Displaying the Final Results

The final result is a blotter that contains proposed trades to get from your current portfolio to a moderate-risk portfolio. To make the trade, you would need to sell 16,049

shares of your bond asset and 8,069 shares of your small-cap equity asset and would need to purchase 23,986 shares of your emerging equities asset.

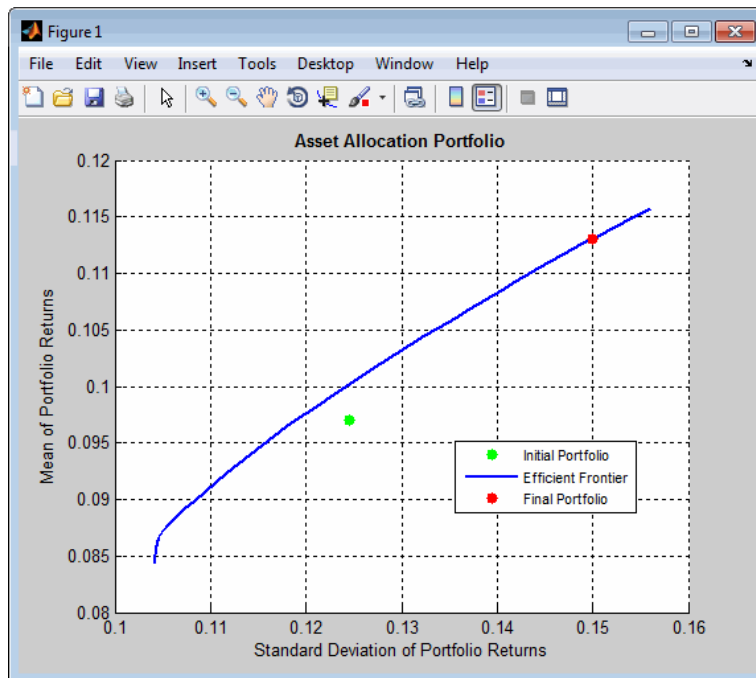
```
display(Blotter);
```

```
Blotter =
```

	Price	InitHolding	InitPort	Port	Holding	BuyShare	SellShare
Bonds	52.4	42938	0.3	0.18787	26889	0	16049
Large-Cap Equities	122.7	24449	0.4	0.4	24449	0	0
Small-Cap Equities	35.2	42612	0.2	0.16213	34543	0	8068.8
Emerging Equities	46.9	15991	0.1	0.25	39977	23986	0

The final plot uses `plotFrontier` to display the efficient frontier and the initial portfolio for the fully specified portfolio optimization problem. It also adds the location of the moderate-risk or final portfolio on the efficient frontier.

```
plotFrontier(q, 40);
hold on
scatter(estimatePortRisk(q, qwgt), estimatePortReturn(q, qwgt), 'filled', 'r');
h = legend('Initial Portfolio', 'Efficient Frontier', 'Final Portfolio', 'location', 'best');
set(h, 'FontSize', 8);
hold off
```



See Also

`Portfolio` | `addGroups` | `estimateAssetMoments` | `estimateBounds` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortRisk` | `plotFrontier` | `setAssetMoments` | `setBounds`

Related Examples

- “Constructing the Portfolio Object” on page 4-26
- “Working with Portfolio Constraints” on page 4-57
- “Validate the Portfolio Problem” on page 4-79
- “Estimate Efficient Portfolios” on page 4-84
- “Estimate Efficient Frontiers” on page 4-96
- “Postprocessing Results” on page 4-104
- “Portfolio Optimization Examples” on page 4-121

More About

- “Portfolio Object” on page 4-17
- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object Workflow”

Portfolio Optimization Examples

The following sequence of examples highlights features of the Portfolio object in the Financial Toolbox™. Specifically, the examples show how to set up mean-variance portfolio optimization problems that focus on the two-fund theorem, the impact of transaction costs and turnover constraints, how to obtain portfolios that maximize the Sharpe ratio, and how to set up two popular hedge-fund strategies - dollar-neutral and 130-30 portfolios.

Set up the Data

Every example works with moments for monthly total returns of a universe of 30 "blue-chip" stocks. Although derived from real data, these data are for illustrative purposes and are not meant to be representative of specific assets or of market performance. The data are contained in the file `BlueChipStockMoments.mat` with a list of asset identifiers in the variable `AssetList`, a mean and covariance of asset returns in the variables `AssetMean` and `AssetCovar`, and the mean and variance of cash and market returns in the variables `CashMean`, `CashVar`, `MarketMean`, and `MarketVar`. Since most of the analysis requires the use of the standard deviation of asset returns as the proxy for risk, cash and market variances are converted into standard deviations.

```
load BlueChipStockMoments
```

```
mret = MarketMean;  
mrsk = sqrt(MarketVar);  
cret = CashMean;  
crsk = sqrt(CashVar);
```

Create a Portfolio Object

The first step is to create a "standard" Portfolio object with the Portfolio constructor and to incorporate the list of assets, the risk-free rate, and the moments of asset returns into the object.

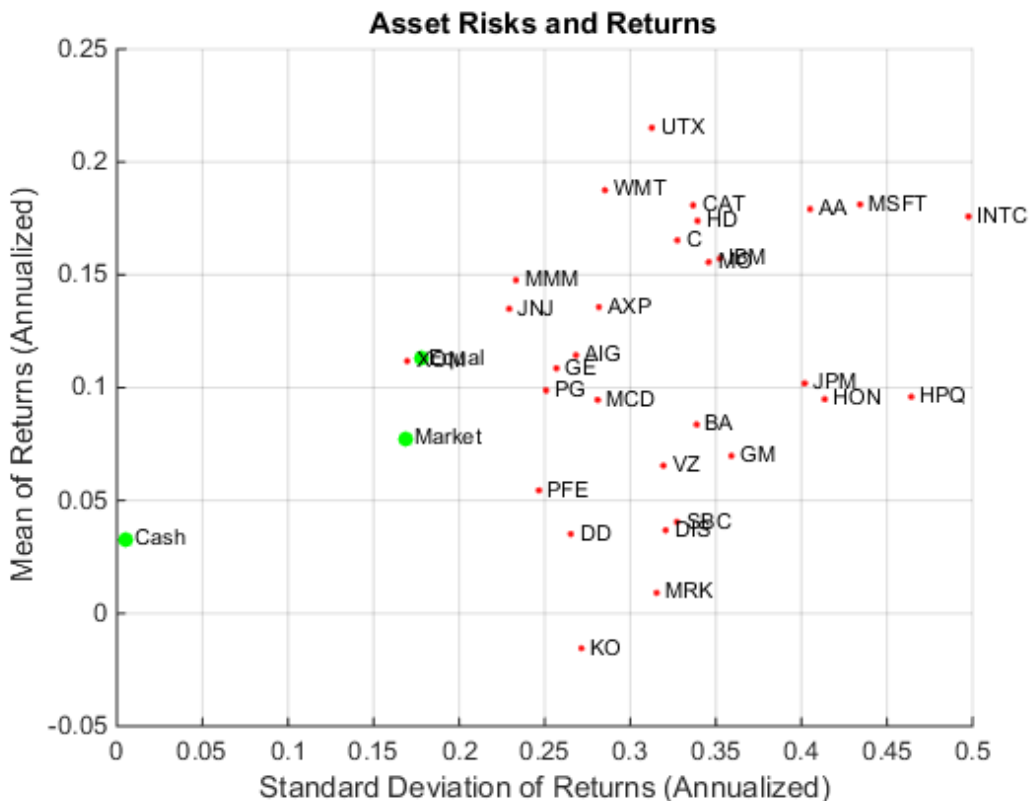
```
p = Portfolio('AssetList', AssetList, 'RiskFreeRate', CashMean);  
p = setAssetMoments(p, AssetMean, AssetCovar);
```

To provide a basis for comparison, set up an equal-weight portfolio and make it the initial portfolio in the Portfolio object. Keep in mind that the hedged portfolios to be constructed later will require a different initial portfolio. Once the initial portfolio is created, the `estimatePortMoments` method estimates the mean and standard deviation of equal-weight portfolio returns.

```
p = setInitPort(p, 1/p.NumAssets);
[ersk, eret] = estimatePortMoments(p, p.InitPort);
```

A specialized "helper" function `portfolioexamples_plot` makes it possible to plot all results to be developed here. This first plot shows the distribution of individual assets according to their means and standard deviations of returns. In addition, the equal-weight, market, and cash portfolios are plotted on the same plot. Note that the plot function converts monthly total returns into annualized total returns.

```
clf;
portfolioexamples_plot('Asset Risks and Returns', ...
  {'scatter', mrsk, mret, {'Market'}}, ...
  {'scatter', crsk, cret, {'Cash'}}, ...
  {'scatter', ersk, eret, {'Equal'}}, ...
  {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



Set up a Portfolio Optimization Problem

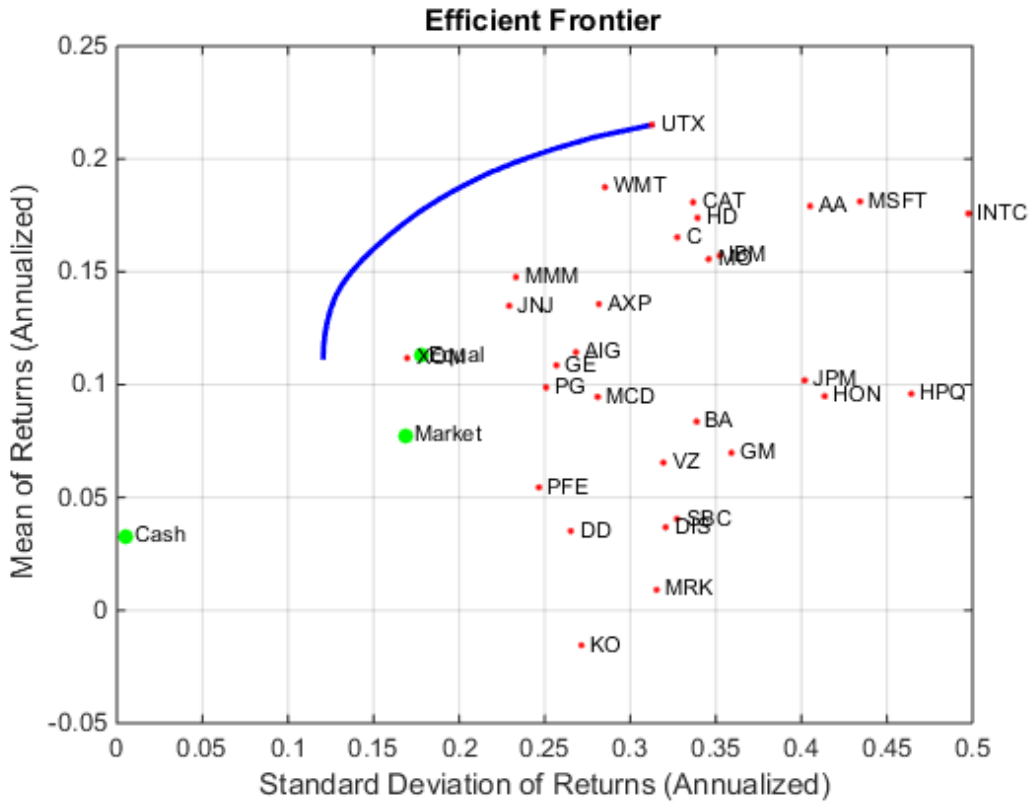
Set up a "standard" or default mean-variance portfolio optimization problem with the `setDefaultConstraints` method that requires fully-invested long-only portfolios (non-negative weights that must sum to 1). Given this initial problem, estimate the efficient frontier with the methods `estimateFrontier` and `estimatePortMoments`, where `estimateFrontier` estimates efficient portfolios and `estimatePortMoments` estimates risks and returns for portfolios. The next figure overlays the efficient frontier on the previous plot.

```
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p, 20);
[prsk, pret] = estimatePortMoments(p, pwgt);

% Plot efficient frontier

clf;
portfolioexamples_plot('Efficient Frontier', ...
    {'line', prsk, pret}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



Illustrate the Tangent Line to the Efficient Frontier

Tobin's mutual fund theorem (Tobin 1958) says that the portfolio allocation problem can be viewed as a decision to allocate between a riskless asset and a risky portfolio. In the mean-variance framework, cash can serve as a proxy for a riskless asset and an efficient portfolio on the efficient frontier serves as the risky portfolio such that any allocation between cash and this portfolio dominates all other portfolios on the efficient frontier. This portfolio is called a tangency portfolio because it is located at the point on the efficient frontier where a tangent line that originates at the riskless asset touches the efficient frontier.

Given that the Portfolio object already has the risk-free rate, obtain the tangent line by creating a copy of the Portfolio object with a budget constraint that permits allocation

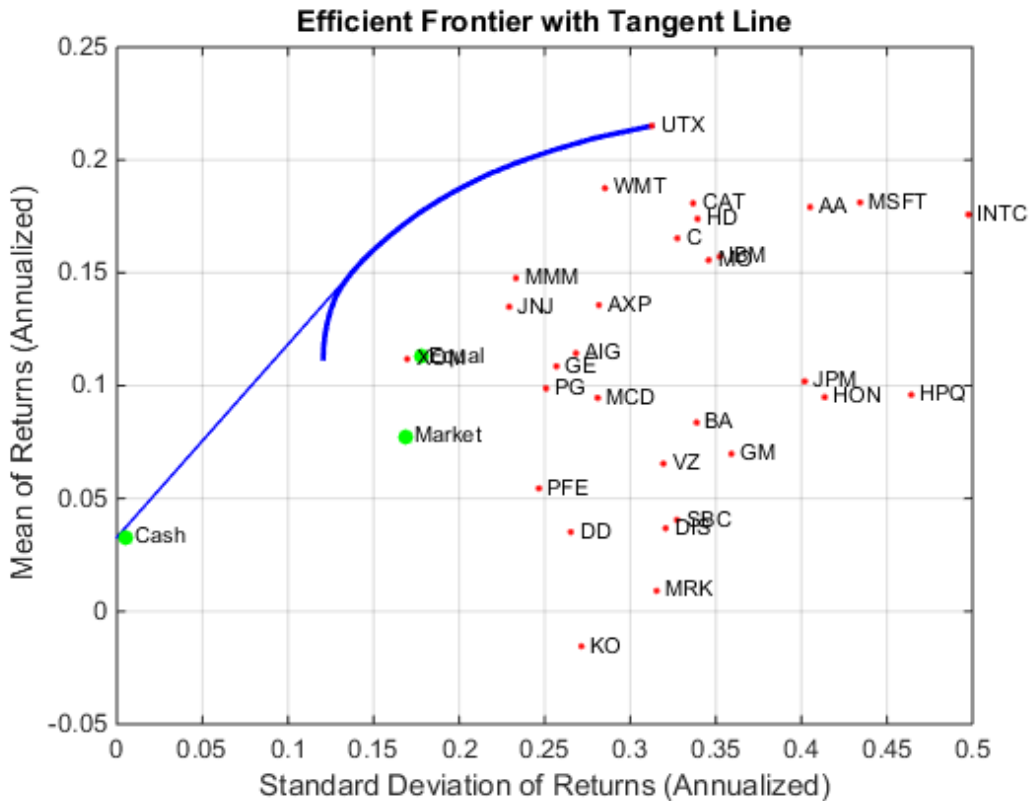
between 0% and 100% in cash. Since the Portfolio object is a value object, it is easy to create a copy by assigning the output of either the constructor or set methods to a new instance of the object. The plot shows the efficient frontier with Tobin's allocations that form the tangent line to the efficient frontier.

```
q = setBudget(p, 0, 1);

qwt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwt);

% Plot efficient frontier with tangent line (0 to 1 cash)

clf;
portfolioexamples_plot('Efficient Frontier with Tangent Line', ...
    {'line', prsk, pret}, ...
    {'line', qrsk, qret, [], [], 1}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetsList, '.r'});
```



Note that cash actually has a small risk so that the tangent line does not pass through the cash asset.

Obtain Range of Risks and Returns

To obtain efficient portfolios with target values of either risk or return, it is necessary to obtain the range of risks and returns among all portfolios on the efficient frontier. This can be accomplished with the `estimateFrontierLimits` method.

```
[rsk, ret] = estimatePortMoments(p, estimateFrontierLimits(p));

display(rsk);
display(ret);
```

```

rsk =

    0.0348
    0.0903

ret =

    0.0094
    0.0179

```

The range of monthly portfolio returns is between 0.9% and 1.8% and the range for portfolio risks is between 3.5% and 9.0%. In annualized terms, the range of portfolio returns is 11.2% to 21.5% and the range of portfolio risks is 12.1% to 31.3%.

Find a Portfolio with a Targeted Return and Targeted Risk

Given the range of risks and returns, it is possible to locate specific portfolios on the efficient frontier that have target values for return and risk using the methods `estimateFrontierByReturn` and `estimateFrontierByRisk`.

```

TargetReturn = 0.20;           % input target annualized return and risk here
TargetRisk = 0.15;

% Obtain portfolios with targeted return and risk

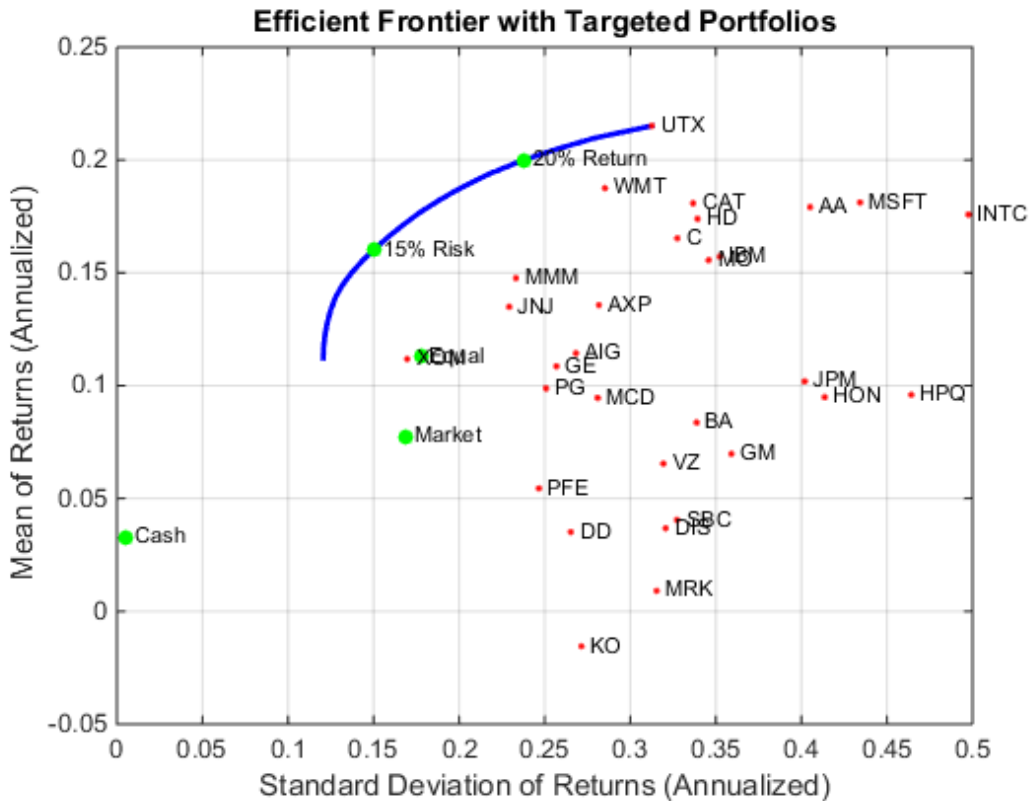
awgt = estimateFrontierByReturn(p, TargetReturn/12);
[arsk, aret] = estimatePortMoments(p, awgt);

bwgt = estimateFrontierByRisk(p, TargetRisk/sqrt(12));
[brsk, bret] = estimatePortMoments(p, bwgt);

% Plot efficient frontier with targeted portfolios

clf;
portfolioexamples_plot('Efficient Frontier with Targeted Portfolios', ...
    {'line', prsk, pret}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', arsk, aret, {sprintf('%g%% Return', 100*TargetReturn)}}, ...
    {'scatter', brsk, bret, {sprintf('%g%% Risk', 100*TargetRisk)}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});

```



To see what these targeted portfolios look like, use the dataset object to set up "blotters" that contain the portfolio weights and asset names (which are obtained from the Portfolio object).

```
aBlotter = dataset({100*awgt(awgt > 0), 'Weight'}, 'obsnames', p.AssetList(awgt > 0));
fprintf('Portfolio with %g%% Target Return\n', 100*TargetReturn);
disp(aBlotter);

bBlotter = dataset({100*bwgt(bwgt > 0), 'Weight'}, 'obsnames', p.AssetList(bwgt > 0));
fprintf('Portfolio with %g%% Target Risk\n', 100*TargetRisk);
disp(bBlotter);

Portfolio with 20% Target Return
```

	Weight
CAT	1.1445
INTC	0.17452
MO	9.6521
MSFT	0.85862
UTX	56.918
WMT	31.253

Portfolio with 15% Target Risk

	Weight
INTC	2.2585
JNJ	9.2162
MMM	16.603
MO	15.388
MSFT	4.4467
PG	4.086
UTX	10.281
WMT	25.031
XOM	12.69

Transactions Costs

The Portfolio object makes it possible to account for transaction costs as part of the optimization problem. Although individual costs can be set for each asset, use the scalar expansion features of the Portfolio object's methods to set up uniform transaction costs across all assets and compare efficient frontiers with gross versus net portfolio returns.

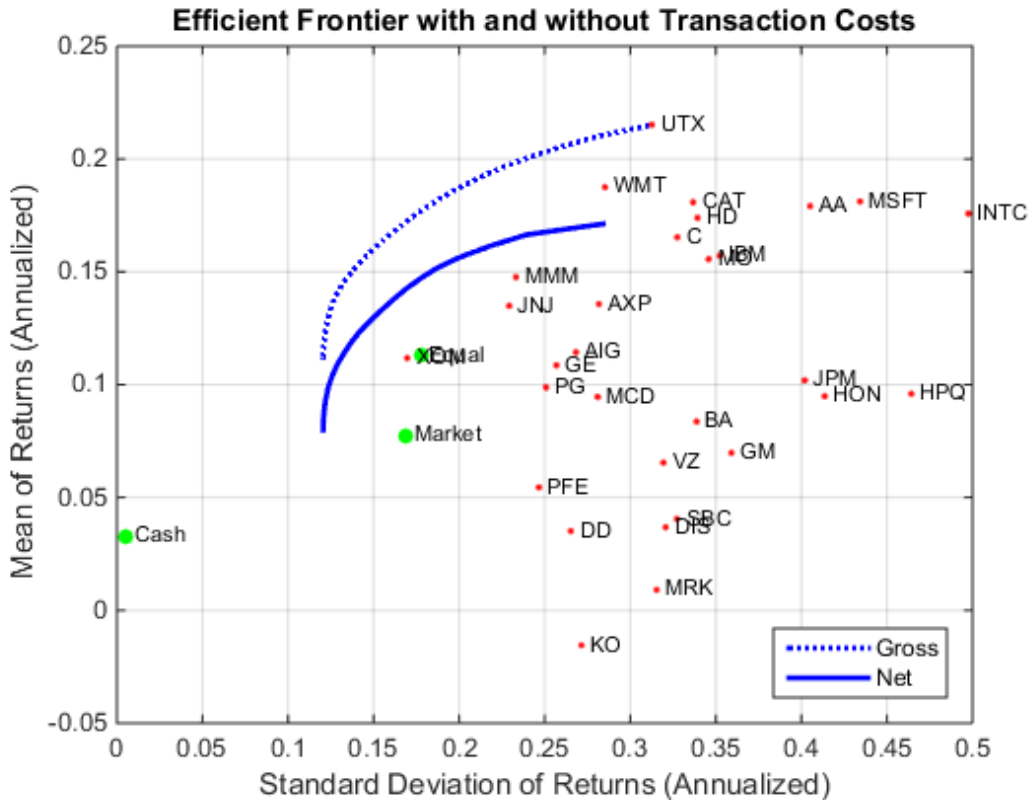
```
BuyCost = 0.0020;
SellCost = 0.0020;
```

```
q = setCosts(p, BuyCost, SellCost);
```

```
qwgts = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgts);
```

```
% Plot efficient frontiers with gross and net returns
```

```
clf;
portfolioexamples_plot('Efficient Frontier with and without Transaction Costs', ...
    {'line', prsk, pret, {'Gross'}, ':b'}, ...
    {'line', qrsk, qret, {'Net'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



Turnover Constraint

In addition to transaction costs, the Portfolio object can handle turnover constraints. The following example demonstrates that a turnover constraint produces an efficient frontier in the neighborhood of an initial portfolio that may restrict trading. Moreover, the introduction of a turnover constraint often implies that multiple trades may be necessary to shift from an initial portfolio to an unconstrained efficient frontier. Consequently, the turnover constraint introduces a form of time diversification that can spread trades out over multiple time periods. In this example, note that the sum of purchases and sales from the estimateFrontier method confirms that the turnover constraint has been satisfied.

```
BuyCost = 0.0020;
```



```

SellCost = 0.0020;
Turnover = 0.2;

q = setCosts(p, BuyCost, SellCost);
q = setTurnover(q, Turnover);

[qwgt, qbuy, qsell] = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

% Plot efficient frontier with turnover constraint

clf;
portfolioexamples_plot('Efficient Frontier with Turnover Constraint', ...
    {'line', prsk, pret, {'Unconstrained'}, ':b'}, ...
    {'line', qrsk, qret, {sprintf('%g%% Turnover', 100*Turnover)}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});

fprintf('Sum of Purchases by Portfolio along Efficient Frontier (Max. Turnover %g%%)\n',
    100*Turnover);

disp(100*sum(qbuy));

fprintf('Sum of Sales by Portfolio along Efficient Frontier (Max. Turnover %g%%)\n',
    100*Turnover);

disp(100*sum(qsell));

Sum of Purchases by Portfolio along Efficient Frontier (Max. Turnover 20%)
Columns 1 through 7

    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000

Columns 8 through 14

    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000

Columns 15 through 20

    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000

Sum of Sales by Portfolio along Efficient Frontier (Max. Turnover 20%)
Columns 1 through 7

    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000    20.0000

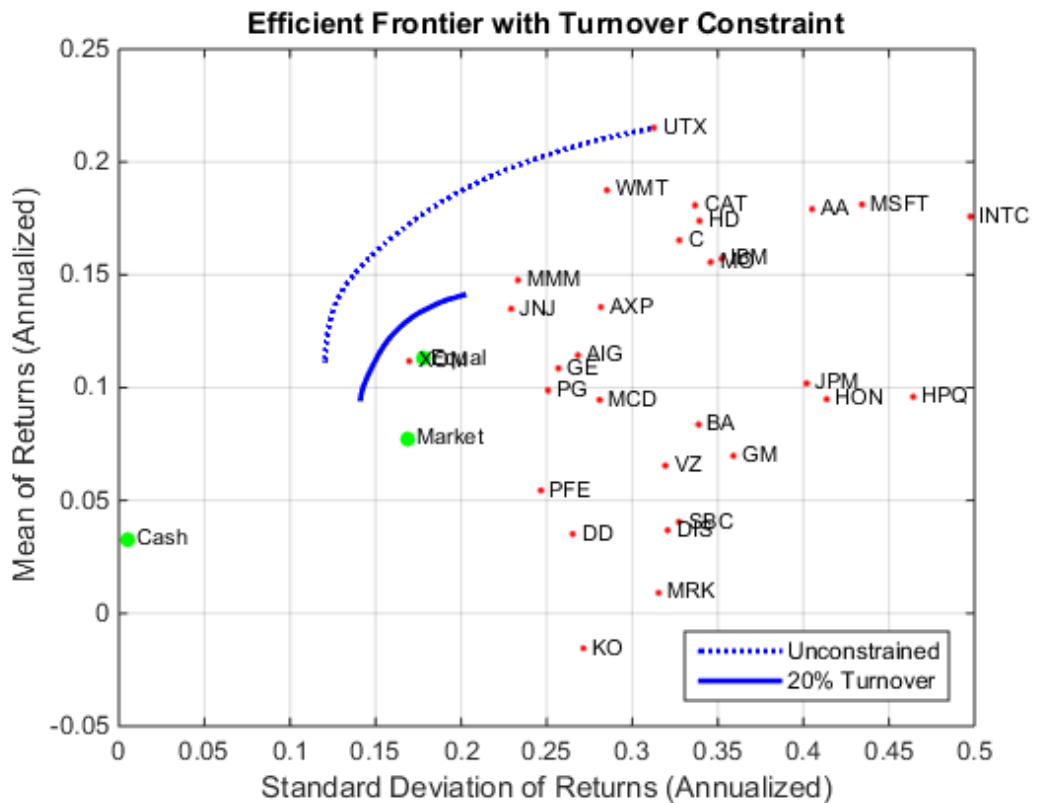
```

Columns 8 through 14

20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000

Columns 15 through 20

20.0000 20.0000 20.0000 20.0000 20.0000 20.0000



Maximize the Sharpe Ratio

The Sharpe ratio (Sharpe 1966) is a measure of return-to-risk that plays an important role in portfolio analysis. Specifically, a portfolio that maximizes the Sharpe ratio is

also the tangency portfolio on the efficient frontier from the mutual fund theorem. The maximum Sharpe ratio portfolio is located on the efficient frontier with the method `estimateMaxSharpeRatio` and the dataset object is used to list the assets in this portfolio.

```
p = setInitPort(p, 0);

swgt = estimateMaxSharpeRatio(p);
[srsk, sret] = estimatePortMoments(p, swgt);

% Plot efficient frontier with portfolio that attains maximum Sharpe ratio

clf;
portfolioexamples_plot('Efficient Frontier with Maximum Sharpe Ratio Portfolio', ...
    {'line', prsk, pret}, ...
    {'scatter', srsk, sret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});

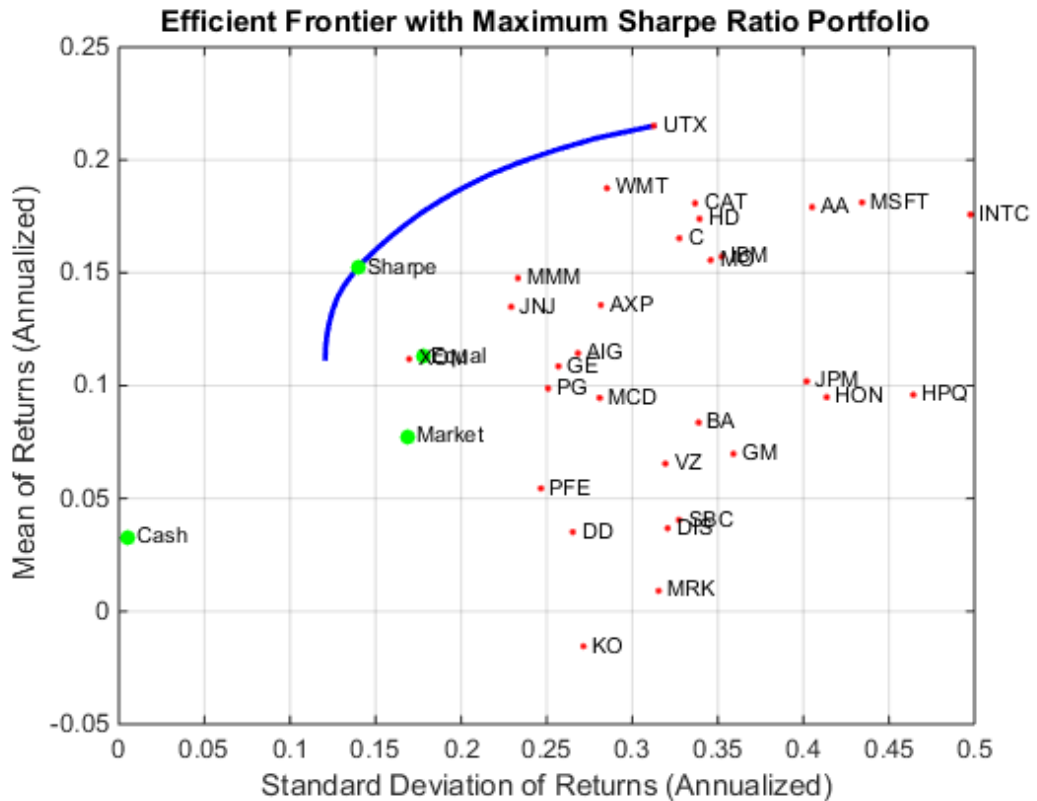
% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio

Blotter = dataset({100*swgt(swgt > 0)}, 'Weight', 'obsnames', AssetList(swgt > 0));

fprintf('Portfolio with Maximum Sharpe Ratio\n');
disp(Blotter);
```

Portfolio with Maximum Sharpe Ratio

	Weight
INTC	2.6638
JNJ	9.0044
MMM	15.502
MO	13.996
MSFT	4.4777
PG	7.4588
UTX	6.0056
WMT	22.051
XOM	18.841



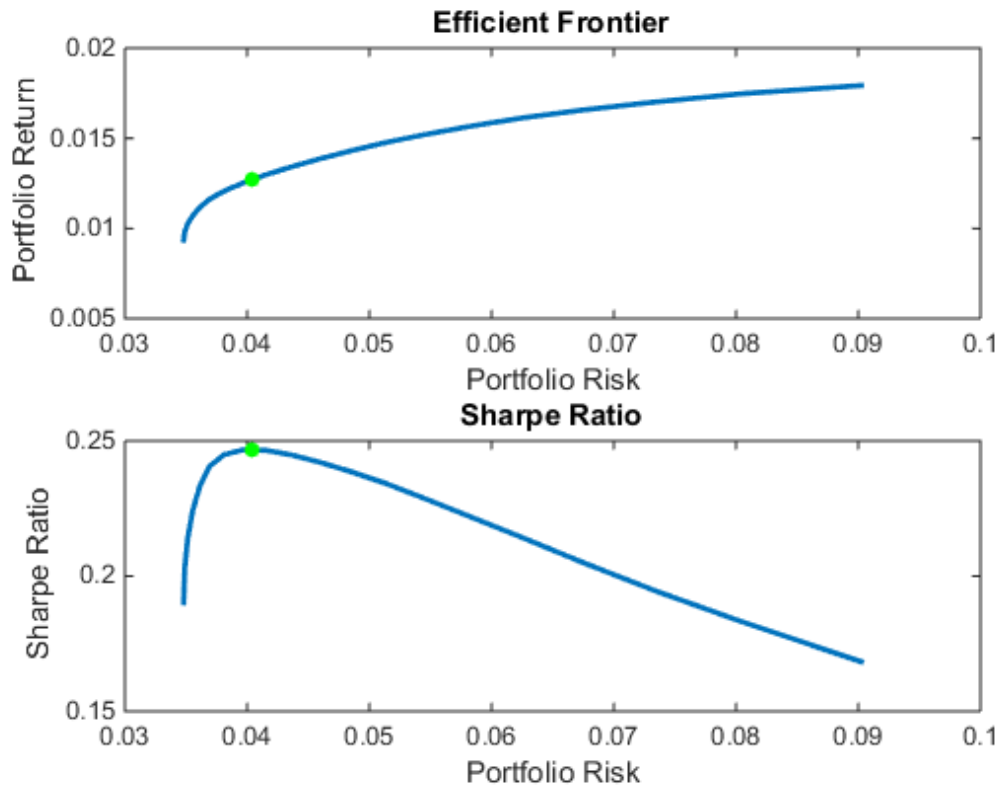
Confirm that Maximum Sharpe Ratio is a Maximum

The following plot demonstrates that this portfolio (which is located at the dot on the plots) indeed maximizes the Sharpe ratio among all portfolios on the efficient frontier.

```
psratio = (pret - p.RiskFreeRate) ./ prsk;
ssratio = (sret - p.RiskFreeRate) / srsk;
```

```
clf;
subplot(2,1,1);
plot(prsk, pret, 'LineWidth', 2);
hold on
scatter(srsk, sret, 'g', 'filled');
title('\bfEfficient Frontier');
```

```
xlabel('Portfolio Risk');  
ylabel('Portfolio Return');  
hold off  
  
subplot(2,1,2);  
plot(prsk, psratio, 'LineWidth', 2);  
hold on  
scatter(srsk, ssratio, 'g', 'filled');  
title('\bfSharpe Ratio');  
xlabel('Portfolio Risk');  
ylabel('Sharpe Ratio');  
hold off
```



Illustrate that Sharpe is the Tangent Portfolio

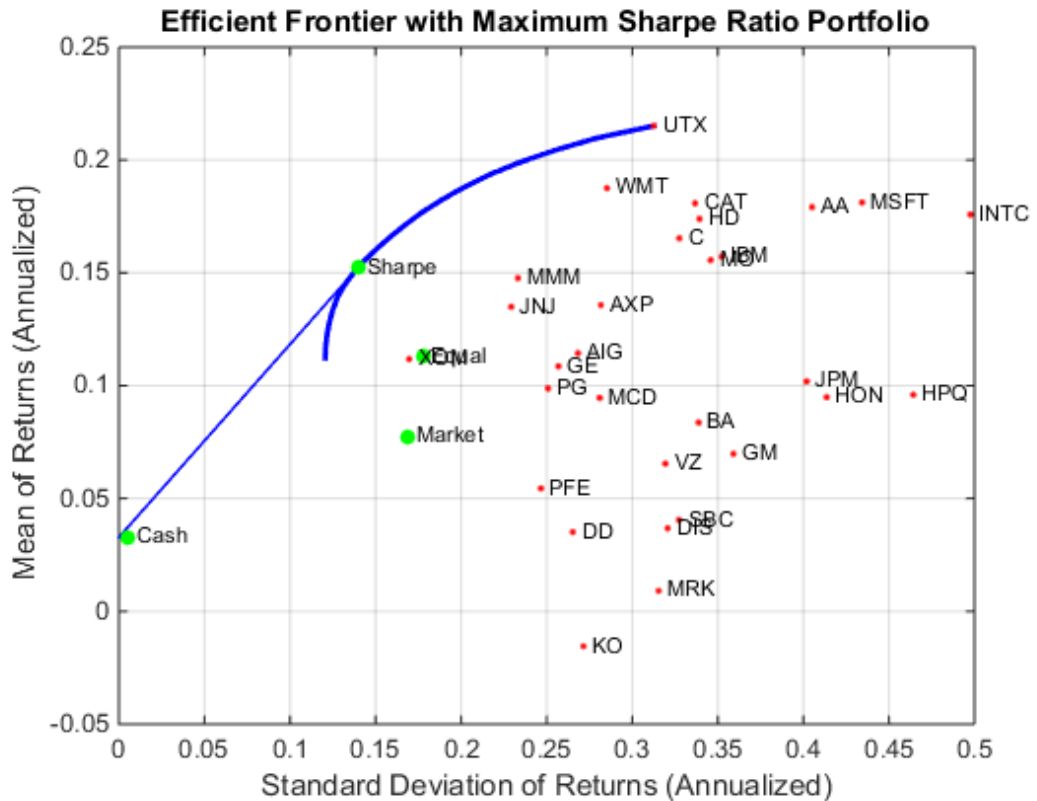
The next plot demonstrates that the portfolio that maximizes the Sharpe ratio is also a tangency portfolio (in this case, the budget constraint is opened up to permit between 0% and 100% in cash).

```
q = setBudget(p, 0, 1);

qwt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwt);

% Plot that shows Sharpe ratio portfolio is the tangency portfolio

clf;
portfolioexamples_plot('Efficient Frontier with Maximum Sharpe Ratio Portfolio', ...
    {'line', prsk, pret}, ...
    {'line', qrsk, qret, [], [], 1}, ...
    {'scatter', srsk, sret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetsList, '.r'});
```



Dollar-Neutral Hedge-Fund Structure

To illustrate how to use the portfolio optimization tools in hedge fund management, two popular strategies with dollar-neutral and 130-30 portfolios are examined. The dollar-neutral strategy invests equally in long and short positions such that the net portfolio position is 0. Such a portfolio is said to be dollar-neutral.

To set up a dollar-neutral portfolio, start with the "standard" portfolio problem and set the maximum exposure in long and short positions in the variable **Exposure**. The bounds for individual asset weights are plus or minus **Exposure**. Since the net position must be dollar-neutral, the budget constraint is 0 and the initial portfolio must be 0. Finally, the one-way turnover constraints provide the necessary long and short restrictions to prevent "double-counting" of long and short positions. The blotter shows the portfolio weights for the dollar-neutral portfolio that maximizes the Sharpe ratio. The

long and short positions are obtained from the buy and sell trades relative to the initial portfolio.

```

Exposure = 1;

q = setBounds(p, -Exposure, Exposure);
q = setBudget(q, 0, 0);
q = setOneWayTurnover(q, Exposure, Exposure, 0);

[qwgt, qlong, qshort] = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

[qswgt, qslong, qsshort] = estimateMaxSharpeRatio(q);
[qsrsk, qsret] = estimatePortMoments(q, qswgt);

% Plot efficient frontier for a dollar-neutral fund structure with tangency portfolio
clf;
portfolioexamples_plot('Efficient Frontier with Dollar-Neutral Portfolio', ...
    {'line', prsk, pret, {'Standard'}, 'b'}, ...
    {'line', qrsk, qret, {'Dollar-Neutral'}, 'b'}, ...
    {'scatter', qsrsk, qsret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});

% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio
Blotter = dataset({100*qswgt(abs(qswgt) > 1.0e-4), 'Weight'}, ...
    {100*qslong(abs(qswgt) > 1.0e-4), 'Long'}, ...
    {100*qsshort(abs(qswgt) > 1.0e-4), 'Short'}, ...
    'obsnames', AssetList(abs(qswgt) > 1.0e-4));

fprintf('Dollar-Neutral Portfolio with Maximum Sharpe Ratio\n');
disp(Blotter);

fprintf('Confirm Dollar-Neutral Portfolio\n');
fprintf(' (Net, Long, Short)\n');
disp([ sum(Blotter.Weight), sum(Blotter.Long), sum(Blotter.Short) ]);

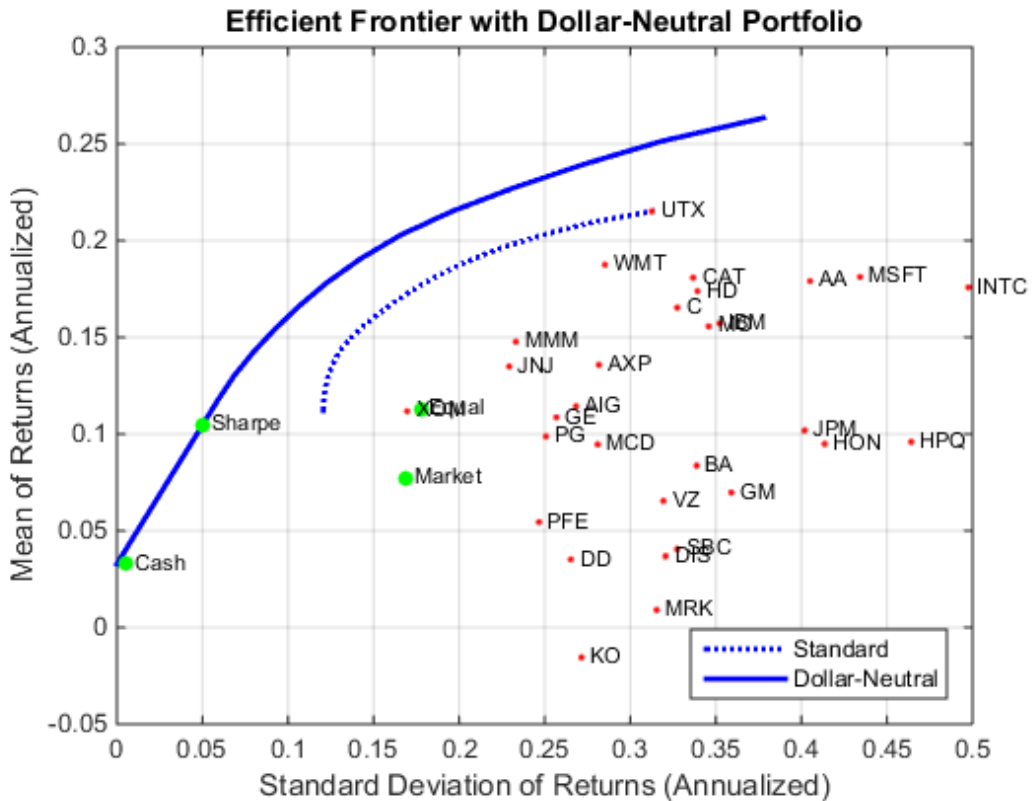
Dollar-Neutral Portfolio with Maximum Sharpe Ratio

```

	Weight	Long	Short
AA	0.54635	0.54635	0
AIG	3.2638	3.2638	0
AXP	0.99646	0.99646	0
BA	-3.7532	0	3.7532

C	15.036	15.036	0
CAT	4.0011	4.0011	0
DD	-19.396	0	19.396
DIS	-5.1796	0	5.1796
GE	-3.8848	0	3.8848
GM	-3.9958	0	3.9958
HD	1.1822	1.1822	0
HON	-1.5408	0	1.5408
HPQ	0.1064	0.1064	0
IBM	-8.6533	0	8.6533
INTC	1.8999	1.8999	0
JNJ	1.4707	1.4707	0
JPM	-2.7136	0	2.7136
KO	-15.253	0	15.253
MCD	4.1986	4.1986	0
MMM	8.1604	8.1604	0
MO	4.3871	4.3871	0
MRK	4.0236	4.0236	0
MSFT	4.3777	4.3777	0
PFE	-9.7673	0	9.7673
PG	1.771	1.771	0
SBC	-5.6426	0	5.6426
UTX	6.1694	6.1694	0
VZ	-2.6179	0	2.6179
WMT	0.91105	0.91105	0
XOM	19.896	19.896	0

Confirm Dollar-Neutral Portfolio
(Net, Long, Short)
-0.0000 82.3985 82.3985



130/30 Fund Structure

Finally, the turnover constraints can be used to set up a 130-30 portfolio structure, which is a structure with a net long position but permits leverage with long and short positions up to a maximum amount of leverage. In the case of a 130-30 portfolio, the leverage is 30%.

To set up a 130-30 portfolio, start with the "standard" portfolio problem and set the maximum value for leverage in the variable `Leverage`. The bounds for individual asset weights range between $-Leverage$ and $1 + Leverage$. Since the net position must be long, the budget constraint is 1 and, once again, the initial portfolio is 0. Finally, the one-way turnover constraints provide the necessary long and short restrictions to prevent "double-counting" of long and short positions. The blotter shows the portfolio weights for

the 130-30 portfolio that maximizes the Sharpe ratio. The long and short positions are obtained from the buy and sell trades relative to the initial portfolio.

```
Leverage = 0.3;
```

```
q = setBounds(p, -Leverage, 1 + Leverage);
q = setBudget(q, 1, 1);
q = setOneWayTurnover(q, 1 + Leverage, Leverage);
```

```
[qwgt, qbuy, qsell] = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);
```

```
[qswgt, qslong, qsshort] = estimateMaxSharpeRatio(q);
[qsrsk, qsret] = estimatePortMoments(q, qswgt);
```

```
% Plot efficient frontier for a 130-30 fund structure with tangency portfolio
```

```
clf;
portfolioexamples_plot(sprintf('Efficient Frontier with %g-%g Portfolio', ...
    100*(1 + Leverage), 100*Leverage), ...
    {'line', prsk, pret, {'Standard'}, 'b:'}, ...
    {'line', qrsk, qret, {'130-30'}, 'b'}, ...
    {'scatter', qsrsk, qsret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```

```
% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio
```

```
Blotter = dataset({100*qswgt(abs(qswgt) > 1.0e-4), 'Weight'}, ...
    {100*qslong(abs(qswgt) > 1.0e-4), 'Long'}, ...
    {100*qsshort(abs(qswgt) > 1.0e-4), 'Short'}, ...
    'obsnames', AssetList(abs(qswgt) > 1.0e-4));
```

```
fprintf('%g-%g Portfolio with Maximum Sharpe Ratio\n', 100*(1 + Leverage), 100*Leverage)
disp(Blotter);
```

```
fprintf('Confirm %g-%g Portfolio\n', 100*(1 + Leverage), 100*Leverage);
fprintf(' (Net, Long, Short)\n');
disp([ sum(Blotter.Weight), sum(Blotter.Long), sum(Blotter.Short) ]);
```

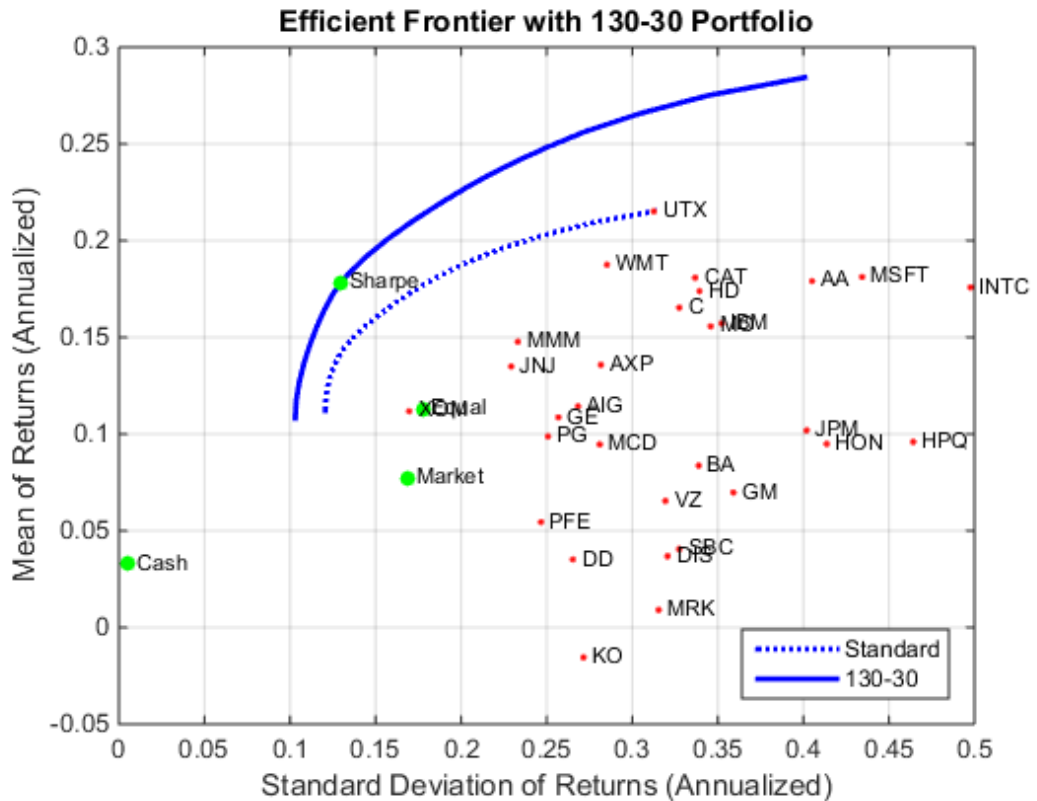
```
130-30 Portfolio with Maximum Sharpe Ratio
```

	Weight	Long	Short
DD	-9.5565	0	9.5565
HON	-6.0245	0	6.0245
INTC	4.0335	4.0335	0

4 Mean-Variance Portfolio Optimization Tools

JNJ	7.1234	7.1234	0
JPM	-0.44583	0	0.44583
KO	-13.646	0	13.646
MMM	20.908	20.908	0
MO	14.433	14.433	0
MSFT	4.5592	4.5592	0
PG	17.243	17.243	0
SBC	-0.32712	0	0.32712
UTX	5.3584	5.3584	0
WMT	21.018	21.018	0
XOM	35.323	35.323	0

Confirm 130-30 Portfolio
(Net, Long, Short)
100.0000 130.0000 30.0000



References

- 1 R. C. Grinold and R. N. Kahn (2000), *Active Portfolio Management*, 2nd ed.
- 2 H. M. Markowitz (1952), "Portfolio Selection," *Journal of Finance*, Vol. 1, No. 1, pp. 77-91.
- 3 J. Lintner (1965), "The Valuation of Risk Assets and the Selection of Risky Investments in Stock Portfolios and Capital Budgets," *Review of Economics and Statistics*, Vol. 47, No. 1, pp. 13-37.
- 4 H. M. Markowitz (1959), *Portfolio Selection: Efficient Diversification of Investments*, John Wiley & Sons, Inc.
- 5 W. F. Sharpe (1966), "Mutual Fund Performance," *Journal of Business*, Vol. 39, No. 1, Part 2, pp. 119-138.

- 6** J. Tobin (1958), "Liquidity Preference as Behavior Towards Risk," *Review of Economic Studies*, Vol. 25, No.1, pp. 65-86.
- 7** J. L. Treynor and F. Black (1973), "How to Use Security Analysis to Improve Portfolio Selection," *Journal of Business*, Vol. 46, No. 1, pp. 68-86.

CVaR Portfolio Optimization Tools

- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow” on page 5-16
- “PortfolioCVaR Object” on page 5-17
- “Constructing the PortfolioCVaR Object” on page 5-26
- “Common Operations on the PortfolioCVaR Object” on page 5-34
- “Asset Returns and Scenarios” on page 5-41
- “Working with CVaR Portfolio Constraints” on page 5-56
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios” on page 5-83
- “Estimate Efficient Frontiers” on page 5-95
- “Postprocessing Results” on page 5-104

Portfolio Optimization Theory

In this section...

“Portfolio Optimization Problems” on page 5-2

“Portfolio Problem Specification” on page 5-2

“Return Proxy” on page 5-3

“Risk Proxy” on page 5-5

“Portfolio Set for Portfolio Optimization” on page 5-7

“Default Portfolio Problem” on page 5-14

Portfolio Optimization Problems

Portfolio optimization problems involve identifying portfolios that satisfy three criteria:

- Minimize a proxy for risk.
- Match or exceed a proxy for return.
- Satisfy basic feasibility requirements.

Portfolios are points from a feasible set of assets that constitute an asset universe. A portfolio specifies either holdings or weights in each individual asset in the asset universe. The convention is to specify portfolios in terms of weights, although the portfolio optimization tools work with holdings as well.

The set of feasible portfolios is necessarily a nonempty, closed, and bounded set. The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices. The proxy for return is a function that characterizes either the gross or net benefits associated with portfolio choices. The terms “risk” and “risk proxy” and “return” and “return proxy” are interchangeable. The fundamental insight of Markowitz (see “Portfolio Optimization” on page A-10) is that the goal of the portfolio choice problem is to seek minimum risk for a given level of return and to seek maximum return for a given level of risk. Portfolios satisfying these criteria are efficient portfolios and the graph of the risks and returns of these portfolios forms a curve called the efficient frontier.

Portfolio Problem Specification

To specify a portfolio optimization problem, you need the following:

- Proxy for portfolio return (μ)
- Proxy for portfolio risk (Σ)
- Set of feasible portfolios (X), called a portfolio set

Financial Toolbox has three objects to solve specific types of portfolio optimization problems:

- The Portfolio object (`Portfolio`) supports mean-variance portfolio optimization (see Markowitz [46], [47] at “Portfolio Optimization” on page A-10). This object has either gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set.
- The PortfolioCVaR object (`PortfolioCVaR`) implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-10), which is generally referred to as CVaR portfolio optimization. CVaR portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses conditional value-at-risk of portfolio returns as the risk proxy.
- The PortfolioMAD object (`PortfolioMAD`) implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10), which is generally referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

The proxy for portfolio return is a function $\mu : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the rewards associated with portfolio choices. In most cases, the proxy for portfolio return has two general forms, gross and net portfolio returns. Both portfolio return forms separate the risk-free rate r_0 so that the portfolio $x \in X$ contains only risky assets.

Regardless of the underlying distribution of asset returns, a collection of S asset returns y_1, \dots, y_S has a mean of asset returns

$$m = \frac{1}{S} \sum_{s=1}^S y_s,$$

and (sample) covariance of asset returns

$$C = \frac{1}{S-1} \sum_{s=1}^S (y_s - m)(y_s - m)^T.$$

These moments (or alternative estimators that characterize these moments) are used directly in mean-variance portfolio optimization to form proxies for portfolio risk and return.

Gross Portfolio Returns

The gross portfolio return for a portfolio $x \in X$ is

$$\mu(x) = r_0 + (m - r_0 \mathbf{1})^T x,$$

where:

r_0 is the risk-free rate (scalar).

m is the mean of asset returns (n vector).

If the portfolio weights sum to 1, the risk-free rate is irrelevant. The properties in the Portfolio object to specify gross portfolio returns are:

- `RiskFreeRate` for r_0
- `AssetMean` for m

Net Portfolio Returns

The net portfolio return for a portfolio $x \in X$ is

$$\mu(x) = r_0 + (m - r_0 \mathbf{1})^T x - b^T \max\{0, x - x_0\} - s^T \max\{0, x_0 - x\},$$

where:

r_0 is the risk-free rate (scalar).

m is the mean of asset returns (n vector).

b is the proportional cost to purchase assets (n vector).

s is the proportional cost to sell assets (n vector).

You can incorporate fixed transaction costs in this model also. Though in this case, it is necessary to incorporate prices into such costs. The properties in the Portfolio object to specify net portfolio returns are:

- `RiskFreeRate` for r_0
- `AssetMean` for m
- `InitPort` for x_0
- `BuyCost` for b
- `SellCost` for s

Risk Proxy

The proxy for portfolio risk is a function $\Sigma : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the risks associated with portfolio choices.

Variance

The variance of portfolio returns for a portfolio $x \in X$ is

$$\Sigma(x) = x^T C x$$

where C is the covariance of asset returns (n -by- n positive-semidefinite matrix).

The property in the Portfolio object to specify the variance of portfolio returns is `AssetCovar` for C .

Although the risk proxy in mean-variance portfolio optimization is the variance of portfolio returns, the square root, which is the standard deviation of portfolio returns, is often reported and displayed. Moreover, this quantity is often called the “risk” of the portfolio. For details, see Markowitz (“Portfolio Optimization” on page A-10).

Conditional Value-at-Risk

The conditional value-at-risk for a portfolio $x \in X$, which is also known as expected shortfall, is defined as

$$CVaR_{\alpha}(x) = \frac{1}{1-\alpha} \int_{f(x,y) \geq VaR_{\alpha}(x)} f(x,y)p(y)dy,$$

where:

α is the probability level such that $0 < \alpha < 1$.

$f(x,y)$ is the loss function for a portfolio x and asset return y .

$p(y)$ is the probability density function for asset return y .

VaR_{α} is the value-at-risk of portfolio x at probability level α .

The value-at-risk is defined as

$$VaR_{\alpha}(x) = \min\{\gamma : \Pr[f(x,Y) \leq \gamma] \geq \alpha\}.$$

An alternative formulation for CVaR has the form:

$$CVaR_{\alpha}(x) = VaR_{\alpha}(x) + \frac{1}{1-\alpha} \int_{R^n} \max\{0, (f(x,y) - VaR_{\alpha}(x))\} p(y) dy$$

The choice for the probability level α is typically 0.9 or 0.95. Choosing α implies that the value-at-risk $VaR_{\alpha}(x)$ for portfolio x is the portfolio return such that the probability of portfolio returns falling below this level is $(1 - \alpha)$. Given $VaR_{\alpha}(x)$ for a portfolio x , the conditional value-at-risk of the portfolio is the expected loss of portfolio returns above the value-at-risk return.

Note: Value-at-risk is a positive value for losses so that the probability level α indicates the probability that portfolio returns are below the negative of the value-at-risk.

The risk proxy for CVaR portfolio optimization is $CVaR_{\alpha}(x)$ for a given portfolio $x \in X$ and $\alpha \in (0,1)$. Note that the value-at-risk, or VaR, for a given probability level is estimated whenever CVaR is estimated.

In addition, keep in mind that VaR and CVaR are sample estimators for VaR and CVaR based on the given scenarios. Better scenario samples yield more reliable estimates of VaR and CVaR.

For more information, see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-10.

Mean Absolute-Deviation

The mean-absolute deviation (MAD) for a portfolio $x \in X$ is defined as

$$\sum(x) = \frac{1}{S} \sum_{s=1}^S |(y_s - m)^T x|$$

where:

y_s are asset returns with scenarios $s = 1, \dots, S$ (S collection of n vectors).

$f(x, y)$ is the loss function for a portfolio x and asset return y .

m is the mean of asset returns (n vector).

such that

$$m = \frac{1}{S} \sum_{s=1}^S y_s$$

For more information, see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10.

Portfolio Set for Portfolio Optimization

The final element for a complete specification of a portfolio optimization problem is the set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset \mathcal{R}^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or “default” portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to 1 (using the budget constraint). The most general portfolio set handled by the portfolio optimization tools can have any of these constraints:

- Linear inequality constraints
- Linear equality constraints
- Bound constraints
- Budget constraints
- Group constraints
- Group ratio constraints
- Average turnover constraints
- One-way turnover constraints

Linear Inequality Constraints

Linear inequality constraints are general linear constraints that model relationships among portfolio weights that satisfy a system of inequalities. Linear inequality constraints take the form

$$A_I x \leq b_I$$

where:

x is the portfolio (n vector).

A_I is the linear inequality constraint matrix (n_I -by- n matrix).

b_I is the linear inequality constraint vector (n_I vector).

n is the number of assets in the universe and n_I is the number of constraints.

Portfolio object properties to specify linear inequality constraints are:

- `AInequality` for A_I
- `bInequality` for b_I
- `NumAssets` for n

The default is to ignore these constraints.

Linear Equality Constraints

Linear equality constraints are general linear constraints that model relationships among portfolio weights that satisfy a system of equalities. Linear equality constraints take the form

$$A_E x = b_E$$

where:

x is the portfolio (n vector).

A_E is the linear equality constraint matrix (n_E -by- n matrix).

b_E is the linear equality constraint vector (n_E vector).

n is the number of assets in the universe and n_E is the number of constraints.

Portfolio object properties to specify linear equality constraints are:

- `AEquality` for A_E
- `bEquality` for b_E
- `NumAssets` for n

The default is to ignore these constraints.

Bound Constraints

Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Since every portfolio set must be bounded, it is often a good practice, albeit not necessary, to set explicit bounds for the portfolio problem. To obtain explicit bounds for a given portfolio set, use the method `estimateBounds`. Bound constraints take the form

$$l_B \leq x \leq u_B$$

where:

x is the portfolio (n vector).

l_B is the lower-bound constraint (n vector).

u_B is the upper-bound constraint (n vector).

n is the number of assets in the universe.

Portfolio object properties to specify bound constraints are:

- `LowerBound` for l_B
- `UpperBound` for u_B
- `NumAssets` for n

The default is to ignore these constraints.

Note that the default portfolio optimization problem (see “Default Portfolio Problem” on page 5-14) has $l_B = 0$ with u_B set implicitly through a budget constraint.

Budget Constraints

Budget constraints are specialized linear constraints that confine the sum of portfolio weights to fall either above or below specific bounds. The constraints take the form

$$l_S \leq \mathbf{1}^T x \leq u_S$$

where:

x is the portfolio (n vector).

$\mathbf{1}$ is the vector of ones (n vector).

l_S is the lower-bound budget constraint (scalar).

u_S is the upper-bound budget constraint (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify budget constraints are:

- `LowerBudget` for l_S
- `UpperBudget` for u_S

- `NumAssets` for n

The default is to ignore this constraint.

The default portfolio optimization problem (see “Default Portfolio Problem” on page 5-14) has $l_S = u_S = 1$, which means that the portfolio weights sum to 1. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0 to 100% invested in cash. If cash is to be a portfolio choice, set `RiskFreeRate` (r_0) to a suitable value (see “Portfolio Problem Specification” on page 5-2 and “Working with a Riskless Asset” on page 5-51).

Group Constraints

Group constraints are specialized linear constraints that enforce “membership” among groups of assets. The constraints take the form

$$l_G \leq Gx \leq u_G$$

where:

x is the portfolio (n vector).

l_G is the lower-bound group constraint (n_G vector).

u_G is the upper-bound group constraint (n_G vector).

G is the matrix of group membership indexes (n_G -by- n matrix).

Each row of G identifies which assets belong to a group associated with that row. Each row contains either 0s or 1s with 1 indicating that an asset is part of the group or 0 indicating that the asset is not part of the group.

Portfolio object properties to specify group constraints are:

- `GroupMatrix` for G
- `LowerGroup` for l_G
- `UpperGroup` for u_G
- `NumAssets` for n

The default is to ignore these constraints.

Group Ratio Constraints

Group ratio constraints are specialized linear constraints that enforce relationships among groups of assets. The constraints take the form

$$l_{Ri} \cdot (G_B x)_i \leq (G_A x)_i \leq u_{Ri} \cdot (G_B x)_i$$

for $i = 1, \dots, n_R$ where:

x is the portfolio (n vector).

l_R is the vector of lower-bound group ratio constraints (n_R vector).

u_R is the vector matrix of upper-bound group ratio constraints (n_R vector).

G_A is the matrix of base group membership indexes (n_R -by- n matrix).

G_B is the matrix of comparison group membership indexes (n_R -by- n matrix).

n is the number of assets in the universe and n_R is the number of constraints.

Each row of G_A and G_B identifies which assets belong to a base and comparison group associated with that row.

Each row contains either 0s or 1s with 1 indicating that an asset is part of the group or 0 indicating that the asset is not part of the group.

Portfolio object properties to specify group ratio constraints are:

- `GroupA` for G_A
- `GroupB` for G_B
- `LowerRatio` for l_R
- `UpperRatio` for u_R
- `NumAssets` for n

The default is to ignore these constraints.

Average Turnover Constraints

Turnover constraint is a linear absolute value constraint that ensures estimated optimal portfolios differ from an initial portfolio by no more than a specified amount. Although

portfolio turnover is defined in many ways, the turnover constraints implemented in Financial Toolbox computes portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

$$\frac{1}{2} \mathbf{1}^T |x - x_0| \leq \tau$$

where:

x is the portfolio (n vector).

$\mathbf{1}$ is the vector of ones (n vector).

x_0 is the initial portfolio (n vector).

τ is the upper bound for turnover (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

- `Turnover` for τ
- `InitPort` for x_0
- `NumAssets` for n

The default is to ignore this constraint.

One-Way Turnover Constraints

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the forms

$$\mathbf{1}^T \times \max\{0, x - x_0\} \leq \tau_B$$

$$\mathbf{1}^T \times \max\{0, x_0 - x\} \leq \tau_S$$

where:

x is the portfolio (n vector)

$\mathbf{1}$ is the vector of ones (n vector).

x_0 is the Initial portfolio (n vector).

τ_B is the upper bound for turnover constraint on purchases (scalar).

τ_S is the upper bound for turnover constraint on sales (scalar).

To specify one-way turnover constraints, use the following properties in the Portfolio or PortfolioCVaR object:

- `BuyTurnover` for τ_B
- `SellTurnover` for τ_S
- `InitPort` for x_0

The default is to ignore this constraint.

Note: The average turnover constraint (see “Average Turnover Constraints” on page 5-12) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

Default Portfolio Problem

The default portfolio optimization problem has a risk and return proxy associated with a given problem, and a portfolio set that specifies portfolio weights to be nonnegative and to sum to 1. The lower bound combined with the budget constraint is sufficient to ensure that the portfolio set is nonempty, closed, and bounded. The default portfolio optimization problem characterizes a long-only investor who is fully invested in a collection of assets.

- For mean-variance portfolio optimization, it is sufficient to set up the default problem. After setting up the problem, data in the form of a mean and covariance of asset returns are then used to solve portfolio optimization problems.
- For conditional value-at-risk portfolio optimization, the default problem requires the additional specification of a probability level that must be set explicitly. Generally, “typical” values for this level are 0.90 or 0.95. After setting up the problem, data in the form of scenarios of asset returns are then used to solve portfolio optimization problems.

- For MAD portfolio optimization, it is sufficient to set up the default problem. After setting up the problem, data in the form of scenarios of asset returns are then used to solve portfolio optimization problems.

See Also

AbstractPortfolio | PortfolioCVaR

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56

More About

- “PortfolioCVaR Object” on page 5-17
- “PortfolioCVaR Object Workflow”

PortfolioCVaR Object Workflow

The PortfolioCVaR object workflow for creating and modeling a mean-variance portfolio is:

1 Create a CVaR Portfolio.

Create a `PortfolioCVaR` object for conditional value-at-risk (CVaR) portfolio optimization. For more information, see “Constructing the PortfolioCVaR Object”.

2 Define asset returns and scenarios.

Evaluate scenarios for portfolio asset returns, including assets with missing data and financial time series data. For more information, see “Asset Returns and Scenarios”.

3 Specify the CVaR Portfolio Constraints.

Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, and turnover constraints. For more information, see “Working with CVaR Portfolio Constraints”.

4 Validate the CVaR Portfolio.

Identify errors for the portfolio specification. For more information, see “Validate the CVaR Portfolio Problem”.

5 Estimate the efficient portfolios and frontiers.

Analyze the efficient portfolios and efficient frontiers for a CVaR portfolio. For more information, see “Estimate Efficient Portfolios” and “Estimate Efficient Frontiers”.

6 Postprocess the results.

Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see “Postprocessing Results”.

PortfolioCVaR Object

In this section...

“PortfolioCVaR Object Properties and Methods” on page 5-17

“Working with PortfolioCVaR Objects” on page 5-21

“Setting and Getting Properties” on page 5-22

“Displaying PortfolioCVaR Objects” on page 5-23

“Saving and Loading PortfolioCVaR Objects” on page 5-23

“Estimating Efficient Portfolios and Frontiers” on page 5-23

“Arrays of PortfolioCVaR Objects” on page 5-23

“Subclassing PortfolioCVaR Objects” on page 5-24

“Conventions for Representation of Data” on page 5-24

PortfolioCVaR Object Properties and Methods

The PortfolioCVaR object implements conditional value-at-risk (CVaR) portfolio optimization and is derived from the abstract class `AbstractPortfolio`. Every property and method of the PortfolioCVaR object is public, although some properties and methods are hidden. See PortfolioCVaR Object Properties and PortfolioCVaR Object Methods for the properties and methods of a PortfolioCVaR object. The PortfolioCVaR object is a value object where every instance of the object is a distinct version of the object. Since the PortfolioCVaR object is also a MATLAB object, it inherits the default methods associated with MATLAB objects.

PortfolioCVaR Object Properties

Property	Description	Characteristics
<code>AEquality</code>	Matrix for equality constraints	Matrix
<code>AInequality</code>	Matrix for inequality constraints	Matrix
<code>AssetList</code>	List of asset names or identifiers	Vector cell array of strings
<code>bEquality</code>	Vector for equality constraints	Vector
<code>bInequality</code>	Vector for inequality constraints	Vector
<code>BuyCost</code>	Cost to purchase assets	Vector
<code>BuyTurnover</code>	Turnover constraint on purchases	Scalar

Property	Description	Characteristics
GroupA	Base group ratio constraint matrix	Boolean matrix or matrix
GroupB	Comparison group ratio constraint matrix	Boolean matrix or matrix
GroupMatrix	Group membership matrix	Boolean matrix or matrix
InitPort	Initial or current portfolio	Vector
LowerBound	Lower-bound constraint	Vector
LowerBudget	Lower-budget constraint	Scalar
LowerGroup	Lower-group constraint	Vector
LowerRatio	Lower-group ratio constraint ratio	Vector
Name	Name for instance of PortfolioCVaR object	String
NumAssets	Number of assets in universe	Scalar positive integer
NumScenarios	Number of scenarios (read only)	Scalar integer
ProbabilityLevel	Probability level which is 1 minus the probability of losses greater than the value-at-risk	Scalar
RiskFreeRate	Period return of riskless asset	Scalar
SellCost	Cost to sell assets	Vector
SellTurnover	Turnover constraint on sales	Scalar
Turnover	Upper-bound portfolio turnover	Scalar
UpperBound	Upper-bound constraint	Vector
UpperBudget	Upper-budget constraint	Scalar
UpperGroup	Upper-group constraint	Vector
UpperRatio	Upper-group ratio constraint ratio	Vector

PortfolioCVaR Object Methods

Method	Description
addEquality	Add equality constraints for portfolio weights to existing constraints.

Method	Description
<code>addGroupRatio</code>	Add group ratio constraints for portfolio weights to existing constraints.
<code>addGroups</code>	Add group constraints for portfolio weights to existing constraints.
<code>addInequality</code>	Add inequality constraints for portfolio weights to existing constraints.
<code>checkFeasibility</code>	Determine if portfolios are members of the set of feasible portfolios.
<code>estimateBounds</code>	Determine if the set of feasible portfolios is nonempty and bounded.
<code>estimateFrontier</code>	Estimate portfolios on the entire efficient frontier.
<code>estimateFrontierByReturn</code>	Estimate portfolios on the efficient frontier with targeted returns or return proxies.
<code>estimateFrontierByRisk</code>	Estimate portfolios on the efficient frontier with targeted risks or risk proxies.
<code>estimateFrontierLimits</code>	Estimate portfolios at the extreme ends of the efficient frontier (minimum risk and maximum return).
<code>estimatePortReturn</code>	Estimate the return or return proxy for specified portfolios.
<code>estimatePortRisk</code>	Estimate the risk or risk proxy for specified portfolios.
<code>estimatePortStd</code>	Estimate standard deviation of portfolio returns.
<code>estimatePortVaR</code>	Estimate value-at-risk for portfolio.
<code>estimateScenarioMoments</code>	Estimate mean and covariance of scenarios.
<code>getBounds</code>	Get lower and upper bounds from object.
<code>simulateNormalScenariosByData</code>	Simulate multivariate normal asset return scenarios from data.

Method	Description
<code>simulateNormalScenariosByMoments</code>	Simulate multivariate normal asset return scenarios from a mean and covariance of asset returns.
<code>getBudget</code>	Get lower and upper budget constraints from object.
<code>getCosts</code>	Get purchase and sales proportional transaction costs from object.
<code>getEquality</code>	Get equality constraint matrix and vector from object.
<code>getGroupRatio</code>	Get base matrix, comparison matrix, and lower and upper bounds for group ratio constraints from object.
<code>getGroups</code>	Get group matrix and lower and upper bounds for group constraints from object.
<code>getInequality</code>	Get inequality constraint matrix and vector from object.
<code>getOneWayTurnover</code>	Get one-way portfolio turnover constraints.
<code>getScenarios</code>	Obtain scenarios from PortfolioCVaR object.
<code>plotFrontier</code>	Plot efficient frontier and optionally obtain risks and returns for portfolios on the efficient frontier.
<code>setAssetList</code>	Set up a list of asset names and symbols to be associated with assets in universe.
<code>setBounds</code>	Set up lower and upper bounds for portfolio weights.
<code>setBudget</code>	Set up lower and upper budget constraints for portfolio weights.
<code>setCosts</code>	Set up purchase and sale proportional transaction costs for assets in universe.
<code>setDefaultConstraints</code>	Set up default constraints for portfolio weights (nonnegative weights that must sum to 1).

Method	Description
setEquality	Set up equality constraints for portfolio weights.
setGroupRatio	Set up group ratio constraints for portfolio weights.
setGroups	Set up group constraints for portfolio weights.
setInequality	Set up inequality constraints for portfolio weights.
setInitPort	Set up initial portfolio weights.
setOneWayTurnover	Set up one-way portfolio turnover constraints.
setOptions	Set up hidden control properties in object (not implemented).
setProbabilityLevel	Set probability level for VaR and CVaR calculations.
setScenarios	Set asset returns scenarios by direct matrix.
setSolver	Set up solver to estimate efficient portfolios.
setTurnover	Set up average turnover constraints for portfolio weights.

Working with PortfolioCVaR Objects

The PortfolioCVaR object and its methods are an interface for conditional value-at-risk portfolio optimization. Consequently, almost everything you do with the PortfolioCVaR object can be done using the methods. The basic workflow is:

- 1 Design your portfolio problem.
- 2 Use the portfolio constructor (`PortfolioCVaR`) to create the PortfolioCVaR object or use the various set methods to set up your portfolio problem.
- 3 Use estimate methods to solve your portfolio problem.

In addition, methods are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a PortfolioCVaR object, you can save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of CVaR portfolio optimization, means that you have either scenarios, data, or moments for asset returns, a probability level,

and a collection of constraints on your portfolios, use the portfolio constructor to set the properties for the PortfolioCVaR object.

The `PortfolioCVaR` constructor lets you create an object from scratch or update an existing object. Since the `PortfolioCVaR` object is a value object, it is easy to create a basic object, then use methods to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see “Constructing the PortfolioCVaR Object” on page 5-26.

Setting and Getting Properties

You can set properties of a `PortfolioCVaR` object using either the constructor (`PortfolioCVaR`) or various set methods.

Note: Although you can also set properties directly, it is not recommended since error-checking is not performed when you set a property directly.

The `PortfolioCVaR` constructor supports setting properties with name-value pair arguments such that each argument name is a property and each value is the value to assign to that property. For example, to set the `LowerBound`, `Budget`, and `ProbabilityLevel` properties in an existing `PortfolioCVaR` object `p`, use the syntax:

```
p = PortfolioCVaR(p, 'LowerBound', 0, 'Budget', 'ProbabilityLevel', 0.95);
```

In addition to the `PortfolioCVaR` constructor, which lets you set individual properties one at a time, groups of properties are set in a `PortfolioCVaR` object with various “set” and “add” methods. For example, to set up an average turnover constraint, use the `setTurnover` method to specify the bound on portfolio turnover and the initial portfolio. To get individual properties from a `PortfolioCVaR` object, obtain properties directly or use an assortment of “get” methods that obtain groups of properties from a `PortfolioCVaR` object. The `PortfolioCVaR` object constructor and set methods have several useful features:

- The constructor and set methods try to determine the dimensions of your problem with either explicit or implicit inputs.
- The constructor and set methods try to resolve ambiguities with default choices.
- The constructor and set methods perform scalar expansion on arrays when possible.
- The methods try to diagnose and warn about problems.

Displaying PortfolioCVaR Objects

The PortfolioCVaR object uses the default display method provided by MATLAB, where `display` and `disp` display a PortfolioCVaR object and its properties with or without the object variable name.

Saving and Loading PortfolioCVaR Objects

Save and load PortfolioCVaR objects using the MATLAB `save` and `load` commands.

Estimating Efficient Portfolios and Frontiers

Estimating efficient portfolios and efficient frontiers is the primary purpose of the CVaR portfolio optimization tools. A collection of “estimate” and “plot” methods provide ways to explore the efficient frontier. The “estimate” methods obtain either efficient portfolios or risk and return proxies to form efficient frontiers. At the portfolio level, a collection of methods estimates efficient portfolios on the efficient frontier with methods to obtain efficient portfolios:

- At the endpoints of the efficient frontier
- That attain targeted values for return proxies
- That attain targeted values for risk proxies
- Along the entire efficient frontier

These methods also provide purchases and sales needed to shift from an initial or current portfolio to each efficient portfolio. At the efficient frontier level, a collection of methods plot the efficient frontier and estimate either risk or return proxies for efficient portfolios on the efficient frontier. You can use the resultant efficient portfolios or risk and return proxies in subsequent analyses.

Arrays of PortfolioCVaR Objects

Although all methods associated with a PortfolioCVaR object are designed to work on a scalar PortfolioCVaR object, the array capabilities of MATLAB enables you to set up and work with arrays of PortfolioCVaR objects. The easiest way to do this is with the `repmat` function. For example, to create a 3-by-2 array of PortfolioCVaR objects:

```
p = repmat(PortfolioCVaR, 3, 2);
```

`disp(p)`

After setting up an array of `PortfolioCVaR` objects, you can work on individual `PortfolioCVaR` objects in the array by indexing. For example:

```
p(i,j) = PortfolioCVaR(p(i,j), ... );
```

This example calls the `PortfolioCVaR` object constructor for the (i,j) element of a matrix of `PortfolioCVaR` objects in the variable `p`.

If you set up an array of `PortfolioCVaR` objects, you can access properties of a particular `PortfolioCVaR` object in the array by indexing so that you can set the lower and upper bounds `lb` and `ub` for the (i,j,k) element of a 3-D array of `PortfolioCVaR` objects with

```
p(i,j,k) = setBounds(p(i,j,k), lb, ub);
```

and, once set, you can access these bounds with

```
[lb, ub] = getBounds(p(i,j,k));
```

`PortfolioCVaR` object methods work on only one `PortfolioCVaR` object at a time.

Subclassing PortfolioCVaR Objects

You can subclass the `PortfolioCVaR` object to override existing methods or to add new properties or methods. To do so, create a derived class from the `PortfolioCVaR` class. This gives you all the properties and methods of the `PortfolioCVaR` class along with any new features that you choose to add to your subclassed object. The `PortfolioCVaR` class is derived from an abstract class called `AbstractPortfolio`. Because of this, you can also create a derived class from `AbstractPortfolio` that implements an entirely different form of portfolio optimization using properties and methods of the `AbstractPortfolio` class.

Conventions for Representation of Data

The CVaR portfolio optimization tools follow these conventions regarding the representation of different quantities associated with portfolio optimization:

- Asset returns or prices for scenarios are in matrix form with samples for a given asset going down the rows and assets going across the columns. In the case of prices, the earliest dates must be at the top of the matrix, with increasing dates going down.
- Portfolios are in vector or matrix form with weights for a given portfolio going down the rows and distinct portfolios going across the columns.
- Constraints on portfolios are formed in such a way that a portfolio is a column vector.

- Portfolio risks and returns are either scalars or column vectors (for multiple portfolio risks and returns).

See Also

AbstractPortfolio | PortfolioCVaR

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56

More About

- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Constructing the PortfolioCVaR Object

In this section...

“Syntax” on page 5-26

“PortfolioCVaR Problem Sufficiency” on page 5-27

“Constructor Examples” on page 5-27

To create a fully specified CVaR portfolio optimization problem, instantiate the PortfolioCVaR object using the CVaR portfolio constructor. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow”.

Syntax

Use the CVaR portfolio constructor `PortfolioCVaR` to create an instance of an object of the `PortfolioCVaR` class. You can use this constructor in several ways. To set up a portfolio optimization problem in a PortfolioCVaR object, the simplest syntax is:

```
p = PortfolioCVaR;
```

This syntax creates a PortfolioCVaR object, `p`, such that all object properties are empty.

The constructor also accepts collections of argument name-value pair arguments for properties and their values. The constructor accepts inputs for public properties (see PortfolioCVaR Object Properties) with the general syntax:

```
p = PortfolioCVaR('property1', value1, 'property2', value2, ... );
```

If a PortfolioCVaR object already exists, the syntax permits the first (and only the first argument) of the CVaR portfolio constructor to be an existing object with subsequent argument name-value pair arguments for properties to be added or modified. For example, given an existing PortfolioCVaR object in `p`, the general syntax is:

```
p = PortfolioCVaR(p, 'property1', value1, 'property2', value2, ... );
```

Input argument names are not case sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 5-31). The `PortfolioCVaR` constructor tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process

to formulate a problem. In addition, a PortfolioCVaR object is a value object so that, given portfolio p , the following code creates two objects, p and q , that are distinct:

```
q = PortfolioCVaR(p, ...)
```

PortfolioCVaR Problem Sufficiency

A CVaR portfolio optimization problem is completely specified with the PortfolioCVaR object if the following three conditions are met:

- You must specify a collection of asset returns or prices known as scenarios such that all scenarios are finite asset returns or prices. These scenarios are meant to be samples from the underlying probability distribution of asset returns. This condition can be satisfied by the `setScenarios` method or with several canned scenario simulation methods.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded. You can satisfy this condition using an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed and several tools, such as the `estimateBounds` method, provide ways to ensure that your problem is properly formulated.
- You must specify a probability level to locate the level of tail loss above which the conditional value-at-risk is to be minimized. This condition can be satisfied by the `setProbabilityLevel` method.

Although the general sufficient conditions for CVaR portfolio optimization go beyond the first three conditions, the PortfolioCVaR object handles all these additional conditions.

Constructor Examples

If you create a PortfolioCVaR object, p , with no input arguments, you can display it using `disp`:

```
p = PortfolioCVaR;
disp(p);
```

PortfolioCVaR with properties:

```
BuyCost: []
SellCost: []
```

```
RiskFreeRate: []
ProbabilityLevel: []
  Turnover: []
  BuyTurnover: []
  SellTurnover: []
NumScenarios: []
  Name: []
  NumAssets: []
  AssetList: []
  InitPort: []
AInequality: []
bInequality: []
  AEquality: []
  bEquality: []
LowerBound: []
UpperBound: []
LowerBudget: []
UpperBudget: []
GroupMatrix: []
  LowerGroup: []
  UpperGroup: []
  GroupA: []
  GroupB: []
LowerRatio: []
UpperRatio: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the CVaR portfolio constructor. The custom set methods offer additional ways to set and modify collections of properties in the PortfolioCVaR object.

Using the Constructor for a Single-Step Setup

You can use the PortfolioCVaR constructor to directly set up a “standard” portfolio optimization problem. Given scenarios of asset returns in the variable AssetScenarios, this problem is completely specified as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

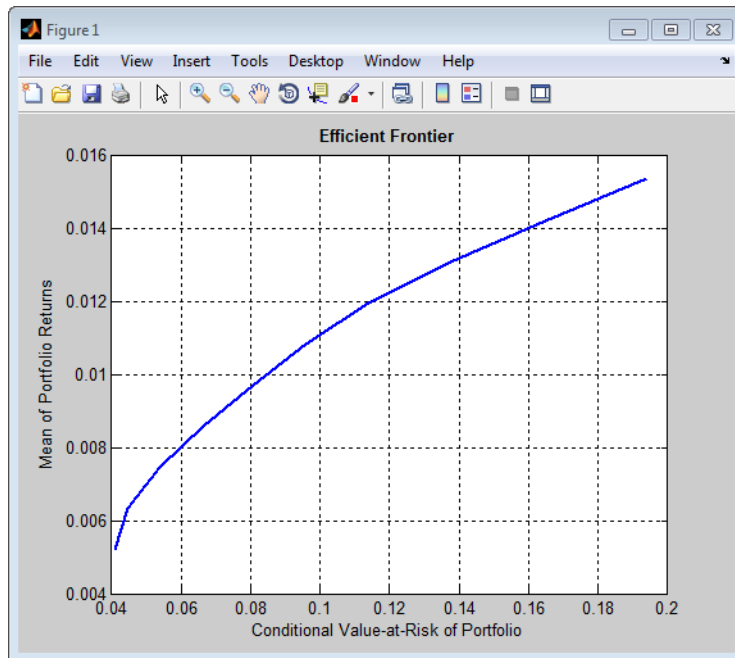
p = PortfolioCVaR('Scenarios', AssetScenarios, ...
  'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1, ...
```

```
'ProbabilityLevel', 0.95);
```

Note that the `LowerBound` property value undergoes scalar expansion since `AssetScenarios` provides the dimensions of the problem.

You can use dot notation with the method `plotFrontier`.

```
plotFrontier(p);
```



Using the Constructor with a Sequence of Steps

An alternative way to accomplish the same task of setting up a “standard” CVaR portfolio optimization problem, given `AssetScenarios` variable is:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
m = m/12;
```

```

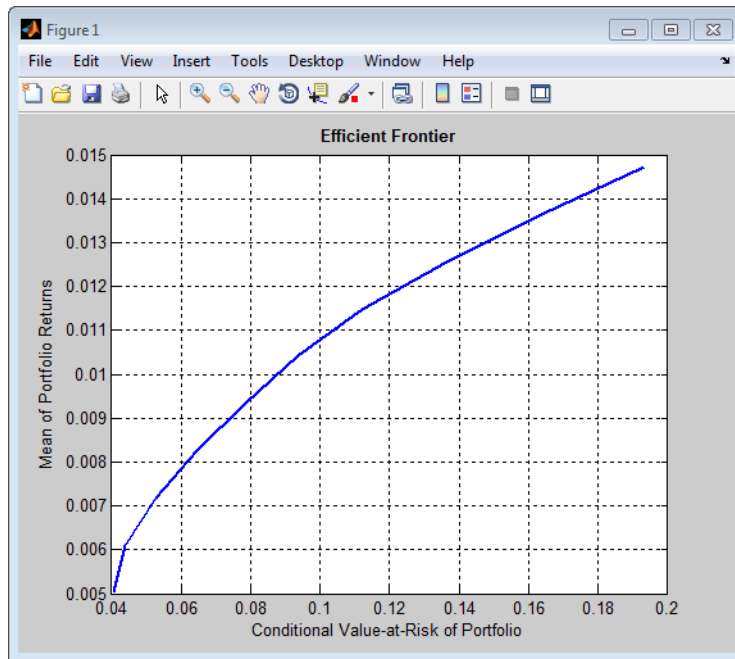
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = PortfolioCVaR(p, 'LowerBound', 0);
p = PortfolioCVaR(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);

```



This way works because the calls to the constructor are in this particular order. In this case, the call to initialize `AssetScenarios` provides the dimensions for the problem. If you were to do this step last, you would have to explicitly dimension the `LowerBound` property as follows:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;

```

```

0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = PortfolioCVaR(p, 'LowerBound', zeros(size(m)));
p = PortfolioCVaR(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setProbabilityLevel(p, 0.95);
p = setScenarios(p, AssetScenarios);

```

Note: If you did not specify the size of `LowerBound` but, instead, input a scalar argument, the constructor assumes that you are defining a single-asset problem and produces an error at the call to set asset scenarios with four assets.

Shortcuts for Property Names

The CVaR portfolio constructor has shorter argument names that replace longer argument names associated with specific properties of the PortfolioCVaR object. For example, rather than enter `'ProbabilityLevel'`, the constructor accepts the case-insensitive name `'plevel'` to set the `ProbabilityLevel` property in a PortfolioCVaR object. Every shorter argument name corresponds with a single property in the CVaR portfolio constructor. The one exception is the alternative argument name `'budget'`, which signifies both the `LowerBudget` and `UpperBudget` properties. When `'budget'` is used, then the `LowerBudget` and `UpperBudget` properties are set to the same value to form an equality budget constraint.

Shortcut Argument Name	Equivalent Argument / Property Name
ae	AEquality
ai	AInequality
assetnames or assets	AssetList
be	bEquality
bi	bInequality
budget	UpperBudget and LowerBudget

Shortcut Argument Name	Equivalent Argument / Property Name
group	GroupMatrix
lb	LowerBound
n or num	NumAssets
level, problevel, or plevel	ProbabilityLevel
rfr	RiskFreeRate
scenario or assetscenarios	Scenarios
ub	UpperBound

For example, this call to the constructor uses these shortcuts for properties:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('scenario', AssetScenarios, 'lb', 0, 'budget', 1, 'plevel', 0.95);
plotFrontier(p);
```

Direct Setting of Portfolio Object Properties

Although not recommended, you can set properties directly using dot notation, however no error-checking is done on your inputs:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;

p = setScenarios(p, AssetScenarios);
```

```
p.ProbabilityLevel = 0.95;  
  
p.LowerBudget = 1;  
p.UpperBudget = 1;  
p.LowerBound = zeros(size(m));  
  
plotFrontier(p);
```

Note: Scenarios cannot be assigned directly to a PortfolioCVaR object. Scenarios must always be set through either the PortfolioCVaR constructor, the setScenarios method, or any of the scenario simulation methods.

See Also

PortfolioCVaR | estimateBounds

Related Examples

- “Common Operations on the PortfolioCVaR Object” on page 5-34
- “Working with CVaR Portfolio Constraints” on page 5-56

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Common Operations on the PortfolioCVaR Object

In this section...

“Naming a PortfolioCVaR Object” on page 5-34

“Configuring the Assets in the Asset Universe” on page 5-34

“Setting Up a List of Asset Identifiers” on page 5-35

“Truncating and Padding Asset Lists” on page 5-36

“Setting Up an Initial or Current Portfolio” on page 5-37

Naming a PortfolioCVaR Object

To name a PortfolioCVaR object, use the `Name` property. `Name` is informational and has no effect on any portfolio calculations. If the `Name` property is nonempty, `Name` is the title for the efficient frontier plot generated by `plotFrontier`. For example, if you set up an asset allocation fund, you could name the PortfolioCVaR object Asset Allocation Fund:

```
p = PortfolioCVaR('Name','Asset Allocation Fund');  
disp(p.Name);
```

```
Asset Allocation Fund
```

Configuring the Assets in the Asset Universe

The fundamental quantity in the PortfolioCVaR object is the number of assets in the asset universe. This quantity is maintained in the `NumAssets` property. Although you can set this property directly, it is usually derived from other properties such as the number of assets in the scenarios or the initial portfolio. In some instances, the number of assets may need to be set directly. This example shows how to set up a PortfolioCVaR object that has four assets:

```
p = PortfolioCVaR('NumAssets', 4);  
disp(p.NumAssets);
```

```
4
```

After setting the `NumAssets` property, you cannot modify it (unless no other properties are set that depend on `NumAssets`). The only way to change the number of assets in an existing PortfolioCVaR object with a known number of assets is to create a new PortfolioCVaR object.

Setting Up a List of Asset Identifiers

When working with portfolios, you must specify a universe of assets. Although you can perform a complete analysis without naming the assets in your universe, it is helpful to have an identifier associated with each asset as you create and work with portfolios. You can create a list of asset identifiers as a cell vector of strings in the property `AssetList`. You can set up the list using the next two methods.

Setting Up Asset Lists Using the Constructor

Suppose you have a `PortfolioCVaR` object, `p`, with assets with symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR'. You can create a list of these asset symbols in the object using the constructor:

```
p = PortfolioCVaR('assetlist', { 'AA', 'BA', 'CAT', 'DD', 'ETR' });
disp(p.AssetList);
```

```
'AA'      'BA'      'CAT'      'DD'      'ETR'
```

Notice that the property `AssetList` is maintained as a cell array that contains strings, and that it is necessary to pass a cell array into the constructor to set `AssetList`. In addition, notice that the property `NumAssets` is set to 5 based on the number of symbols used to create the asset list:

```
disp(p.NumAssets);
```

```
5
```

Setting Up Asset Lists Using the `setAssetList` Method

You can also specify a list of assets using the method `setAssetList`. Given the list of asset symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR', you can use `setAssetList` with:

```
p = PortfolioCVaR;
p = setAssetList(p, { 'AA', 'BA', 'CAT', 'DD', 'ETR' });
disp(p.AssetList);
```

```
'AA'      'BA'      'CAT'      'DD'      'ETR'
```

`setAssetList` also enables you enter symbols directly as a comma-separated list without creating a cell array of strings. For example, given the list of assets symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR', use `setAssetList`:

```
p = PortfolioCVaR;
p = setAssetList(p, 'AA', 'BA', 'CAT', 'DD', 'ETR');
```

```
disp(p.AssetList);
'AA'      'BA'      'CAT'      'DD'      'ETR'
```

`setAssetList` has many additional features to create lists of asset identifiers. If you use `setAssetList` with just a `PortfolioCVaR` object, it creates a default asset list according to the name specified in the hidden public property `defaultforAssetList` (which is `'Asset'` by default). The number of asset names created depends on the number of assets in the property `NumAssets`. If `NumAssets` is not set, then `NumAssets` is assumed to be 1.

For example, if a `PortfolioCVaR` object `p` is created with `NumAssets = 5`, then this code fragment shows the default naming behavior:

```
p = PortfolioCVaR('numassets',5);
p = setAssetList(p);
disp(p.AssetList);

'Asset1'      'Asset2'      'Asset3'      'Asset4'      'Asset5'
```

Suppose that your assets are, for example, ETFs and you change the hidden property `defaultforAssetList` to `'ETF'`, you can then create a default list for ETFs:

```
p = PortfolioCVaR('numassets',5);
p.defaultforAssetList = 'ETF';
p = setAssetList(p);
disp(p.AssetList);
'ETF1'      'ETF2'      'ETF3'      'ETF4'      'ETF5'
```

Truncating and Padding Asset Lists

If the `NumAssets` property is already set and you pass in too many or too few identifiers, the `CVaR` portfolio constructor, and the `setAssetList` method truncate or pad the list with numbered default asset names that use the name specified in the hidden public property `defaultforAssetList`. If the list is truncated or padded, a warning message indicates the discrepancy. For example, assume you have a `PortfolioCVaR` object with five ETFs and you only know the first three CUSIPs `'921937835'`, `'922908769'`, and `'922042775'`. Use this syntax to create an asset list that pads the remaining asset identifiers with numbered `'UnknownCUSIP'` placeholders:

```
p = PortfolioCVaR('numassets',5);
p.defaultforAssetList = 'UnknownCUSIP';
p = setAssetList(p, '921937835', '922908769', '922042775');
disp(p.AssetList);
```

Warning: Input list of assets has 2 too few identifiers. Padding with numbered assets.

```
> In PortfolioCVaR.setAssetList at 118
'921937835' '922908769' '922042775' 'UnknownCUSIP4' 'UnknownCUSIP5'
```

Alternatively, suppose you have too many identifiers and need only the first four assets. This example illustrates truncation of the asset list using the `PortfolioCVaR` constructor:

```
p = PortfolioCVaR('numassets',4);
p = PortfolioCVaR(p, 'assetlist', { 'AGG', 'EEM', 'MDY', 'SPY', 'VEU' });
disp(p.AssetList);
```

```
Warning: AssetList has 1 too many identifiers. Using first 4 assets.
> In PortfolioCVaR.checkarguments at 399
In PortfolioCVaR.PortfolioCVaR>PortfolioCVaR.PortfolioCVaR at 195
'AGG' 'EEM' 'MDY' 'SPY'
```

The hidden public property `uppercaseAssetList` is a Boolean flag to specify whether to convert asset names to uppercase letters. The default value for `uppercaseAssetList` is `false`. This example shows how to use the `uppercaseAssetList` flag to force identifiers to be uppercase letters:

```
p = PortfolioCVaR;
p.uppercaseAssetList = true;
p = setAssetList(p, { 'aa', 'ba', 'cat', 'dd', 'etr' });
disp(p.AssetList);
```

```
'AA' 'BA' 'CAT' 'DD' 'ETR'
```

Setting Up an Initial or Current Portfolio

In many applications, creating a new optimal portfolio requires comparing the new portfolio with an initial or current portfolio to form lists of purchases and sales. The `PortfolioCVaR` object property `InitPort` lets you identify an initial or current portfolio. The initial portfolio also plays an essential role if you have either transaction costs or turnover constraints. The initial portfolio need not be feasible within the constraints of the problem. This can happen if the weights in a portfolio have shifted such that some constraints become violated. To check if your initial portfolio is feasible, use the method `checkFeasibility` described in “Validating CVaR Portfolios” on page 5-80. Suppose you have an initial portfolio in x_0 , then use the `PortfolioCVaR` object constructor to set up an initial portfolio:

```
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = PortfolioCVaR('InitPort', x0);
disp(p.InitPort);
```

```
0.3000
```

```
0.2000
0.2000
0
```

As with all array properties, you can set `InitPort` with scalar expansion. This is helpful to set up an equally weighted initial portfolio of, for example, 10 assets:

```
p = PortfolioCVaR('NumAssets', 10, 'InitPort', 1/10);
disp(p.InitPort);
```

```
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
0.1000
```

To clear an initial portfolio from your `PortfolioCVaR` object, use either the constructor or the `setInitPort` method with an empty input for the `InitPort` property. If transaction costs or turnover constraints are set, it is not possible to clear the `InitPort` property in this way. In this case, to clear `InitPort`, first clear the dependent properties and then clear the `InitPort` property.

The `InitPort` property can also be set with `setInitPort` which lets you specify the number of assets if you want to use scalar expansion. For example, given an initial portfolio in `x0`, use `setInitPort` to set the `InitPort` property:

```
p = PortfolioCVaR;
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = setInitPort(p, x0);
disp(p.InitPort);
```

```
0.3000
0.2000
0.2000
0
```

To create an equally weighted portfolio of four assets, use `setInitPort`:

```
p = PortfolioCVaR;
p = setInitPort(p, 1/4, 4);
```

```
disp(p.InitPort);
0.2500
0.2500
0.2500
0.2500
```

PortfolioCVaR object methods that work with either transaction costs or turnover constraints also depend on the `InitPort` property. Consequently, the set methods for transaction costs or turnover constraints permit the assignment of a value for the `InitPort` property as part of their implementation. For details, see “Working with Average Turnover Constraints” on page 5-72, “Working with One-Way Turnover Constraints” on page 5-74, and “Working with Transaction Costs” on page 5-51. If either transaction costs or turnover constraints are used, then the `InitPort` property must have a nonempty value. Absent a specific value assigned through the constructor or various set methods, the PortfolioCVaR object sets `InitPort` to 0 and warns if `BuyCost`, `SellCost`, or `Turnover` properties are set. This example shows what happens if you specify an average turnover constraint with an initial portfolio:

```
p = PortfolioCVaR('Turnover', 0.3, 'InitPort', [ 0.3; 0.2; 0.2; 0.0 ]);
disp(p.InitPort);

0.3000
0.2000
0.2000
0
```

In contrast, this example shows what happens if an average turnover constraint is specified without an initial portfolio:

```
p = PortfolioCVaR('Turnover', 0.3);
disp(p.InitPort);

Warning: InitPort and NumAssets are empty and either transaction costs or turnover constraints specified.
Will set NumAssets = 1 and InitPort = 0.
> In PortfolioCVaR.checkarguments at 322
   In PortfolioCVaR.PortfolioCVaR>PortfolioCVaR.PortfolioCVaR at 195
0
```

See Also

PortfolioCVaR | checkFeasibility | estimateBounds | setAssetList | setInitPort

Related Examples

- “Working with CVaR Portfolio Constraints” on page 5-56

- “Asset Returns and Scenarios” on page 5-41

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Asset Returns and Scenarios

In this section...

- “How Stochastic Optimization Works” on page 5-41
- “What Are Scenarios?” on page 5-42
- “Setting Scenarios Using the PortfolioCVaR Constructor” on page 5-42
- “Setting Scenarios Using the setScenarios Method” on page 5-44
- “Estimating the Mean and Covariance of Scenarios” on page 5-44
- “Simulating Normal Scenarios” on page 5-45
- “Simulating Normal Scenarios from Returns or Prices” on page 5-45
- “Simulating Normal Scenarios with Missing Data” on page 5-47
- “Simulating Normal Scenarios from Time Series Data” on page 5-49
- “Simulating Normal Scenarios with Mean and Covariance” on page 5-50
- “Working with a Riskless Asset” on page 5-51
- “Working with Transaction Costs” on page 5-51

How Stochastic Optimization Works

The CVaR of a portfolio is a conditional expectation. (For the definition of the CVaR function, see “Risk Proxy” on page 5-5.) Therefore, the CVaR portfolio optimization problem is a stochastic optimization problem. Given a sample of scenarios, the conditional expectation that defines the sample CVaR of the portfolio can be expressed as a finite sum, a weighted average of losses. The weights of the losses depend on their relative magnitude; for a confidence level α , only the worst $(1 - \alpha) \times 100\%$ losses get a positive weight. As a function of the portfolio weights, the CVaR of the portfolio is a convex function (see [48], [49] Rockafellar & Uryasev at “Portfolio Optimization” on page A-10). It is also a nonsmooth function, but its edges are less sharp as the sample size increases.

There are reformulations of the CVaR portfolio optimization problem (see [48], [49] at Rockafellar & Uryasev) that result in a linear programming problem, which can be solved either with standard linear programming techniques or with stochastic programming solvers. The PortfolioCVaR object, however, does not reformulate the problem in such a manner. The PortfolioCVaR object computes the CVaR as a nonlinear function. The

convexity of the CVaR, as a function of the portfolio weights and the somewhat dull edges when the number of scenarios is large, make the CVaR portfolio optimization problem tractable, in practice, for certain nonlinear programming solvers, such as `fmincon` from Optimization Toolbox. The problem can also be solved using a cutting-plane method (see Kelley [45] at “Portfolio Optimization” on page A-10). For more information, see Algorithms section of `setSolver`. To learn more about the workflow when using `PortfolioCVaR` objects, see “PortfolioCVaR Object Workflow”.

What Are Scenarios?

Since conditional value-at-risk portfolio optimization works with scenarios of asset returns to perform the optimization, several ways exist to specify and simulate scenarios. In many applications with CVaR portfolio optimization, asset returns may have distinctly nonnormal probability distributions with either multiple modes, binning of returns, truncation of distributions, and so forth. In other applications, asset returns are modeled as the result of various simulation methods that might include Monte-Carlo simulation, quasi-random simulation, and so forth. In many cases, the underlying probability distribution for risk factors may be multivariate normal but the resultant transformations are sufficiently nonlinear to result in distinctively nonnormal asset returns.

For example, this occurs with bonds and derivatives. In the case of bonds with a nonzero probability of default, such scenarios would likely include asset returns that are -100% to indicate default and some values slightly greater than -100% to indicate recovery rates.

Although the `PortfolioCVaR` object has methods to simulate multivariate normal scenarios from either data or moments (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`), the usual approach is to specify scenarios directly from your own simulation methods. These scenarios are entered directly as a matrix with a sample for all assets across each row of the matrix and with samples for an asset down each column of the matrix. The architecture of the CVaR portfolio optimization tools references the scenarios through a function handle so scenarios that have been set cannot be accessed directly as a property of the `PortfolioCVaR` object.

Setting Scenarios Using the PortfolioCVaR Constructor

Suppose you have a matrix of scenarios in the `AssetScenarios` variable. The scenarios are set through the `PortfolioCVaR` constructor with:


```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('Scenarios', AssetScenarios);

disp(p.NumAssets);
disp(p.NumScenarios);

4

20000

```

Notice that the `PortfolioCVaR` object determines and fixes the number of assets in `NumAssets` and the number of scenarios in `NumScenarios` based on the scenario's matrix. You can change the number of scenarios by calling the `PortfolioCVaR` constructor with a different scenario matrix. However, once the `NumAssets` property has been set in the object, you cannot enter a scenario matrix with a different number of assets. The `getScenarios` method lets you recover scenarios from a `PortfolioCVaR` object. You can also obtain the mean and covariance of your scenarios using `estimateScenarioMoments`.

Although not recommended for the casual user, an alternative way exists to recover scenarios by working with the function handle that points to scenarios in the `PortfolioCVaR` object. To access some or all of the scenarios from a `PortfolioCVaR` object, the hidden property `localScenarioHandle` is a function handle that points to a function to obtain scenarios that have already been set. To get scenarios directly from a `PortfolioCVaR` object `p`, use

```

scenarios = p.localScenarioHandle([], []);
and to obtain a subset of scenarios from rows startrow to endrow, use

scenarios = p.localScenarioHandle(startrow, endrow);
where  $1 \leq \text{startrow} \leq \text{endrow} \leq \text{numScenarios}$ .

```

Setting Scenarios Using the setScenarios Method

You can also set scenarios using `setScenarios`. For example, given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);

disp(p.NumAssets);
disp(p.NumScenarios);
```

```
4
```

```
20000
```

Estimating the Mean and Covariance of Scenarios

The `estimateScenarioMoments` method obtains estimates for the mean and covariance of scenarios in a `PortfolioCVaR` object.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
[mean, covar] = estimateScenarioMoments(p)

mean =

    0.0043
    0.0085
    0.0098
    0.0153

covar =

    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0049    0.0029
    0.0000    0.0010    0.0029    0.0102

```

Simulating Normal Scenarios

As a convenience, the two methods (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`) exist to simulate scenarios from data or moments under an assumption that they are distributed as multivariate normal random asset returns.

Simulating Normal Scenarios from Returns or Prices

Given either return or price data, use the method `simulateNormalScenariosByData` to simulate multivariate normal scenarios. Either returns or prices are stored as matrices with samples going down the rows and assets going across the columns. In addition, returns or prices can be stored in a financial time series `fints` object (see “Simulating Normal Scenarios from Time Series Data” on page 5-49). To illustrate using `simulateNormalScenariosByData`, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with `portsim`. The default behavior of `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by `portsim` in the variable `X`, a price series is created in the variable `Y`:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
```

```
C = [ 0.005333 0.00034 0.00016 0;  
0.00034 0.002408 0.0017 0.000992;  
0.00016 0.0017 0.0048 0.0028;  
0 0.000992 0.0028 0.010208 ];
```

```
X = portsim(m', C, 120);  
Y = ret2tick(X);
```

Note: Portfolio optimization requires that you use total returns and not just price returns. Consequently, “returns” should be total returns and “prices” should be total return prices.

Given asset returns and prices in variables X and Y from above, this sequence of examples demonstrates equivalent ways to simulate multivariate normal scenarios for the PortfolioCVaR object. Assume a PortfolioCVaR object created in p that uses the asset returns in X uses `simulateNormalScenariosByData`:

```
p = PortfolioCVaR;  
p = simulateNormalScenariosByData(p, X, 20000);  
  
[passetmean, passetcovar] = estimateScenarioMoments(p)  
  
passetmean =  
  
    0.0043  
    0.0083  
    0.0102  
    0.1507  
  
passetcovar =  
  
    0.0053    0.0003    0.0002    0.0000  
    0.0003    0.0024    0.0017    0.0010  
    0.0002    0.0017    0.0049    0.0028  
    0.0000    0.0010    0.0028    0.0101
```

The moments that you obtain from this simulation will likely differ from the moments listed here because the scenarios are random samples from the estimated multivariate normal probability distribution of the input returns X .

The default behavior of `simulateNormalScenariosByData` is to work with asset returns. If, instead, you have asset prices as in the variable Y ,

`simulateNormalScenariosByData` accepts an argument name `'DataFormat'` with a corresponding value set to `'prices'` to indicate that the input to the method is in the form of asset prices and not returns (the default value for the `'DataFormat'` argument is `'returns'`). This example simulates scenarios with the asset price data in `Y` for the `PortfolioCVaR` object `q`:

```
p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0043
    0.0084
    0.0094
    0.1490

passetcovar =

    0.0054    0.0004    0.0001   -0.0000
    0.0004    0.0024    0.0016    0.0009
    0.0001    0.0016    0.0048    0.0028
   -0.0000    0.0009    0.0028    0.0100
```

Simulating Normal Scenarios with Missing Data

Often when working with multiple assets, you have missing data indicated by NaN values in your return or price data. Although “Multivariate Normal Regression” on page 9-2 goes into detail about regression with missing data, the `simulateNormalScenariosByData` method has an argument name `'MissingData'` that indicates with a Boolean value whether to use the missing data capabilities of Financial Toolbox. The default value for `'MissingData'` is `false` which removes all samples with NaN values. If, however, `'MissingData'` is set to `true`, `simulateNormalScenariosByData` uses the ECM algorithm to estimate asset moments. This example shows how this works on price data with missing values:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
      0.00034 0.002408 0.0017 0.000992;
      0.00016 0.0017 0.0048 0.0028;
      0 0.000992 0.0028 0.010208 ];
```

```
X = portsim(m', C, 120);
Y = ret2tick(X);
Y(1:20,1) = NaN;
Y(1:12,4) = NaN;
```

Notice that the prices above in `Y` have missing values in the first and fourth series.

```
p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

q = PortfolioCVaR;
q = simulateNormalScenariosByData(q, Y, 20000, 'dataformat', 'prices', 'missingdata', true);

[passetmean, passetcovar] = estimateScenarioMoments(p)
[qassetmean, qassetcovar] = estimateScenarioMoments(q)

passetmean =

    0.0020
    0.0074
    0.0078
    0.1476

passetcovar =

    0.0055    0.0003   -0.0001   -0.0003
    0.0003    0.0024    0.0019    0.0012
   -0.0001    0.0019    0.0050    0.0028
   -0.0003    0.0012    0.0028    0.0101

qassetmean =

    0.0024
    0.0085
    0.0106
    0.1482

qassetcovar =

    0.0071    0.0004   -0.0001   -0.0004
    0.0004    0.0032    0.0022    0.0012
   -0.0001    0.0022    0.0063    0.0034
   -0.0004    0.0012    0.0034    0.0127
```

The first `PortfolioCVaR` object, `p`, contains scenarios obtained from price data in `Y` where NaN values are discarded and the second `PortfolioCVaR` object, `q`, contains scenarios obtained from price data in `Y` that accommodate missing values. Each time you run this example, you will get different estimates for the moments in `p` and `q`.

Simulating Normal Scenarios from Time Series Data

The `simulateNormalScenariosByData` method also accepts asset returns or prices stored in financial time series (`fints`) objects. The method implicitly works with matrices of data or data in a `fints` object using the same rules for whether the data are returns or prices. To illustrate, use `fints` to create the `fints` object `Xfts` that contains asset returns generated with `fints` (see “Estimating Asset Moments from Prices or Returns” on page 4-45) and add series labels:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
      0.00034 0.002408 0.0017 0.000992;
      0.00016 0.0017 0.0048 0.0028;
      0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);

d = (datetime('31-jan-2001'):datetime('31-dec-2010'))';
Xfts = fints(d, zeros(numel(d),4), {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'});
Xfts = tomonthly(Xfts);

Xfts.Bonds = X(:,1);
Xfts.LargeCap = X(:,2);
Xfts.SmallCap = X(:,3);
Xfts.Emerging = X(:,4);

p = PortfolioCVar;
p = simulateNormalScenariosByData(p, Xfts, 20000);

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0044
    0.0082
    0.0102
    0.1504

passetcovar =

    0.0054    0.0004    0.0002   -0.0000
    0.0004    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0047    0.0027
   -0.0000    0.0010    0.0027    0.0102
```

The name-value inputs `'DataFormat'` to handle return or price data and `'MissingData'` to ignore or use samples with missing values also work for `fints` data. In addition, `simulateNormalScenariosByData` extracts asset names or identifiers from a `fints` object if the argument name `'GetAssetList'` is set to `true`

(the default value is `false`). If the `'GetAssetList'` value is `true`, the identifiers are used to set the `AssetList` property of the `PortfolioCVaR` object. Thus, repeating the formation of the `PortfolioCVaR` object `q` from the previous example with the `'GetAssetList'` flag set to `true` extracts the series labels from the `fints` object:

```
p = simulateNormalScenariosByData(p, Xfts, 20000, 'getassetlist', true);  
disp(p.AssetList)
```

```
'Bonds' 'LargeCap' 'SmallCap' 'Emerging'
```

If you set the `'GetAssetList'` flag set to `true` and your input data is in a matrix, `simulateNormalScenariosByData` uses the default labeling scheme from `setAssetList` as described in “Setting Up a List of Asset Identifiers” on page 5-35.

Simulating Normal Scenarios with Mean and Covariance

Given the mean and covariance of asset returns, use the `simulateNormalScenariosByMoments` method to simulate multivariate normal scenarios. The mean can be either a row or column vector and the covariance matrix must be a symmetric positive-semidefinite matrix. Various rules for scalar expansion apply. To illustrate using `simulateNormalScenariosByMoments`, start with moments in `m` and `C` and generate 20,000 scenarios:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];  
C = [ 0.005333 0.00034 0.00016 0;  
0.00034 0.002408 0.0017 0.000992;  
0.00016 0.0017 0.0048 0.0028;  
0 0.000992 0.0028 0.010208 ];
```

```
p = PortfolioCVaR;  
p = simulateNormalScenariosByMoments(p, m, C, 20000);  
[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =
```

```
0.0049  
0.0083  
0.0101  
0.1503
```

```
passetcovar =
```



```

0.0053    0.0003    0.0002   -0.0000
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0047    0.0028
-0.0000    0.0010    0.0028    0.0101

```

`simulateNormalScenariosByMoments` performs scalar expansion on arguments for the moments of asset returns. If `NumAssets` has not already been set, a scalar argument is interpreted as a scalar with `NumAssets` set to 1. `simulateNormalScenariosByMoments` provides an additional optional argument to specify the number of assets so that scalar expansion works with the correct number of assets. In addition, if either a scalar or vector is input for the covariance of asset returns, a diagonal matrix is formed such that a scalar expands along the diagonal and a vector becomes the diagonal.

Working with a Riskless Asset

The `PortfolioCVaR` object has a separate `RiskFreeRate` property that stores the rate of return of a riskless asset. Thus, you can separate your universe into a riskless asset and a collection of risky assets. For example, assume that your riskless asset has a return in the scalar variable `r0`, then the property for the `RiskFreeRate` is set using the constructor:

```

r0 = 0.01/12;

p = PortfolioCVaR;
p = PortfolioCVaR('RiskFreeRate', r0);
disp(p.RiskFreeRate);

8.3333e-04

```

Note: If your portfolio problem has a budget constraint such that your portfolio weights must sum to 1, then the riskless asset is irrelevant.

Working with Transaction Costs

The difference between net and gross portfolio returns is transaction costs. The net portfolio return proxy has distinct proportional costs to purchase and to sell assets which are maintained in the `PortfolioCVaR` object properties `BuyCost` and `SellCost`. Transaction costs are in units of total return and, as such, are proportional to the price

of an asset so that they enter the model for net portfolio returns in return form. For example, suppose you have a stock currently priced \$40 and your usual transaction costs are 5 cents per share. Then the transaction cost for the stock is $0.05/40 = 0.00125$ (as defined in “Net Portfolio Returns” on page 5-4). Costs are entered as positive values and credits are entered as negative values.

Setting Transaction Costs Using the Constructor

To set up transaction costs, you must specify an initial or current portfolio in the `InitPort` property. If the initial portfolio is not set when you set up the transaction cost properties, `InitPort` is 0. The properties for transaction costs can be set using the constructor `PortfolioCVaR`. For example, assume that purchase and sale transaction costs are in the variables `bc` and `sc` and an initial portfolio is in the variable `x0`, then transaction costs are set:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioCVaR('BuyCost', bc, 'SellCost', sc, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

5

0.0013
0.0013
0.0013
0.0013
0.0013

0.0013
0.0070
0.0013
0.0013
0.0024

0.4000
0.2000
0.2000
0.1000
0.1000

Setting Transaction Costs Using the `setCosts` Method

You can also set the properties for transaction costs using `setCosts`. Assume that you have the same costs and initial portfolio as in the previous example. Given a `PortfolioCVaR` object `p` with an initial portfolio already set, use `setCosts` to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = PortfolioCVaR('InitPort', x0);
p = setCosts(p, bc, sc);
```

```
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5
0.0013
0.0013
0.0013
0.0013
0.0013
0.0013
0.0070
0.0013
0.0013
0.0024
0.4000
0.2000
0.2000
0.1000
0.1000
```

You can also set up the initial portfolio's `InitPort` value as an optional argument to `setCosts` so that the following is an equivalent way to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
```

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioCVaR;
p = setCosts(p, bc, sc, x0);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);

5

0.0013
0.0013
0.0013
0.0013
0.0013

0.0013
0.0070
0.0013
0.0013
0.0024

0.4000
0.2000
0.2000
0.1000
0.1000
```

Setting Transaction Costs with Scalar Expansion

Both the constructor `PortfolioCVaR` and `setCosts` method implement scalar expansion on the arguments for transaction costs and the initial portfolio. If the `NumAssets` property is already set in the `PortfolioCVaR` object, scalar arguments for these properties are expanded to have the same value across all dimensions. In addition, `setCosts` lets you specify `NumAssets` as an optional final argument. For example, assume that you have an initial portfolio `x0` and you want to set common transaction costs on all assets in your universe. You can set these costs in any of these equivalent ways:

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioCVaR('InitPort', x0, 'BuyCost', 0.002, 'SellCost', 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];  
p = PortfolioCVaR('InitPort', x0);  
p = setCosts(p, 0.002, 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];  
p = PortfolioCVaR;  
p = setCosts(p, 0.002, 0.002, x0);
```

To clear costs from your PortfolioCVaR object, use either the constructor or `setCosts` with empty inputs for the properties to be cleared. For example, you can clear sales costs from the PortfolioCVaR object `p` in the previous example:

```
p = PortfolioCVaR(p, 'SellCost', []);
```

See Also

PortfolioCVaR | estimatePortVaR | setCosts | setProbabilityLevel
| setScenarios | simulateNormalScenariosByData |
simulateNormalScenariosByMoments

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios” on page 5-83
- “Estimate Efficient Frontiers” on page 5-95
- “Asset Returns and Scenarios” on page 5-41

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Working with CVaR Portfolio Constraints

In this section...

“Setting Default Constraints for Portfolio Weights” on page 5-56

“Working with Bound Constraints” on page 5-59

“Working with Budget Constraints” on page 5-61

“Working with Group Constraints” on page 5-63

“Working with Group Ratio Constraints” on page 5-66

“Working with Linear Equality Constraints” on page 5-68

“Working with Linear Inequality Constraints” on page 5-70

“Working with Average Turnover Constraints” on page 5-72

“Working with One-Way Turnover Constraints” on page 5-74

The final element for a complete specification of a portfolio optimization problem is the set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset \mathcal{R}^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or “default” portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to 1 (using the budget constraint). For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow”.

Setting Default Constraints for Portfolio Weights

The “default” CVaR portfolio problem has two constraints on portfolio weights:

- Portfolio weights must be nonnegative.
- Portfolio weights must sum to 1.

Implicitly, these constraints imply that portfolio weights are no greater than 1, although this is a superfluous constraint to impose on the problem.

Setting Default Constraints Using the Constructor

Given a portfolio optimization problem with `NumAssets = 20` assets, use the `PortfolioCVaR` constructor to set up a default problem and explicitly set bounds and budget constraints:

```
p = PortfolioCVaR('NumAssets', 20, 'LowerBound', 0, 'Budget', 1);
disp(p);
```

PortfolioCVaR with properties:

```

    BuyCost: []
    SellCost: []
    RiskFreeRate: []
    ProbabilityLevel: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
    Name: []
    NumAssets: 20
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: [20x1 double]
    UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
    GroupA: []
    GroupB: []
    LowerRatio: []
    UpperRatio: []
```

Setting Default Constraints Using the `setDefaultConstraints` Method

An alternative approach is to use the `setDefaultConstraints` method. If the number of assets is already known in a `PortfolioCVaR` object, use `setDefaultConstraints` with no arguments to set up the necessary bound and budget constraints. Suppose you have 20 assets to set up the portfolio set for a default problem:

```
p = PortfolioCVaR('NumAssets', 20);  
p = setDefaultConstraints(p);  
disp(p);
```

PortfolioCVaR with properties:

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []  
ProbabilityLevel: []  
Turnover: []  
BuyTurnover: []  
SellTurnover: []  
NumScenarios: []  
Name: []  
NumAssets: 20  
AssetList: []  
InitPort: []  
AInequality: []  
bInequality: []  
AEquality: []  
bEquality: []  
LowerBound: [20x1 double]  
UpperBound: []  
LowerBudget: 1  
UpperBudget: 1  
GroupMatrix: []  
LowerGroup: []  
UpperGroup: []  
GroupA: []  
GroupB: []  
LowerRatio: []  
UpperRatio: []
```

If the number of assets is unknown, `setDefaultConstraints` accepts `NumAssets` as an optional argument to form a portfolio set for a default problem. Suppose you have 20 assets:

```
p = PortfolioCVaR;  
p = setDefaultConstraints(p, 20);  
disp(p);
```

PortfolioCVaR with properties:

```
BuyCost: []
```



```

    SellCost: []
    RiskFreeRate: []
    ProbabilityLevel: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
    Name: []
    NumAssets: 20
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: [20x1 double]
    UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
    GroupA: []
    GroupB: []
    LowerRatio: []
    UpperRatio: []

```

Working with Bound Constraints

Bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “Bound Constraints” on page 5-9). Although every CVaR portfolio set must be bounded, it is not necessary to specify a CVaR portfolio set with explicit bound constraints. For example, you can create a CVaR portfolio set with an implicit upper bound constraint or a CVaR portfolio set with just average turnover constraints. The bound constraints have properties `LowerBound` for the lower-bound constraint and `UpperBound` for the upper-bound constraint. Set default values for these constraints using the `setDefaultConstraints` method (see “Setting Default Constraints for Portfolio Weights” on page 5-56).

Setting Bounds Using the Constructor

The properties for bound constraints are set through the constructor `PortfolioCVaR`. Suppose you have a balanced fund with stocks that can range from 50 to 75% of

your portfolio and bonds that can range from 25 to 50% of your portfolio. The bound constraints for a balanced fund are set with:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioCVaR('LowerBound', lb, 'UpperBound', ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
2
```

```
0.5000
0.2500
```

```
0.7500
0.5000
```

To continue with this example, you must set up a budget constraint. For details, see “Working with Budget Constraints” on page 5-61.

Setting Bounds Using the `setBounds` Method

You can also set the properties for bound constraints using `setBounds`. Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. Given a `PortfolioCVaR` object `p`, use `setBounds` to set the bound constraints:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioCVaR;
p = setBounds(p, lb, ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
2
```

```
0.5000
0.2500
```

```
0.7500
0.5000
```

Setting Bounds Using the Constructor or `setBounds` Method

Both the constructor `PortfolioCVaR` and `setBounds` method implement scalar expansion on either the `LowerBound` or `UpperBound` properties. If the `NumAssets` property is already set in the `PortfolioCVaR` object, scalar arguments for either property expand to have the same value across all dimensions. In addition, `setBounds` lets you specify `NumAssets` as an optional argument. Suppose you have a universe of 500 assets and you want to set common bound constraints on all assets in your universe. Specifically, you are a long-only investor and want to hold no more than 5% of your portfolio in any single asset. You can set these bound constraints in any of these equivalent ways:

```
p = PortfolioCVaR('NumAssets', 500, 'LowerBound', 0, 'UpperBound', 0.05);
```

or

```
p = PortfolioCVaR('NumAssets', 500);
p = setBounds(p, 0, 0.05);
```

or

```
p = PortfolioCVaR;
p = setBounds(p, 0, 0.05, 500);
```

To clear bound constraints from your `PortfolioCVaR` object, use either the `PortfolioCVaR` constructor or `setBounds` with empty inputs for the properties to be cleared. For example, to clear the upper-bound constraint from the `PortfolioCVaR` object `p` in the previous example:

```
p = PortfolioCVaR(p, 'UpperBound', []);
```

Working with Budget Constraints

The budget constraint is an optional linear constraint that maintains upper and lower bounds on the sum of portfolio weights (see “Budget Constraints” on page 5-10). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a CVaR portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to 1 using `setDefaultConstraints` (see “Setting Default Constraints Using the Constructor” on page 5-57).

Setting Budget Constraints Using the Constructor

The properties for the budget constraint can also be set using the constructor `PortfolioCVaR`. Suppose you have an asset universe with many risky assets and a riskless asset and you want to ensure that your portfolio never holds more than 1% cash, that is, you want to ensure that you are 99 to 100% invested in risky assets. The budget constraint for this portfolio can be set with:

```
p = PortfolioCVaR('LowerBudget', 0.99, 'UpperBudget', 1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
0.9900
```

```
1
```

Setting Budget Constraints Using the `setBudget` Method

You can also set the properties for a budget constraint using `setBudget`. Suppose you have a fund that permits up to 10% leverage which means that your portfolio can be from 100 to 110% invested in risky assets. Given a `PortfolioCVaR` object `p`, use `setBudget` to set the budget constraints:

```
p = PortfolioCVaR;  
p = setBudget(p, 1, 1.1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
1
```

```
1.1000
```

If you were to continue with this example, then set the `RiskFreeRate` property to the borrowing rate to finance possible leveraged positions. For details on the `RiskFreeRate` property, see “Working with a Riskless Asset” on page 5-51. To clear either bound for the budget constraint from your `PortfolioCVaR` object, use either the constructor `PortfolioCVaR` or `setBudget` with empty inputs for the properties to be cleared. For example, clear the upper-budget constraint from the `PortfolioCVaR` object `p` in the previous example with:

```
p = PortfolioCVaR(p, 'UpperBudget', []);
```

Working with Group Constraints

Group constraints are optional linear constraints that group assets together and enforce bounds on the group weights (see “Group Constraints” on page 5-11). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with 1 or 0) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the Constructor

The properties for group constraints are set through the constructor `PortfolioCVaR`. Suppose you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio, then you can set group constraints:

```
G = [ 1 1 1 0 0 ];
p = PortfolioCVaR('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

5

```
1    1    1    0    0
```

0.3000

The group matrix `G` can also be a logical matrix so that the following code achieves the same result.

```
G = [ true true true false false ];
p = PortfolioCVaR('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

5

```
1    1    1    0    0
```

0.3000

Setting Group Constraints Using the `setGroups` and `addGroups` Methods

You can also set the properties for group constraints using `setGroups`. Suppose you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a `PortfolioCVaR` object `p`, use `setGroups` to set the group constraints:

```
G = [ true true true false false ];
p = PortfolioCVaR;
p = setGroups(p, G, [], 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5
```

```
1     1     1     0     0
```

```
0.3000
```

In this example, you would set the `LowerGroup` property to be empty (`[]`).

Suppose you want to add another group constraint to make odd-numbered assets constitute at least 20% of your portfolio. Set up an augmented group matrix and introduce infinite bounds for unconstrained group bounds or use the `addGroups` method to build up group constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioCVaR;
G = [ true true true false false ]; % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ]; % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
5
```

```
1     1     1     0     0
1     0     1     0     1
```

```
-Inf
0.2000
```

```
0.3000
```

```
Inf
```

`addGroups` determines which bounds are unbounded so you only need to focus on the constraints that you want to set.

The constructor `PortfolioCVaR` and `setGroups` and `addGroups` implement scalar expansion on either the `LowerGroup` or `UpperGroup` properties based on the dimension of the group matrix in the property `GroupMatrix`. Suppose you have a universe of 30 assets with 6 asset classes such that assets 1-5, assets 6-12, assets 13-18, assets 19-22, assets 23-27, and assets 28-30 constitute each of your asset classes and you want each asset class to fall from 0 to 25% of your portfolio. Let the following group matrix define your groups and scalar expansion define the common bounds on each group:

```
p = PortfolioCVaR;
G = blkdiag(true(1,5), true(1,7), true(1,6), true(1,4), true(1,5), true(1,3));
p = setGroups(p, G, 0, 0.25);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
30
```

```
Columns 1 through 16
```

```

1   1   1   1   1   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   1   1   1   1   1   1   1   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   1   1   1
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
Columns 17 through 30
```

```

0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   1   0   0   0   0   0   0   0   0   0   0   0   0
0   0   1   1   1   1   0   0   0   0   0   0   0   0
0   0   0   0   0   0   1   1   1   1   1   0   0   0
0   0   0   0   0   0   0   0   0   0   0   1   1   1
```

```

0
0
0
0
0
0
```

```

0.2500
0.2500
0.2500
0.2500
0.2500
0.2500
```

Working with Group Ratio Constraints

Group ratio constraints are optional linear constraints that maintain bounds on proportional relationships among groups of assets (see “Group Ratio Constraints” on page 5-12). Although the constraints are implemented as general constraints, the usual convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with 1 or 0) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group to a comparison group fall within specified bounds. Group ratio constraints have properties:

- `GroupA` for the base membership matrix
- `GroupB` for the comparison membership matrix
- `LowerRatio` for the lower-bound constraint on the ratio of groups
- `UpperRatio` for the upper-bound constraint on the ratio of groups

Setting Group Ratio Constraints Using the Constructor

The properties for group ratio constraints are set using constructor `PortfolioCVaR`. For example, assume you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ]; % financial companies
GB = [ 0 0 0 1 1 1 ]; % nonfinancial companies
p = PortfolioCVaR('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

6

```
1    1    1    0    0    0
```

```
0    0    0    1    1    1
```

0.5000

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
```



```
p = PortfolioCVaR('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6
```

```
1    1    1    0    0    0
```

```
0    0    0    1    1    1
```

```
0.5000
```

Setting Group Ratio Constraints Using the `setGroupRatio` and `addGroupRatio` Methods

You can also set the properties for group ratio constraints using `setGroupRatio`. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). Given a `PortfolioCVaR` object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6
```

```
1    1    1    0    0    0
```

```
0    0    0    1    1    1
```

```
0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose you want to add another group ratio constraint to ensure that the weights in odd-numbered assets constitute at least 20% of the weights in nonfinancial assets your portfolio. You can set up augmented group ratio matrices and introduce infinite bounds for unconstrained group ratio bounds, or you can use the `addGroupRatio` method to build up group ratio constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioCVaR;
```

```

GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ]; % odd-numbered companies
GB = [ false false false true true true ]; % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);

6

1      1      1      0      0      0
1      0      1      0      1      0

0      0      0      1      1      1
0      0      0      1      1      1

-Inf
0.2000

0.5000
Inf

```

Notice that `addGroupRatio` determines which bounds are unbounded so you only need to focus on the constraints you want to set.

The constructor `PortfolioCVaR`, `setGroupRatio`, and `addGroupRatio` implement scalar expansion on either the `LowerRatio` or `UpperRatio` properties based on the dimension of the group matrices in `GroupA` and `GroupB` properties.

Working with Linear Equality Constraints

Linear equality constraints are optional linear constraints that impose systems of equalities on portfolio weights (see “Linear Equality Constraints” on page 5-9). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `BEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the Constructor

The properties for linear equality constraints are set using the constructor `PortfolioCVaR`. Suppose you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. To set this constraint:

```

A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR('AEquality', A, 'bEquality', b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

```

```
5
```

```
1    1    1    0    0
```

```
0.5000
```

Setting Linear Equality Constraints Using the `setEquality` and `addEquality` Methods

You can also set the properties for linear equality constraints using `setEquality`. Suppose you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. Given a `PortfolioCVaR` object `p`, use `setEquality` to set the linear equality constraints:

```

A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setEquality(p, A, b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

```

```
5
```

```
1    1    1    0    0
```

```
0.5000
```

Suppose you want to add another linear equality constraint to ensure that the last three assets also constitute 50% of your portfolio. You can set up an augmented system of linear equalities or use `addEquality` to build up linear equality constraints. For this example, create another system of equalities:

```

p = PortfolioCVaR;
A = [ 1 1 1 0 0 ];    % first equality constraint
b = 0.5;
p = setEquality(p, A, b);

```

```
A = [ 0 0 1 1 1 ];    % second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000
```

The constructor `PortfolioCVaR` and `setEquality` and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

Working with Linear Inequality Constraints

Linear inequality constraints are optional linear constraints that impose systems of inequalities on portfolio weights (see “Linear Inequality Constraints” on page 5-8). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the Constructor

The properties for linear inequality constraints are set using the constructor `PortfolioCVaR`. Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. To set up these constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR('AInequality', A, 'bInequality', b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);

5

1     1     1     0     0
```

```
0.5000
```

Setting Linear Inequality Constraints Using the `setInequality` and `addInequality` Methods

You can also set the properties for linear inequality constraints using `setInequality`. Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 50% of your portfolio. Given a `PortfolioCVaR` object `p`, use `setInequality` to set the linear inequality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setInequality(p, A, b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5
```

```
1     1     1     0     0
```

```
0.5000
```

Suppose you want to add another linear inequality constraint to ensure that the last three assets constitute at least 50% of your portfolio. You can set up an augmented system of linear inequalities or use the `addInequality` method to build up linear inequality constraints. For this example, create another system of inequalities:

```
p = PortfolioCVaR;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5
```

```
1      1      1      0      0
0      0     -1     -1     -1
```

```
0.5000
-0.5000
```

The `PortfolioCVaR` constructor and `setInequality` and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

Working with Average Turnover Constraints

The turnover constraint is an optional linear absolute value constraint (see “Average Turnover Constraints” on page 5-12) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `PortfolioCVaR` constructor or the `setTurnover` method. The turnover constraint depends on an initial or current portfolio, which is assumed to be zero if not set when the turnover constraint is set. The turnover constraint has properties `Turnover`, for the upper bound on average turnover, and `InitPort`, for the portfolio against which turnover is computed.

Setting Average Turnover Constraints Using the Constructor

The properties for the turnover constraints are set using the `PortfolioCVaR` constructor. Suppose you have an initial portfolio of 10 assets in a variable `x0` and you want to ensure that average turnover is no more than 30%. To set this turnover constraint:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('Turnover', 0.3, 'InitPort', x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
```

```
0.3000
```

```
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
```

```
0.1500
0.1100
0.0800
0.1000
```

Note if the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 5-37).

Setting Average Turnover Constraints Using the `setTurnover` Method

You can also set properties for portfolio turnover using `setTurnover` to specify both the upper bound for average turnover and an initial portfolio. Suppose you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that average turnover is no more than 30%. Given a `PortfolioCVaR` object `p`, use `setTurnover` to set the turnover constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setTurnover(p, 0.3);
```

```
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
```

```
0.3000
```

```
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR;
p = setTurnover(p, 0.3, x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
0.3000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

`setTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the `PortfolioCVaR` object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setTurnover` lets you specify `NumAssets` as an optional argument. To clear turnover from your `PortfolioCVaR` object, use the constructor `PortfolioCVaR` or `setTurnover` with empty inputs for the properties to be cleared.

Working with One-Way Turnover Constraints

One-way turnover constraints are optional constraints (see “One-Way Turnover Constraints” on page 5-13) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the `PortfolioCVaR` constructor or the `setOneWayTurnover` method. One-way turnover constraints depend upon an initial or current portfolio, which is assumed to be zero if not set when the turnover constraints are set. One-way turnover constraints have properties `BuyTurnover`, for the upper bound on net purchases, `SellTurnover`, for the upper bound on net sales, and `InitPort`, for the portfolio against which turnover is computed.

Setting One-Way Turnover Constraints Using the Constructor

The Properties for the one-way turnover constraints are set using the `PortfolioCVaR` constructor. Suppose you have an initial portfolio with 10 assets in a variable `x0` and you want to ensure that turnover on purchases is no more than 30% and turnover on sales is no more than 20% of the initial portfolio. To set these turnover constraints:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('BuyTurnover', 0.3, 'SellTurnover', 0.2, 'InitPort', x0);
disp(p.NumAssets);
```



```

disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);

10

0.3000

0.2000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000

```

If the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 5-37).

Setting Turnover Constraints Using the `setOneWayTurnover` Method

You can also set properties for portfolio turnover using `setOneWayTurnover` to specify to the upper bounds for turnover on purchases (`BuyTurnover`) and sales (`SellTurnover`) and an initial portfolio. Suppose you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that turnover on purchases is no more than 30% and that turnover on sales is no more than 20% of the initial portfolio. Given a `PortfolioCVaR` object `p`, use `setOneWayTurnover` to set the turnover constraints with and without the initial portfolio being set previously:

```

x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);

disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);

```

```
10
```

```
0.3000
0.2000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR;
p = setOneWayTurnover(p, 0.3, 0.2, x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
10
0.3000
0.2000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

`setOneWayTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the `PortfolioCVaR` object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setOneWayTurnover` lets you specify `NumAssets` as an optional argument. To remove one-way turnover from your `PortfolioCVaR` object, use the constructor

PortfolioCVaR or setOneWayTurnover with empty inputs for the properties to be cleared.

See Also

PortfolioCVaR | setBounds | setBudget | setDefaultConstraints
| setEquality | setGroupRatio | setGroups | setInequality |
setOneWayTurnover | setTurnover

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios” on page 5-83
- “Estimate Efficient Frontiers” on page 5-95
- “Asset Returns and Scenarios” on page 5-41

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Validate the CVaR Portfolio Problem

In this section...

“Validating a CVaR Portfolio Set” on page 5-78

“Validating CVaR Portfolios” on page 5-80

In some cases, you may want to validate either your inputs to, or outputs from, a portfolio optimization problem. Although most error checking that occurs during the problem setup phase catches most difficulties with a portfolio optimization problem, the processes to validate CVaR portfolio sets and portfolios are time consuming and are best done offline. Consequently, the portfolio optimization tools have specialized methods to validate CVaR portfolio sets and portfolios. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow”.

Validating a CVaR Portfolio Set

Since it is necessary and sufficient that your CVaR portfolio set must be a nonempty, closed, and bounded set to have a valid portfolio optimization problem, the method `estimateBounds` lets you examine your portfolio set to determine if it is nonempty and, if nonempty, whether it is bounded. Suppose you have the following CVaR portfolio set which is an empty set because the initial portfolio at 0 is too far from a portfolio that satisfies the budget and turnover constraint:

```
p = PortfolioCVaR('NumAssets', 3, 'Budget', 1);
p = setTurnover(p, 0.3, 0);
```

If a CVaR portfolio set is empty, `estimateBounds` returns NaN bounds and sets the `isbounded` flag to []:

```
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
NaN
NaN
NaN
```

```
ub =
```

```

NaN
NaN
NaN

isbounded =

[]

```

Suppose you create an unbounded CVaR portfolio set as follows:

```

p = PortfolioCVaR('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

```

```

lb =

-Inf
-Inf

ub =

1.0e-008 *
-0.3712
Inf

```

```

isbounded =

0

```

In this case, `estimateBounds` returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Finally, suppose you created a CVaR portfolio set that is both nonempty and bounded. `estimateBounds` not only validates the set, but also obtains tighter bounds which is useful if you are concerned with the actual range of portfolio choices for individual assets in your portfolio set:

```

p = PortfolioCVaR;
p = setBudget(p, 1,1);
p = setBounds(p, [ -0.1; 0.2; 0.3; 0.2 ], [ 0.5; 0.3; 0.9; 0.8 ]);

[lb, ub, isbounded] = estimateBounds(p)

lb =

-0.1000

```

```
0.2000
0.3000
0.2000

ub =

0.3000
0.3000
0.7000
0.6000

isbounded =

1
```

In this example, all but the second asset have tighter upper bounds than the input upper bound implies.

Validating CVaR Portfolios

Given a CVaR portfolio set specified in a `PortfolioCVaR` object, you often want to check if specific portfolios are feasible with respect to the portfolio set. This can occur with, for example, initial portfolios and with portfolios obtained from other procedures. The `checkFeasibility` method determines whether a collection of portfolios is feasible. Suppose you perform the following portfolio optimization and want to determine if the resultant efficient portfolios are feasible relative to a modified problem.

First, set up a problem in the `PortfolioCVaR` object `p`, estimate efficient portfolios in `pwgt`, and then confirm that these portfolios are feasible relative to the initial problem:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```

p = setProbabilityLevel(p, 0.95);
pwgt = estimateFrontier(p);
checkFeasibility(p, pwgt)

ans =

     1     1     1     1     1     1     1     1     1     1

```

Next, set up a different portfolio problem that starts with the initial problem with an additional a turnover constraint and an equally weighted initial portfolio:

```

q = setTurnover(p, 0.3, 0.25);
checkFeasibility(q, pwgt)

ans =

     0     0     0     1     1     0     0     0     0     0

```

In this case, only two of the ten efficient portfolios from the initial problem are feasible relative to the new problem in PortfolioCVaR object `q`. Solving the second problem using `checkFeasibility` demonstrates that the efficient portfolio for PortfolioCVaR object `q` is feasible relative to the initial problem:

```

qwgt = estimateFrontier(q);
checkFeasibility(p, qwgt)

ans =

     1     1     1     1     1     1     1     1     1     1

```

See Also

PortfolioCVaR | checkFeasibility | estimateBounds

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56
- “Estimate Efficient Portfolios” on page 5-83
- “Estimate Efficient Frontiers” on page 5-95
- “Asset Returns and Scenarios” on page 5-41

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Estimate Efficient Portfolios

In this section...

“Obtaining Portfolios Along the Entire Efficient Frontier” on page 5-83

“Obtaining Endpoints of the Efficient Frontier” on page 5-85

“Obtaining Efficient Portfolios for Target Returns” on page 5-87

“Obtaining Efficient Portfolios for Target Risks” on page 5-89

“Choosing and Controlling the Solver” on page 5-92

There are two ways to look at a portfolio optimization problem that depends on what you are trying to do. One goal is to estimate efficient portfolios and the other is to estimate efficient frontiers. This section focuses on the former goal and “Estimate Efficient Frontiers” on page 5-95 focuses on the latter goal. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow”.

Obtaining Portfolios Along the Entire Efficient Frontier

The most basic way to obtain optimal portfolios is to obtain points over the entire range of the efficient frontier. Given a portfolio optimization problem in a PortfolioCVaR object, the `estimateFrontier` method computes efficient portfolios spaced evenly according to the return proxy from the minimum to maximum return efficient portfolios. The number of portfolios estimated is controlled by the hidden property `defaultNumPorts` which is set to 10. A different value for the number of portfolios estimated is specified as input to `estimateFrontier`. This example shows the default number of efficient portfolios over the entire range of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);
disp(pwgt);

Columns 1 through 8
```

```

0.8670    0.7046    0.5421    0.3825    0.2236    0.0570    0.0000    0.0000
0.0413    0.1193    0.1963    0.2667    0.3392    0.4159    0.3392    0.1753
0.0488    0.0640    0.0811    0.1012    0.1169    0.1427    0.1568    0.1754
0.0429    0.1120    0.1806    0.2496    0.3203    0.3844    0.5040    0.6493

```

Columns 9 through 10

```

0.0000    0.0000
0.0230    0.0000
0.1777    0.0000
0.7993    1.0000

```

If you want only four portfolios in the previous example:

```
pwgt = estimateFrontier(p, 4);
```

```
disp(pwgt);
```

```

0.8670    0.3825    0.0000    0.0000
0.0413    0.2667    0.3392    0.0000
0.0488    0.1012    0.1568    0.0000
0.0429    0.2496    0.5040    1.0000

```

Starting from the initial portfolio, `estimateFrontier` also returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

```

```

display(pwgt);
display(pbuy);
display(psell);

```

pwgt =

Columns 1 through 8

```

0.8670    0.7046    0.5421    0.3825    0.2236    0.0570    0.0000    0.0000
0.0413    0.1193    0.1963    0.2667    0.3392    0.4159    0.3392    0.1753
0.0488    0.0640    0.0811    0.1012    0.1169    0.1427    0.1568    0.1754
0.0429    0.1120    0.1806    0.2496    0.3203    0.3844    0.5040    0.6493

```

Columns 9 through 10

```

0.0000    0.0000
0.0230    0.0000
0.1777    0.0000
0.7993    1.0000

```

pbuy =

Columns 1 through 8

```

0.5670    0.4046    0.2421    0.0825    0          0          0          0

```

```

0      0      0      0      0.0392  0.1159  0.0392  0
0      0      0      0      0      0      0      0
0      0.0120  0.0806  0.1496  0.2203  0.2844  0.4040  0.5493

```

Columns 9 through 10

```

0      0
0      0
0      0
0.6993  0.9000

```

psell =

Columns 1 through 8

```

0      0      0      0      0.0764  0.2430  0.3000  0.3000
0.2587  0.1807  0.1037  0.0333  0      0      0      0.1247
0.1512  0.1360  0.1189  0.0988  0.0831  0.0573  0.0432  0.0246
0.0571  0      0      0      0      0      0      0

```

Columns 9 through 10

```

0.3000  0.3000
0.2770  0.3000
0.0223  0.2000
0      0

```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Endpoints of the Efficient Frontier

In many cases, you might be interested in the endpoint portfolios for the efficient frontier. Suppose you want to determine the range of returns from minimum to maximum to refine a search for a portfolio with a specific target return. Use the `estimateFrontierLimits` method to obtain the endpoint portfolios:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

```

```

pwgt = estimateFrontierLimits(p);

disp(pwgt);

0.8646    0.0000
0.0470    0.0000
0.0414    0.0000
0.0470    1.0000

```

Note: The endpoints of the efficient frontier depend upon the Scenarios in the PortfolioCVaR object. If you change the Scenarios, you are likely to obtain different endpoints.

Starting from an initial portfolio, `estimateFrontierLimits` also returns purchases and sales to get from the initial portfolio to the endpoint portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierLimits(p);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

```

```

0.8624  0.0000
0.0513  0.0000
0.0452  0.0000
0.0411  1.0000

```

pbuy =

```

0.5624      0
      0      0
      0      0
      0  0.9000

```

psell =

```

      0  0.3000
0.2487  0.3000
0.1548  0.2000
0.0589      0

```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Efficient Portfolios for Target Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` method accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. For example, assume that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 12%:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVar;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

```

```
p = setProbabilityLevel(p, 0.95);  
  
pwgt = estimateFrontierByReturn(p, [0.07, 0.10, .12]);  
display(pwgt);  
  
pwgt =  
  
    0.7526    0.3773    0.1306  
    0.1047    0.3079    0.4348  
    0.0662    0.1097    0.1426  
    0.0765    0.2051    0.2920
```

In some cases, you can request a return for which no efficient portfolio exists. Based on the previous example, suppose you want a portfolio with a 4% return (which is the return of the first asset). A portfolio that is fully invested in the first asset, however, is inefficient. `estimateFrontierByReturn` warns if your target returns are outside the range of efficient portfolio returns and replaces it with the endpoint portfolio of the efficient frontier closest to your target return:

```
pwgt = estimateFrontierByReturn(p, [0.04]);  
  
Warning: One or more target return values are outside the feasible range [  
0.066388, 0.178834 ].  
Will return portfolios associated with endpoints of the range for these values.  
> In PortfolioCVaR.estimateFrontierByReturn at 93
```

The best way to avoid this situation is to bracket your target portfolio returns with `estimateFrontierLimits` and `estimatePortReturn` (see “Obtaining Endpoints of the Efficient Frontier” on page 5-85 and “Obtaining CVaR Portfolio Risks and Returns” on page 5-95).

```
pret = estimatePortReturn(p, p.estimateFrontierLimits);  
  
display(pret);  
  
pret =  
  
    0.0664  
    0.1788
```

This result indicates that efficient portfolios have returns that range from 6.5 to 17.8%. Note, your results for these examples may be different due to the random generation of scenarios.

If you have an initial portfolio, `estimateFrontierByReturn` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, to obtain purchases and sales with target returns of 7%, 10%, and 12%:

```

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByReturn(p, [0.07, 0.10, .12]);

```

```

display(pwgt);
display(pbuy);
display(psell);

```

```
pwgt =
```

0.7526	0.3773	0.1306
0.1047	0.3079	0.4348
0.0662	0.1097	0.1426
0.0765	0.2051	0.2920

```
pbuy =
```

0.4526	0.0773	0
0	0.0079	0.1348
0	0	0
0	0.1051	0.1920

```
psell =
```

0	0	0.1694
0.1953	0	0
0.1338	0.0903	0.0574
0.0235	0	0

If you do not have an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Efficient Portfolios for Target Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` method accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Suppose you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;

```

```
0 0.0119 0.0336 0.1225 ];  
  
AssetScenarios = mvnrnd(m, C, 20000);  
  
p = PortfolioCVaR;  
p = setScenarios(p, AssetScenarios);  
p = setDefaultConstraints(p);  
p = setProbabilityLevel(p, 0.9);  
  
pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);  
  
display(pwgt);  
  
pwgt =  
  
    0.3594    0.2524    0.1543  
    0.3164    0.3721    0.4248  
    0.1044    0.1193    0.1298  
    0.2199    0.2563    0.2910
```

In some cases, you can request a risk for which no efficient portfolio exists. Based on the previous example, suppose you want a portfolio with 6% risk (individual assets in this universe have risks ranging from 7 to 42.5%). It turns out that a portfolio with 6% risk cannot be formed with these four assets. `estimateFrontierByRisk` warns if your target risks are outside the range of efficient portfolio risks and replaces it with the endpoint of the efficient frontier closest to your target risk:

```
pwgt = estimateFrontierByRisk(p, 0.06)  
  
Warning: One or more target risk values are outside the feasible range [  
0.0735749, 0.436667 ].  
Will return portfolios associated with endpoints of the range for these values.  
> In PortfolioCVaR.estimateFrontierByRisk at 80  
  
pwgt =  
  
    0.7899  
    0.0856  
    0.0545  
    0.0700
```

The best way to avoid this situation is to bracket your target portfolio risks with `estimateFrontierLimits` and `estimatePortRisk` (see “Obtaining Endpoints of the Efficient Frontier” on page 5-85 and “Obtaining CVaR Portfolio Risks and Returns” on page 5-95).

```
prsk = estimatePortRisk(p, p.estimateFrontierLimits);
```



```
display(prsk);
```

```
prsk =
    0.0736
    0.4367
```

This result indicates that efficient portfolios have risks that range from 7 to 42.5%. Note, your results for these examples may be different due to the random generation of scenarios.

Starting with an initial portfolio, `estimateFrontierByRisk` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the example with target risks of 12%, 14%, and 16%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);
```

```
display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =
    0.3594    0.2524    0.1543
    0.3164    0.3721    0.4248
    0.1044    0.1193    0.1298
    0.2199    0.2563    0.2910
```

```
pbuy =
    0.0594         0         0
    0.0164    0.0721    0.1248
         0         0         0
    0.1199    0.1563    0.1910
```

```
psell =
         0    0.0476    0.1457
         0         0         0
    0.0956    0.0807    0.0702
```

0 0 0

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Choosing and Controlling the Solver

When solving portfolio optimizations for a `PortfolioCVaR` object, all variations of `fmincon` from Optimization Toolbox are supported. Alternatively, you can use `'cuttingplane'`, a solver that implements Kelley's cutting plane method (see Kelley [45] at "Portfolio Optimization" on page A-10).

Unlike Optimization Toolbox which uses the `interior-point` algorithm as the default algorithm for `fmincon`, the portfolio optimization for a `PortfolioCVaR` object uses the `sqp` algorithm. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see "Constrained Nonlinear Optimization Algorithms".

To modify `fmincon` options for CVaR portfolio optimizations, use `setSolver` to set the hidden properties `solverType` and `solverOptions` to specify and control the solver. (Note that you can see the default options by creating a dummy `PortfolioCVaR` object, using `p = PortfolioCVaR` and then type `p.solverOptions`.) Since these solver properties are hidden, you cannot set them using the `PortfolioCVaR` constructor. The default solver is `fmincon` with the `sqp` algorithm objective function, gradients turned on, and no displayed output, so you do not need to use `setSolver` to specify this.

If you want to specify additional options associated with the `fmincon` solver, `setSolver` accepts these options as name-value pair arguments. For example, if you want to use `fmincon` with the `trust-region-reflective` algorithm and with no displayed output, use `setSolver` with:

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'on');
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
on
```

Alternatively, `setSolver` accepts an `optimoptions` object as the second argument. For example, you can change the algorithm to `trust-region-reflective` with no displayed output as follows:

```
p = PortfolioCVaR;
options = optimoptions('fmincon','Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
```

```
display(p.solverOptions.Display);
trust-region-reflective
off
```

The `'cuttingplane'` solver has options to control the number iterations and stopping tolerances. Moreover, this solver uses `linprog` as the master solver, and all `linprog` options are supported using `optimoptions` structures. All of these options are set using `setSolver`.

For example, you can use `setSolver` to increase the number of iterations for `'cuttingplane'`:

```
p = PortfolioCVaR;
p = setSolver(p, 'cuttingplane', 'MaxIter', 2000);
display(p.solverType);
display(p.solverOptions);

cuttingplane
           MaxIter: 2000
           AbsTol: 1.0000e-06
           RelTol: 1.0000e-05
MasterSolverOptions: [1x1 optim.options.Linprog]
```

To change the master solver algorithm to `'interior-point'`, with no display, use `setSolver` to modify `'MasterSolverOptions'`:

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','interior-point','Display','off');
p = setSolver(p,'cuttingplane','MasterSolverOptions',options);
display(p.solverType)
display(p.solverOptions)
display(p.solverOptions.MasterSolverOptions.Algorithm)
display(p.solverOptions.MasterSolverOptions.Display)

cuttingplane
           MaxIter: 1000
           AbsTol: 1.0000e-06
           RelTol: 1.0000e-05
MasterSolverOptions: [1x1 optim.options.Linprog]

interior-point
off
```

See Also

[PortfolioCVaR](#) | [estimateFrontier](#) | [estimateFrontierByReturn](#)
| [estimateFrontierByRisk](#) | [estimateFrontierByRisk](#) |

`estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56
- “Estimate Efficient Frontiers” on page 5-95
- “Asset Returns and Scenarios” on page 5-41
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-108

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Estimate Efficient Frontiers

In this section...

“Obtaining CVaR Portfolio Risks and Returns” on page 5-95

“Obtaining Portfolio Standard Deviation and VaR” on page 5-97

“Plotting the Efficient Frontier” on page 5-98

Whereas “Estimate Efficient Portfolios” on page 5-83 focused on estimation of efficient portfolios, this section focuses on the estimation of efficient frontiers. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow”.

Obtaining CVaR Portfolio Risks and Returns

Given any portfolio and, in particular, efficient portfolios, the methods `estimatePortReturn` and `estimatePortRisk` provide estimates for the return (or return proxy), risk (or the risk proxy). Each method has the same input syntax but with different combinations of outputs. Suppose you have this following portfolio optimization problem that gave you a collection of portfolios along the efficient frontier in `pwgt`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = setInitPort(p, pwgt0);
pwgt = estimateFrontier(p);
```

Note: Remember that the risk proxy for CVaR portfolio optimization is CVaR.

Given `pwgt0` and `pwgt`, use the portfolio risk and return estimation methods to obtain risks and returns for your initial portfolio and the portfolios on the efficient frontier:

```
prsk0 = estimatePortRisk(p, pwgt0);  
pret0 = estimatePortReturn(p, pwgt0);  
prsk = estimatePortRisk(p, pwgt);  
pret = estimatePortReturn(p, pwgt);  
You obtain these risks and returns:
```

```
display(prsk0);  
display(pret0);  
display(prsk);  
display(pret);
```

```
prsk0 =  
  
    0.0591
```

```
pret0 =  
  
    0.0067
```

```
prsk =  
  
    0.0414  
    0.0453  
    0.0553  
    0.0689  
    0.0843  
    0.1006  
    0.1193  
    0.1426  
    0.1689  
    0.1969
```

```
pret =  
  
    0.0050  
    0.0060  
    0.0070  
    0.0080  
    0.0089
```

```

0.0099
0.0109
0.0119
0.0129
0.0139

```

Obtaining Portfolio Standard Deviation and VaR

The PortfolioCVaR object has methods to compute standard deviations of portfolio returns and the value-at-risk of portfolios with the methods `estimatePortStd` and `estimatePortVaR`. The methods work with any portfolios, not necessarily efficient portfolios. For example, the following example obtains five portfolios (`pwgt`) on the efficient frontier and also has an initial portfolio in `pwgt0`. Various portfolio statistics are computed that include the return, risk, standard deviation, and value-at-risk. The listed estimates are for the initial portfolio in the first row followed by estimates for each of the five efficient portfolios in subsequent rows.

```

m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
      0.00034 0.002408 0.0017 0.000992;
      0.00016 0.0017 0.0048 0.0028;
      0 0.000992 0.0028 0.010208 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR('initport', pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

pwgt = estimateFrontier(p, 5);

pret = estimatePortReturn(p, [pwgt0, pwgt]);
prsk = estimatePortRisk(p, [pwgt0, pwgt]);
pstd = estimatePortStd(p, [pwgt0, pwgt]);
pvar = estimatePortVaR(p, [pwgt0, pwgt]);

[pret, prsk, pstd, pvar]

ans =

    0.0207    0.0464    0.0381    0.0283
    0.1009    0.0214    0.0699   -0.0109

```

```
0.1133    0.0217    0.0772   -0.0137
0.1256    0.0226    0.0849   -0.0164
0.1380    0.0240    0.0928   -0.0182
0.1503    0.0262    0.1011   -0.0197
```

Plotting the Efficient Frontier

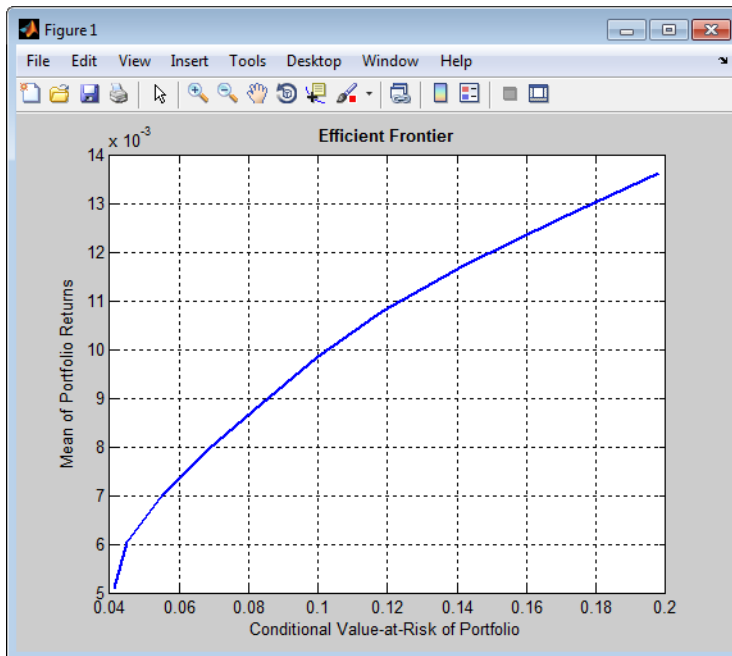
The `plotFrontier` method creates a plot of the efficient frontier for a given portfolio optimization problem. This method accepts several types of inputs and generates a plot with an optional possibility to output the estimates for portfolio risks and returns along the efficient frontier. `plotFrontier` has four different ways that it can be used. In addition to a plot of the efficient frontier, if you have an initial portfolio in the `InitPort` property, `plotFrontier` also displays the return versus risk of the initial portfolio on the same plot. If you have a well-posed portfolio optimization problem set up in a `PortfolioCVaR` object and you use `plotFrontier`, you will get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is currently 10 and is maintained in the hidden property `defaultNumPorts`). This example illustrates a typical use of `plotFrontier` to create a new plot:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```

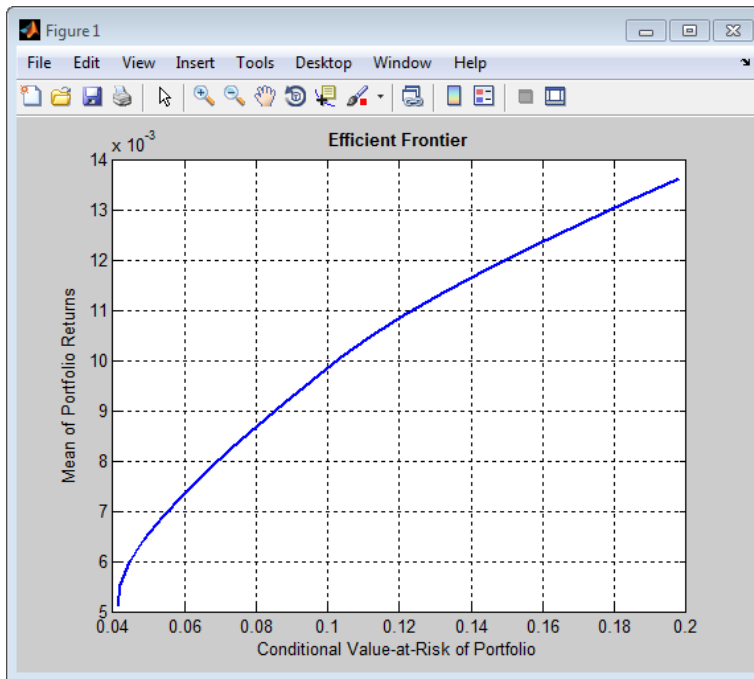
The `Name` property appears as the title of the efficient frontier plot if you set it in the `PortfolioCVaR` object. Without an explicit name, the title on the plot would be “Efficient Frontier.” If you want to obtain a specific number of portfolios along the efficient frontier, use `plotFrontier` with the number of portfolios that you want. Suppose you have the `PortfolioCVaR` object from the previous example and you want to plot 20 portfolios along the efficient frontier and to obtain 20 risk and return values for each portfolio:

```
[prsk, pret] = plotFrontier(p, 20);
display([pret, prsk]);
```

```
ans =
```

```
0.0051    0.0406
0.0056    0.0414
0.0061    0.0437
0.0066    0.0471
0.0071    0.0515
0.0076    0.0567
0.0082    0.0624
0.0087    0.0687
```

0.0092	0.0753
0.0097	0.0821
0.0102	0.0891
0.0107	0.0962
0.0112	0.1044
0.0117	0.1142
0.0122	0.1251
0.0127	0.1369
0.0133	0.1496
0.0138	0.1628
0.0143	0.1766
0.0148	0.1907



Plotting Existing Efficient Portfolios

If you already have efficient portfolios from any of the "estimateFrontier" methods (see "Estimate Efficient Portfolios" on page 5-83), pass them into plotFrontier directly to plot the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
```

```

C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

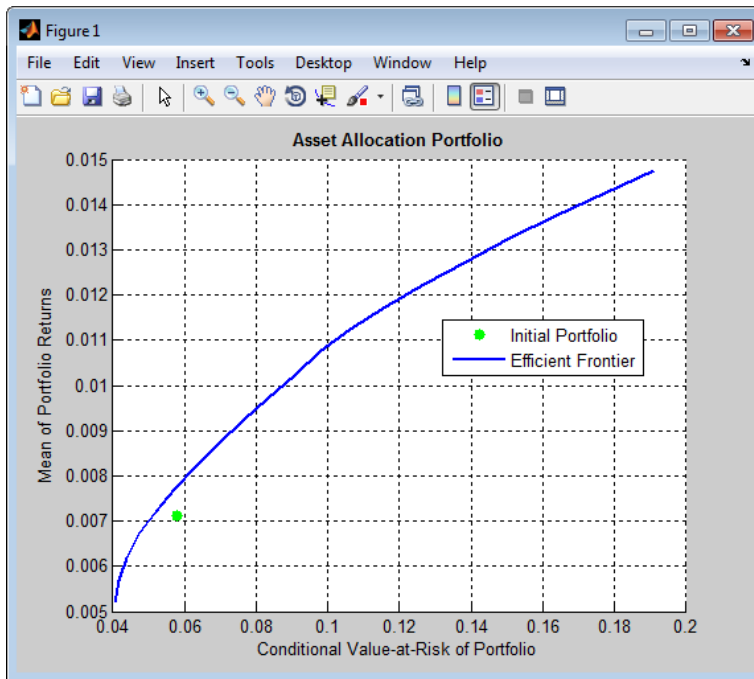
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p, 20);
plotFrontier(p, pwgt);

```



Plotting Existing Efficient Portfolio Risks and Returns

If you already have efficient portfolio risks and returns, you can use the interface to `plotFrontier` to pass them into `plotFrontier` to obtain a plot of the efficient frontier:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

AssetScenarios = mvnrnd(m, C, 20000);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

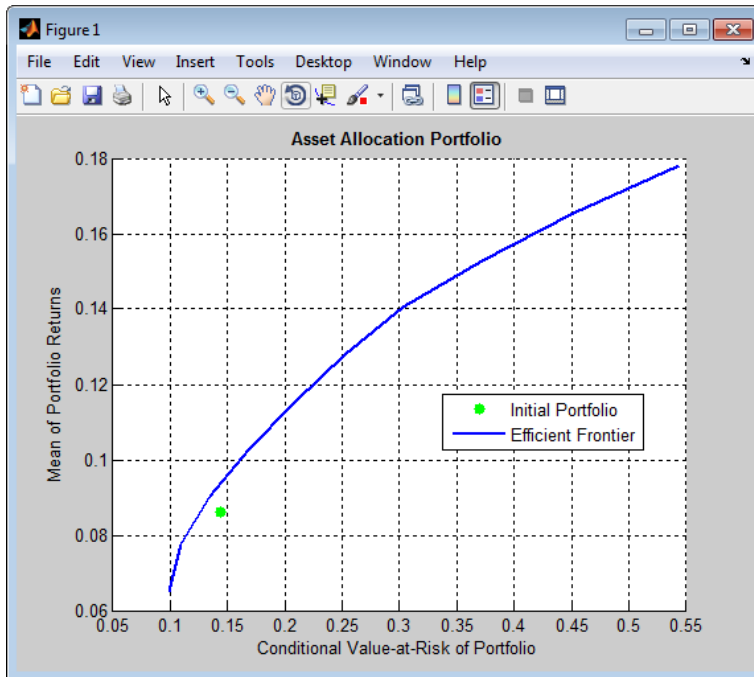
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);

pret= estimatePortReturn(p, pwgt)
prsk = estimatePortRisk(p, pwgt)

plotFrontier(p, prsk, pret);

```



See Also

PortfolioCVaR | estimatePortReturn | estimatePortStd | estimatePortVaR | plotFrontier

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56
- “Asset Returns and Scenarios” on page 5-41
- “Estimate Efficient Portfolios” on page 5-83
- “Postprocessing Results” on page 5-104

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

Postprocessing Results

In this section...

“Setting Up Tradable Portfolios” on page 5-104

“Working with Other Portfolio Objects” on page 5-105

“Troubleshooting CVaR Portfolio Optimization Results” on page 5-108

After obtaining efficient portfolios or estimates for expected portfolio risks and returns, use your results to set up trades to move toward an efficient portfolio. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow”.

Setting Up Tradable Portfolios

Suppose you set up a portfolio optimization problem and obtained portfolios on the efficient frontier. Use the `dataset` object from Statistics Toolbox to form a blotter that lists your portfolios with the names for each asset. For example, suppose you want to obtain five portfolios along the efficient frontier. You can set up a blotter with weights multiplied by 100 to view the allocations for each portfolio:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR;
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([100*pwgt],pnames,'obsnames',p.AssetList);
display(Blotter);

Blotter =
```

	Port1	Port2	Port3	Port4	Port5
Bonds	78.84	43.688	8.3448	0	1.2501e-12

Large-Cap Equities	9.3338	29.131	48.467	23.602	9.4219e-13
Small-Cap Equities	4.8843	8.1284	12.419	16.357	8.281e-14
Emerging Equities	6.9419	19.053	30.769	60.041	100

Note: Your results may differ from this result due to the simulation of scenarios.

This result indicates that you would invest primarily in bonds at the minimum-risk/minimum-return end of the efficient frontier (**Port1**), and that you would invest completely in emerging equity at the maximum-risk/maximum-return end of the efficient frontier (**Port5**). You can also select a particular efficient portfolio, for example, suppose you want a portfolio with 15% risk and you add purchase and sale weights outputs obtained from the “estimateFrontier” methods to set up a trade blotter:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');

p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

[pwgt, pbuy, psell] = estimateFrontierByRisk(p, 0.15);

Blotter = dataset([100*[pwgt0, pwgt, pbuy, psell]], ...
{'Initial','Weight', 'Purchases','Sales'},'obsnames',p.AssetList);
display(Blotter);

Blotter =
```

	Initial	Weight	Purchases	Sales
Bonds	30	15.036	0	14.964
Large-Cap Equities	30	45.357	15.357	0
Small-Cap Equities	20	12.102	0	7.8982
Emerging Equities	10	27.505	17.505	0

If you have prices for each asset (in this example, they can be ETFs), add them to your blotter and then use the tools of the `dataset` object to obtain shares and shares to be traded.

Working with Other Portfolio Objects

The `PortfolioCVaR` object is for CVaR portfolio optimization. The `Portfolio` object is for mean-variance portfolio optimization. In some cases, you might want to examine portfolio

optimization problems according to different combinations of return and risk proxies. A common example is that you want to do a CVaR portfolio optimization and then want to work primarily with moments of portfolio returns. Suppose you set up a CVaR portfolio optimization problem with:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR;
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);
```

To work with the same problem in a mean-variance framework, you can use the scenarios from the PortfolioCVaR object to set up a Portfolio object so that `p` contains a CVaR optimization problem and `q` contains a mean-variance optimization problem based on the same data.

```
q = Portfolio('AssetList', p.AssetList);
q = estimateAssetMoments(q, p.getScenarios);
q = setDefaultConstraints(q);

pwgt = estimateFrontier(p);
qwgt = estimateFrontier(q);
```

Note that since each object has a different risk proxy, it is not possible to compare results side by side. To obtain means and standard deviations of portfolio returns, you can use the methods associated with each object to obtain:

```
pret = estimatePortReturn(p, pwgt);
pstd = estimatePortStd(p, pwgt);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);

[pret, qret]
[pstd, qstd]

ans =

    0.0665    0.0585
    0.0787    0.0716
```



```
0.0910    0.0848
0.1033    0.0979
0.1155    0.1111
0.1278    0.1243
0.1401    0.1374
0.1523    0.1506
0.1646    0.1637
0.1769    0.1769
```

```
ans =
```

```
0.0797    0.0774
0.0912    0.0835
0.1095    0.0995
0.1317    0.1217
0.1563    0.1472
0.1823    0.1746
0.2135    0.2059
0.2534    0.2472
0.2985    0.2951
0.3499    0.3499
```

To produce comparable results, you can use the returns or risks from one portfolio optimization as target returns or risks for the other portfolio optimization.

```
qwgt = estimateFrontierByReturn(q, pret);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);
```

```
[pret, qret]
[pstd, qstd]
```

```
ans =
```

```
0.0665    0.0665
0.0787    0.0787
0.0910    0.0910
0.1033    0.1033
0.1155    0.1155
0.1278    0.1278
0.1401    0.1401
0.1523    0.1523
0.1646    0.1646
0.1769    0.1769
```

```
ans =  
  
    0.0797    0.0797  
    0.0912    0.0912  
    0.1095    0.1095  
    0.1317    0.1317  
    0.1563    0.1563  
    0.1823    0.1823  
    0.2135    0.2135  
    0.2534    0.2534  
    0.2985    0.2985  
    0.3499    0.3499
```

Now it is possible to compare standard deviations of portfolio returns from either type of portfolio optimization.

Troubleshooting CVaR Portfolio Optimization Results

PortfolioCVaR Object Destroyed When Modifying

If a PortfolioCVaR object is destroyed when modifying, remember to pass an existing object into the constructor if you want to modify it, otherwise it creates a new object. See “Constructing the PortfolioCVaR Object” on page 5-26 for details.

Matrix Incompatibility and "Non-Conformable" Errors

If you get matrix incompatibility or "non-conformable" errors, the representation of data in the tools follows a specific set of basic rules described in “Conventions for Representation of Data” on page 5-24.

CVaR Portfolio Optimization Warns About “Max Iterations”

If the 'cuttingplane' solver displays the following warning:

```
Warning: Max iterations reached. Consider modifying the solver options, or using fmincon.  
> In @PortfolioCVaR\private\cvar_cuttingplane_solver at 255  
    In @PortfolioCVaR\private\cvar_optim_min_risk at 85  
    In PortfolioCVaR.estimateFrontier at 69
```

this warning indicates that some of the reported efficient portfolios may not be accurate enough.

This warning is usually related to portfolios in the lower-left end of the efficient frontier. The cutting plane solver may have gotten very close to the solution, but there may be too

many portfolios with very similar risks and returns in that neighborhood, and the solver runs out of iterations before reaching the desired accuracy.

To correct this problem, you can use `setSolver` to make any of these changes:

- Increase the maximum number of iterations ('MaxIter').
- Relax the stopping tolerances ('AbsTol' and/or 'RelTol').
- Use a different master solver algorithm ('MasterSolverOptions').
- Alternatively, you can try the 'fmincon' solver.

When the default maximum number of iterations of the 'cuttingplane' solver is reached, the solver usually needs many more iterations to reach the accuracy required by the default stopping tolerances. You may want to combine increasing the number of iterations (e.g., multiply by 5) with relaxing the stopping tolerances (e.g., multiply by 10 or 100). Since the CVaR is a stochastic optimization problem, the accuracy of the solution is relative to the scenario sample, so a looser stopping tolerance may be acceptable. Keep in mind that the solution time may increase significantly when you increase the number of iterations. For example, doubling the number of iterations more than doubles the solution time. Sometimes using a different master solver (e.g., switching to 'interior-point' if you are using the default 'simplex') can get the 'cuttingplane' solver to converge without changing the maximum number of iterations.

Alternatively, the 'fmincon' solver may be faster than the 'cuttingplane' solver for problems where cutting plane reaches the maximum number of iterations.

CVaR Portfolio Optimization Errors with “Could Not Solve” Message

If the 'cuttingplane' solver generates the following error:

```
Error using cvar_cuttingplane_solver (line 251)
Could not solve the problem. Consider modifying the solver options, or using fmincon.
```

```
Error in cvar_optim_by_return (line 100)
[X,-,exitflag] = cvar_cuttingplane_solver(...
```

```
Error in PortfolioCVaR/estimateFrontier (line 80)
pwgt = cvar_optim_by_return(obj, r(2:end-1), obj.NumAssets, ...
```

this error means the master solver failed to solve one of the master problems. The error may be due to numerical instability or other problem-specific situation.

To correct this problem, you can use `setSolver` to make any of these changes:

- Modify the master solver options ('MasterSolverOptions'), for example, change the algorithm ('Algorithm') or the termination tolerance ('TolFun').

- Alternatively, you can try the 'fmincon' solver.

Missing Data Estimation Fails

If asset return data has missing or NaN values, the method `simulateNormalScenariosByData` with the 'missingdata' flag set to `true` may fail with either too many iterations or a singular covariance. To correct this problem, consider this:

- If you have asset return data with no missing or NaN values, you can compute a covariance matrix that may be singular without difficulties. If you have missing or NaN values in your data, the supported missing data feature requires that your covariance matrix must be positive-definite, i.e., nonsingular.
- `simulateNormalScenariosByData` uses default settings for the missing data estimation procedure that might not be appropriate for all problems.

In either case, you might want to estimate the moments of asset returns separately with either the ECM estimation functions such as `ecmmle` or with your own methods.

cvar_optim_transform Errors

If you obtain optimization errors such as:

```
Error using cvar_optim_transform (line 276)
Portfolio set appears to be either empty or unbounded. Check constraints.
```

```
Error in PortfolioCVaR/estimateFrontier (line 64)
[AI, bI, AE, bE, lB, uB, f0, f, x0] = cvar_optim_transform(obj);
```

or

```
Error using cvar_optim_transform (line 281)
Cannot obtain finite lower bounds for specified portfolio set.
```

```
Error in PortfolioCVaR/estimateFrontier (line 64)
```

```
[AI, bI, AE, bE, lB, uB, f0, f, x0] = cvar_optim_transform(obj);
```

Since the portfolio optimization tools require a bounded portfolio set, these errors (and similar errors) can occur if your portfolio set is either empty and, if nonempty, unbounded. Specifically, the portfolio optimization algorithm requires that your portfolio set have at least a finite lower bound. The best way to deal with these problems is to use the validation methods in “Validate the CVaR Portfolio Problem” on page 5-78. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

If you obtain efficient portfolios that do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between 0 and 1 and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the PortfolioCVaR object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

See Also

PortfolioCVaR | `checkFeasibility` | `estimateScenarioMoments`

Related Examples

- “Constructing the PortfolioCVaR Object” on page 5-26
- “Working with CVaR Portfolio Constraints” on page 5-56
- “Asset Returns and Scenarios” on page 5-41
- “Estimate Efficient Portfolios” on page 5-83
- “Estimate Efficient Frontiers” on page 5-95

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object Workflow”

MAD Portfolio Optimization Tools

- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-16
- “PortfolioMAD Object” on page 6-17
- “Constructing the PortfolioMAD Object” on page 6-26
- “Common Operations on the PortfolioMAD Object” on page 6-34
- “Asset Returns and Scenarios” on page 6-41
- “Working with MAD Portfolio Constraints” on page 6-56
- “Validate the MAD Portfolio Problem” on page 6-78
- “Estimate Efficient Portfolios” on page 6-83
- “Estimate Efficient Frontiers” on page 6-94
- “Postprocessing Results” on page 6-104

Portfolio Optimization Theory

In this section...
“Portfolio Optimization Problems” on page 6-2
“Portfolio Problem Specification” on page 6-2
“Return Proxy” on page 6-3
“Risk Proxy” on page 6-5
“Portfolio Set for Portfolio Optimization” on page 6-7
“Default Portfolio Problem” on page 6-14

Portfolio Optimization Problems

Portfolio optimization problems involve identifying portfolios that satisfy three criteria:

- Minimize a proxy for risk.
- Match or exceed a proxy for return.
- Satisfy basic feasibility requirements.

Portfolios are points from a feasible set of assets that constitute an asset universe. A portfolio specifies either holdings or weights in each individual asset in the asset universe. The convention is to specify portfolios in terms of weights, although the portfolio optimization tools work with holdings as well.

The set of feasible portfolios is necessarily a nonempty, closed, and bounded set. The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices. The proxy for return is a function that characterizes either the gross or net benefits associated with portfolio choices. The terms “risk” and “risk proxy” and “return” and “return proxy” are interchangeable. The fundamental insight of Markowitz (see “Portfolio Optimization” on page A-10) is that the goal of the portfolio choice problem is to seek minimum risk for a given level of return and to seek maximum return for a given level of risk. Portfolios satisfying these criteria are efficient portfolios and the graph of the risks and returns of these portfolios forms a curve called the efficient frontier.

Portfolio Problem Specification

To specify a portfolio optimization problem, you need the following:

- Proxy for portfolio return (μ)
- Proxy for portfolio risk (Σ)
- Set of feasible portfolios (X), called a portfolio set

Financial Toolbox has three objects to solve specific types of portfolio optimization problems:

- The Portfolio object (`Portfolio`) supports mean-variance portfolio optimization (see Markowitz [46], [47] at “Portfolio Optimization” on page A-10). This object has either gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set.
- The PortfolioCVaR object (`PortfolioCVaR`) implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-10), which is generally referred to as CVaR portfolio optimization. CVaR portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses conditional value-at-risk of portfolio returns as the risk proxy.
- The PortfolioMAD object (`PortfolioMAD`) implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10), which is generally referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

The proxy for portfolio return is a function $\mu : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the rewards associated with portfolio choices. In most cases, the proxy for portfolio return has two general forms, gross and net portfolio returns. Both portfolio return forms separate the risk-free rate r_0 so that the portfolio $x \in X$ contains only risky assets.

Regardless of the underlying distribution of asset returns, a collection of S asset returns y_1, \dots, y_S has a mean of asset returns

$$m = \frac{1}{S} \sum_{s=1}^S y_s,$$

and (sample) covariance of asset returns

$$C = \frac{1}{S-1} \sum_{s=1}^S (y_s - m)(y_s - m)^T.$$

These moments (or alternative estimators that characterize these moments) are used directly in mean-variance portfolio optimization to form proxies for portfolio risk and return.

Gross Portfolio Returns

The gross portfolio return for a portfolio $x \in X$ is

$$\mu(x) = r_0 + (m - r_0 \mathbf{1})^T x,$$

where:

r_0 is the risk-free rate (scalar).

m is the mean of asset returns (n vector).

If the portfolio weights sum to 1, the risk-free rate is irrelevant. The properties in the Portfolio object to specify gross portfolio returns are:

- RiskFreeRate for r_0
- AssetMean for m

Net Portfolio Returns

The net portfolio return for a portfolio $x \in X$ is

$$\mu(x) = r_0 + (m - r_0 \mathbf{1})^T x - b^T \max\{0, x - x_0\} - s^T \max\{0, x_0 - x\},$$

where:

r_0 is the risk-free rate (scalar).

m is the mean of asset returns (n vector).

b is the proportional cost to purchase assets (n vector).

s is the proportional cost to sell assets (n vector).

You can incorporate fixed transaction costs in this model also. Though in this case, it is necessary to incorporate prices into such costs. The properties in the Portfolio object to specify net portfolio returns are:

- `RiskFreeRate` for r_0
- `AssetMean` for m
- `InitPort` for x_0
- `BuyCost` for b
- `SellCost` for s

Risk Proxy

The proxy for portfolio risk is a function $\Sigma : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the risks associated with portfolio choices.

Variance

The variance of portfolio returns for a portfolio $x \in X$ is

$$\Sigma(x) = x^T C x$$

where C is the covariance of asset returns (n -by- n positive-semidefinite matrix).

The property in the Portfolio object to specify the variance of portfolio returns is `AssetCovar` for C .

Although the risk proxy in mean-variance portfolio optimization is the variance of portfolio returns, the square root, which is the standard deviation of portfolio returns, is often reported and displayed. Moreover, this quantity is often called the “risk” of the portfolio. For details, see Markowitz [46], [47] at (“Portfolio Optimization” on page A-10).

Conditional Value-at-Risk

The conditional value-at-risk for a portfolio $x \in X$, which is also known as expected shortfall, is defined as

$$CVaR_{\alpha}(x) = \frac{1}{1-\alpha} \int_{f(x,y) \geq VaR_{\alpha}(x)} f(x,y)p(y)dy,$$

where:

α is the probability level such that $0 < \alpha < 1$.

$f(x,y)$ is the loss function for a portfolio x and asset return y .

$p(y)$ is the probability density function for asset return y .

VaR_{α} is the value-at-risk of portfolio x at probability level α .

The value-at-risk is defined as

$$VaR_{\alpha}(x) = \min\{\gamma : \Pr[f(x,Y) \leq \gamma] \geq \alpha\}.$$

An alternative formulation for CVaR has the form:

$$CVaR_{\alpha}(x) = VaR_{\alpha}(x) + \frac{1}{1-\alpha} \int_{R^n} \max\{0, (f(x,y) - VaR_{\alpha}(x))\} p(y) dy$$

The choice for the probability level α is typically 0.9 or 0.95. Choosing α implies that the value-at-risk $VaR_{\alpha}(x)$ for portfolio x is the portfolio return such that the probability of portfolio returns falling below this level is $(1 - \alpha)$. Given $VaR_{\alpha}(x)$ for a portfolio x , the conditional value-at-risk of the portfolio is the expected loss of portfolio returns above the value-at-risk return.

Note: Value-at-risk is a positive value for losses so that the probability level α indicates the probability that portfolio returns are below the negative of the value-at-risk.

The risk proxy for CVaR portfolio optimization is $CVaR_{\alpha}(x)$ for a given portfolio $x \in X$ and $\alpha \in (0,1)$. Note that the value-at-risk, or VaR, for a given probability level is estimated whenever CVaR is estimated.

In addition, keep in mind that VaR and CVaR are sample estimators for VaR and CVaR based on the given scenarios. Better scenario samples yield more reliable estimates of VaR and CVaR.

For more information, see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-10.

Mean Absolute-Deviation

The mean-absolute deviation (MAD) for a portfolio $x \in X$ is defined as

$$\sum(x) = \frac{1}{S} \sum_{s=1}^S |(y_s - m)^T x|$$

where:

y_s are asset returns with scenarios $s = 1, \dots, S$ (S collection of n vectors).

$f(x, y)$ is the loss function for a portfolio x and asset return y .

m is the mean of asset returns (n vector).

such that

$$m = \frac{1}{S} \sum_{s=1}^S y_s$$

For more information, see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10.

Portfolio Set for Portfolio Optimization

The final element for a complete specification of a portfolio optimization problem is the set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset \mathcal{R}^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or “default” portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to 1 (using the budget constraint). The most general portfolio set handled by the portfolio optimization tools can have any of these constraints:

- Linear inequality constraints
- Linear equality constraints
- Bound constraints
- Budget constraints
- Group constraints
- Group ratio constraints
- Average turnover constraints
- One-way turnover constraints

Linear Inequality Constraints

Linear inequality constraints are general linear constraints that model relationships among portfolio weights that satisfy a system of inequalities. Linear inequality constraints take the form

$$A_I x \leq b_I$$

where:

x is the portfolio (n vector).

A_I is the linear inequality constraint matrix (n_I -by- n matrix).

b_I is the linear inequality constraint vector (n_I vector).

n is the number of assets in the universe and n_I is the number of constraints.

Portfolio object properties to specify linear inequality constraints are:

- `AInequality` for A_I
- `bInequality` for b_I
- `NumAssets` for n

The default is to ignore these constraints.

Linear Equality Constraints

Linear equality constraints are general linear constraints that model relationships among portfolio weights that satisfy a system of equalities. Linear equality constraints take the form

$$A_E x = b_E$$

where:

x is the portfolio (n vector).

A_E is the linear equality constraint matrix (n_E -by- n matrix).

b_E is the linear equality constraint vector (n_E vector).

n is the number of assets in the universe and n_E is the number of constraints.

Portfolio object properties to specify linear equality constraints are:

- `AEquality` for A_E
- `bEquality` for b_E
- `NumAssets` for n

The default is to ignore these constraints.

Bound Constraints

Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Since every portfolio set must be bounded, it is often a good practice, albeit not necessary, to set explicit bounds for the portfolio problem. To obtain explicit bounds for a given portfolio set, use the method `estimateBounds`. Bound constraints take the form

$$l_B \leq x \leq u_B$$

where:

x is the portfolio (n vector).

l_B is the lower-bound constraint (n vector).

u_B is the upper-bound constraint (n vector).

n is the number of assets in the universe.

Portfolio object properties to specify bound constraints are:

- `LowerBound` for l_B
- `UpperBound` for u_B
- `NumAssets` for n

The default is to ignore these constraints.

Note that the default portfolio optimization problem (see “Default Portfolio Problem” on page 6-14) has $l_B = 0$ with u_B set implicitly through a budget constraint.

Budget Constraints

Budget constraints are specialized linear constraints that confine the sum of portfolio weights to fall either above or below specific bounds. The constraints take the form

$$l_S \leq \mathbf{1}^T x \leq u_S$$

where:

x is the portfolio (n vector).

$\mathbf{1}$ is the vector of ones (n vector).

l_S is the lower-bound budget constraint (scalar).

u_S is the upper-bound budget constraint (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify budget constraints are:

- `LowerBudget` for l_S
- `UpperBudget` for u_S

- `NumAssets` for n

The default is to ignore this constraint.

The default portfolio optimization problem (see “Default Portfolio Problem” on page 6-14) has $l_S = u_S = 1$, which means that the portfolio weights sum to 1. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0 to 100% invested in cash. If cash is to be a portfolio choice, set `RiskFreeRate` (r_0) to a suitable value (see “Return Proxy” on page 6-3 and “Working with a Riskless Asset” on page 6-51).

Group Constraints

Group constraints are specialized linear constraints that enforce “membership” among groups of assets. The constraints take the form

$$l_G \leq Gx \leq u_G$$

where:

x is the portfolio (n vector).

l_G is the lower-bound group constraint (n_G vector).

u_G is the upper-bound group constraint (n_G vector).

G is the matrix of group membership indexes (n_G -by- n matrix).

Each row of G identifies which assets belong to a group associated with that row. Each row contains either 0s or 1s with 1 indicating that an asset is part of the group or 0 indicating that the asset is not part of the group.

Portfolio object properties to specify group constraints are:

- `GroupMatrix` for G
- `LowerGroup` for l_G
- `UpperGroup` for u_G
- `NumAssets` for n

The default is to ignore these constraints.

Group Ratio Constraints

Group ratio constraints are specialized linear constraints that enforce relationships among groups of assets. The constraints take the form

$$l_{Ri} \cdot (G_B x)_i \leq (G_A x)_i \leq u_{Ri} \cdot (G_B x)_i$$

for $i = 1, \dots, n_R$ where:

x is the portfolio (n vector).

l_R is the vector of lower-bound group ratio constraints (n_R vector).

u_R is the vector matrix of upper-bound group ratio constraints (n_R vector).

G_A is the matrix of base group membership indexes (n_R -by- n matrix).

G_B is the matrix of comparison group membership indexes (n_R -by- n matrix).

n is the number of assets in the universe and n_R is the number of constraints.

Each row of G_A and G_B identifies which assets belong to a base and comparison group associated with that row.

Each row contains either 0s or 1s with 1 indicating that an asset is part of the group or 0 indicating that the asset is not part of the group.

Portfolio object properties to specify group ratio constraints are:

- `GroupA` for G_A
- `GroupB` for G_B
- `LowerRatio` for l_R
- `UpperRatio` for u_R
- `NumAssets` for n

The default is to ignore these constraints.

Average Turnover Constraints

Turnover constraint is a linear absolute value constraint that ensures estimated optimal portfolios differ from an initial portfolio by no more than a specified amount. Although

portfolio turnover is defined in many ways, the turnover constraints implemented in Financial Toolbox computes portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

$$\frac{1}{2} \mathbf{1}^T |x - x_0| \leq \tau$$

where:

x is the portfolio (n vector).

$\mathbf{1}$ is the vector of ones (n vector).

x_0 is the initial portfolio (n vector).

τ is the upper bound for turnover (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

- `Turnover` for τ
- `InitPort` for x_0
- `NumAssets` for n

The default is to ignore this constraint.

One-Way Turnover Constraints

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the forms

$$\mathbf{1}^T \times \max\{0, x - x_0\} \leq \tau_B$$

$$\mathbf{1}^T \times \max\{0, x_0 - x\} \leq \tau_S$$

where:

x is the portfolio (n vector)

$\mathbf{1}$ is the vector of ones (n vector).

x_θ is the Initial portfolio (n vector).

τ_B is the upper bound for turnover constraint on purchases (scalar).

τ_S is the upper bound for turnover constraint on sales (scalar).

To specify one-way turnover constraints, use the following properties in the Portfolio, PortfolioCVaR, or PotfolioMAD object:

- `BuyTurnover` for τ_B
- `SellTurnover` for τ_S
- `InitPort` for x_θ

The default is to ignore this constraint.

Note: The average turnover constraint (see “Working with Average Turnover Constraints” on page 6-72) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

Default Portfolio Problem

The default portfolio optimization problem has a risk and return proxy associated with a given problem, and a portfolio set that specifies portfolio weights to be nonnegative and to sum to 1. The lower bound combined with the budget constraint is sufficient to ensure that the portfolio set is nonempty, closed, and bounded. The default portfolio optimization problem characterizes a long-only investor who is fully invested in a collection of assets.

- For mean-variance portfolio optimization, it is sufficient to set up the default problem. After setting up the problem, data in the form of a mean and covariance of asset returns are then used to solve portfolio optimization problems.
- For conditional value-at-risk portfolio optimization, the default problem requires the additional specification of a probability level that must be set explicitly. Generally, “typical” values for this level are 0.90 or 0.95. After setting up the problem, data in

the form of scenarios of asset returns are then used to solve portfolio optimization problems.

- For MAD portfolio optimization, it is sufficient to set up the default problem. After setting up the problem, data in the form of scenarios of asset returns are then used to solve portfolio optimization problems.

See Also

[AbstractPortfolio](#) | [Portfolio](#) | [PortfolioCVaR](#) | [PortfolioMAD](#)

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56

More About

- “PortfolioMAD Object” on page 6-17
- “PortfolioMAD Object Workflow”

PortfolioMAD Object Workflow

The PortfolioMAD object workflow for creating and modeling a mean-variance portfolio is:

- 1** Create a MAD Portfolio.

Create a `PortfolioMAD` object for mean-absolute deviation (MAD) portfolio optimization. “Constructing the PortfolioMAD Object”.

- 2** Define asset returns and scenarios.

Evaluate scenarios for portfolio asset returns, including assets with missing data and financial time series data. For more information, see “Asset Returns and Scenarios”.

- 3** Specify the MAD Portfolio Constraints.

Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, and turnover constraints. For more information, see “Working with MAD Portfolio Constraints”.

- 4** Validate the MAD Portfolio.

Identify errors for the portfolio specification. For more information, see “Validate the MAD Portfolio Problem”.

- 5** Estimate the efficient portfolios and frontiers.

Analyze the efficient portfolios and efficient frontiers for a portfolio. For more information, see “Estimate Efficient Portfolios” and “Estimate Efficient Frontiers”.

- 6** Postprocess the results.

Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see “Postprocessing Results”.

PortfolioMAD Object

In this section...

“PortfolioMAD Object Properties and Methods” on page 6-17

“Working with PortfolioMAD Objects” on page 6-21

“Setting and Getting Properties” on page 6-22

“Displaying PortfolioMAD Objects” on page 6-22

“Saving and Loading PortfolioMAD Objects” on page 6-23

“Estimating Efficient Portfolios and Frontiers” on page 6-23

“Arrays of PortfolioMAD Objects” on page 6-23

“Subclassing PortfolioMAD Objects” on page 6-24

“Conventions for Representation of Data” on page 6-24

PortfolioMAD Object Properties and Methods

The PortfolioMAD object implements mean absolute-deviation (MAD) portfolio optimization and is derived from the abstract class `AbstractPortfolio`. Every property and method of the PortfolioMAD object is public, although some properties and methods are hidden. See PortfolioMAD Object Properties and PortfolioMAD Object Methods for the properties and methods of a PortfolioMAD object. The PortfolioMAD object is a value object where every instance of the object is a distinct version of the object. Since the PortfolioMAD object is also a MATLAB object, it inherits the default methods associated with MATLAB objects.

PortfolioMAD Object Properties

Property	Description	Characteristics
<code>AEquality</code>	Matrix for equality constraints	Matrix
<code>AInequality</code>	Matrix for inequality constraints	Matrix
<code>AssetList</code>	List of asset names or identifiers	Vector cell array of strings
<code>bEquality</code>	Vector for equality constraints	Vector
<code>bInequality</code>	Vector for inequality constraints	Vector
<code>BuyCost</code>	Cost to purchase assets	Vector

Property	Description	Characteristics
BuyTurnover	Turnover constraint on purchases	Scalar
GroupA	Base group ratio constraint matrix	Boolean matrix or matrix
GroupB	Comparison group ratio constraint matrix	Boolean matrix or matrix
GroupMatrix	Group membership matrix	Boolean matrix or matrix
InitPort	Initial or current portfolio	Vector
LowerBound	Lower-bound constraint	Vector
LowerBudget	Lower-budget constraint	Scalar
LowerGroup	Lower-group constraint	Vector
LowerRatio	Lower-group ratio constraint ratio	Vector
Name	Name for instance of PortfolioMAD object	String
NumAssets	Number of assets in universe	Scalar positive integer
NumScenarios	Number of scenarios (read only)	Scalar integer
RiskFreeRate	Period return of riskless asset	Scalar
SellCost	Cost to sell assets	Vector
SellTurnover	Turnover constraint on sales	Scalar
Turnover	Upper-bound portfolio turnover	Scalar
UpperBound	Upper-bound constraint	Vector
UpperBudget	Upper-budget constraint	Scalar
UpperGroup	Upper-group constraint	Vector
UpperRatio	Upper-group ratio constraint ratio	Vector

PortfolioMAD Object Methods

Method	Description
addEquality	Add equality constraints for portfolio weights to existing constraints.
addGroupRatio	Add group ratio constraints for portfolio weights to existing constraints.

Method	Description
addGroups	Add group constraints for portfolio weights to existing constraints.
addInequality	Add inequality constraints for portfolio weights to existing constraints.
checkFeasibility	Determine if portfolios are members of the set of feasible portfolios.
estimateBounds	Determine if the set of feasible portfolios is nonempty and bounded.
estimateFrontier	Estimate portfolios on the entire efficient frontier.
estimateFrontierByReturn	Estimate portfolios on the efficient frontier with targeted returns or return proxies.
estimateFrontierByRisk	Estimate portfolios on the efficient frontier with targeted risks or risk proxies.
estimateFrontierLimits	Estimate portfolios at the extreme ends of the efficient frontier (minimum risk and maximum return).
estimatePortReturn	Estimate the return or return proxy for specified portfolios.
estimatePortRisk	Estimate the risk or risk proxy for specified portfolios.
estimatePortStd	Estimate standard deviation of portfolio returns.
estimateScenarioMoments	Estimate mean and covariance of scenarios.
getBounds	Get lower and upper bounds from object.
simulateNormalScenariosByData	Simulate multivariate normal asset return scenarios from data.
simulateNormalScenariosByMoments	Simulate multivariate normal asset return scenarios from a mean and covariance of asset returns.
getBudget	Get lower and upper budget constraints from object.

Method	Description
getCosts	Get purchase and sales proportional transaction costs from object.
getEquality	Get equality constraint matrix and vector from object.
getGroupRatio	Get base matrix, comparison matrix, and lower and upper bounds for group ratio constraints from object.
getGroups	Get group matrix and lower and upper bounds for group constraints from object.
getInequality	Get inequality constraint matrix and vector from object.
getOneWayTurnover	Get one-way portfolio turnover constraints.
getScenarios	Obtain scenarios from PortfolioMAD object.
plotFrontier	Plot efficient frontier and optionally obtain risks and returns for portfolios on the efficient frontier.
setAssetList	Set up a list of asset names and symbols to be associated with assets in universe.
setBounds	Set up lower and upper bounds for portfolio weights.
setBudget	Set up lower and upper budget constraints for portfolio weights.
setCosts	Set up purchase and sale proportional transaction costs for assets in universe.
setDefaultConstraints	Set up default constraints for portfolio weights (nonnegative weights that must sum to 1).
setEquality	Set up equality constraints for portfolio weights.
setGroupRatio	Set up group ratio constraints for portfolio weights.
setGroups	Set up group constraints for portfolio weights.

Method	Description
<code>setInequality</code>	Set up inequality constraints for portfolio weights.
<code>setInitPort</code>	Set up initial portfolio weights.
<code>setOneWayTurnover</code>	Set up one-way portfolio turnover constraints.
<code>setOptions</code>	Set up hidden control properties in object (not implemented).
<code>setScenarios</code>	Set asset returns scenarios by direct matrix.
<code>setSolver</code>	Set up solver to estimate efficient portfolios.
<code>setTurnover</code>	Set up average turnover constraints for portfolio weights.

Working with PortfolioMAD Objects

The PortfolioMAD object and its methods are an interface for mean absolute-deviation portfolio optimization. Consequently, almost everything you do with the PortfolioMAD object can be done using the methods. The basic workflow is:

- 1 Design your portfolio problem.
- 2 Use the portfolio constructor (`PortfolioMAD`) to create the PortfolioMAD object or use the various set methods to set up your portfolio problem.
- 3 Use estimate methods to solve your portfolio problem.

In addition, methods are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a PortfolioMAD object, you can save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of MAD portfolio optimization, means that you have either scenarios, data, or moments for asset returns, and a collection of constraints on your portfolios, use the portfolio constructor to set the properties for the PortfolioMAD object.

The `PortfolioMAD` constructor lets you create an object from scratch or update an existing object. Since the PortfolioMAD object is a value object, it is easy to create a basic object, then use methods to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see “Constructing the PortfolioMAD Object” on page 6-26.

Setting and Getting Properties

You can set properties of a PortfolioMAD object using either the constructor (PortfolioMAD) or various set methods.

Note: Although you can also set properties directly, it is not recommended since error-checking is not performed when you set a property directly.

The PortfolioMAD constructor supports setting properties with name-value pair arguments such that each argument name is a property and each value is the value to assign to that property. For example, to set the LowerBound and Budget properties in an existing PortfolioMAD object `p`, use the syntax:

```
p = PortfolioMAD(p, 'LowerBound', 0, 'Budget', 1);
```

In addition to the PortfolioMAD constructor, which lets you set individual properties one at a time, groups of properties are set in a PortfolioMAD object with various “set” and “add” methods. For example, to set up an average turnover constraint, use the `setTurnover` method to specify the bound on portfolio turnover and the initial portfolio. To get individual properties from a PortfolioMAD object, obtain properties directly or use an assortment of “get” methods that obtain groups of properties from a PortfolioMAD object. The PortfolioMAD object constructor and set methods have several useful features:

- The constructor and set methods try to determine the dimensions of your problem with either explicit or implicit inputs.
- The constructor and set methods try to resolve ambiguities with default choices.
- The constructor and set methods perform scalar expansion on arrays when possible.
- The methods try to diagnose and warn about problems.

Displaying PortfolioMAD Objects

The PortfolioMAD object uses the default display method provided by MATLAB, where `display` and `disp` display a PortfolioMAD object and its properties with or without the object variable name.

Saving and Loading PortfolioMAD Objects

Save and load PortfolioMAD objects using the MATLAB `save` and `load` commands.

Estimating Efficient Portfolios and Frontiers

Estimating efficient portfolios and efficient frontiers is the primary purpose of the MAD portfolio optimization tools. A collection of “estimate” and “plot” methods provide ways to explore the efficient frontier. The “estimate” methods obtain either efficient portfolios or risk and return proxies to form efficient frontiers. At the portfolio level, a collection of methods estimates efficient portfolios on the efficient frontier with methods to obtain efficient portfolios:

- At the endpoints of the efficient frontier
- That attain targeted values for return proxies
- That attain targeted values for risk proxies
- Along the entire efficient frontier

These methods also provide purchases and sales needed to shift from an initial or current portfolio to each efficient portfolio. At the efficient frontier level, a collection of methods plot the efficient frontier and estimate either risk or return proxies for efficient portfolios on the efficient frontier. You can use the resultant efficient portfolios or risk and return proxies in subsequent analyses.

Arrays of PortfolioMAD Objects

Although all methods associated with a PortfolioMAD object are designed to work on a scalar PortfolioMAD object, the array capabilities of MATLAB enables you to set up and work with arrays of PortfolioMAD objects. The easiest way to do this is with the `repmat` function. For example, to create a 3-by-2 array of PortfolioMAD objects:

```
p = repmat(PortfolioMAD, 3, 2);
disp(p)
```

After setting up an array of PortfolioMAD objects, you can work on individual PortfolioMAD objects in the array by indexing. For example:

```
p(i,j) = PortfolioMAD(p(i,j), ... );
```

This example calls the PortfolioMAD object constructor for the (i,j) element of a matrix of PortfolioMAD objects in the variable `p`.

If you set up an array of PortfolioMAD objects, you can access properties of a particular PortfolioMAD object in the array by indexing so that you can set the lower and upper bounds `lb` and `ub` for the `(i,j,k)` element of a 3-D array of PortfolioMAD objects with

```
p(i,j,k) = setBounds(p(i,j,k),lb, ub);  
and, once set, you can access these bounds with
```

```
[lb, ub] = getBounds(p(i,j,k));  
PortfolioMAD object methods work on only one PortfolioMAD object at a time.
```

Subclassing PortfolioMAD Objects

You can subclass the PortfolioMAD object to override existing methods or to add new properties or methods. To do so, create a derived class from the `PortfolioMAD` class. This gives you all the properties and methods of the `PortfolioMAD` class along with any new features that you choose to add to your subclassed object. The `PortfolioMAD` class is derived from an abstract class called `AbstractPortfolio`. Because of this, you can also create a derived class from `AbstractPortfolio` that implements an entirely different form of portfolio optimization using properties and methods of the `AbstractPortfolio` class.

Conventions for Representation of Data

The MAD portfolio optimization tools follow these conventions regarding the representation of different quantities associated with portfolio optimization:

- Asset returns or prices for scenarios are in matrix form with samples for a given asset going down the rows and assets going across the columns. In the case of prices, the earliest dates must be at the top of the matrix, with increasing dates going down.
- Portfolios are in vector or matrix form with weights for a given portfolio going down the rows and distinct portfolios going across the columns.
- Constraints on portfolios are formed in such a way that a portfolio is a column vector.
- Portfolio risks and returns are either scalars or column vectors (for multiple portfolio risks and returns).

See Also

`AbstractPortfolio` | `PortfolioMAD`

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56

More About

- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Constructing the PortfolioMAD Object

In this section...
“Syntax” on page 6-26
“PortfolioMAD Problem Sufficiency” on page 6-27
“Constructor Examples” on page 6-27

To create a fully specified MAD portfolio optimization problem, instantiate the PortfolioMAD object using the MAD portfolio constructor. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

Syntax

Use the MAD portfolio constructor `PortfolioMAD` to create an instance of an object of the `PortfolioMAD` class. You can use this constructor in several ways. To set up a portfolio optimization problem in a PortfolioMAD object, the simplest syntax is:

```
p = PortfolioMAD;
```

This syntax creates a PortfolioMAD object, `p`, such that all object properties are empty.

The constructor also accepts collections of argument name-value pair arguments for properties and their values. The constructor accepts inputs for public properties (see PortfolioMAD Object Properties) with the general syntax:

```
p = PortfolioMAD('property1', value1, 'property2', value2, ... );
```

If a PortfolioMAD object already exists, the syntax permits the first (and only the first argument) of the MAD portfolio constructor to be an existing object with subsequent argument name-value pair arguments for properties to be added or modified. For example, given an existing PortfolioMAD object in `p`, the general syntax is:

```
p = PortfolioMAD(p, 'property1', value1, 'property2', value2, ... );
```

Input argument names are not case sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 6-31). The `PortfolioMAD` constructor tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process

to formulate a problem. In addition, a PortfolioMAD object is a value object so that, given portfolio `p`, the following code creates two objects, `p` and `q`, that are distinct:

```
q = PortfolioMAD(p, ...)
```

PortfolioMAD Problem Sufficiency

A MAD portfolio optimization problem is completely specified with the PortfolioMAD object if the following three conditions are met:

- You must specify a collection of asset returns or prices known as scenarios such that all scenarios are finite asset returns or prices. These scenarios are meant to be samples from the underlying probability distribution of asset returns. This condition can be satisfied by the `setScenarios` method or with several canned scenario simulation methods.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded. You can satisfy this condition using an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed and several tools, such as the `estimateBounds` method, provide ways to ensure that your problem is properly formulated.

Although the general sufficient conditions for MAD portfolio optimization go beyond these conditions, the PortfolioMAD object handles all these additional conditions.

Constructor Examples

If you create a PortfolioMAD object, `p`, with no input arguments, you can display it using `disp`:

```
p = PortfolioMAD;  
disp(p);
```

PortfolioMAD with properties:

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []  
Turnover: []  
BuyTurnover: []  
SellTurnover: []
```

```
NumScenarios: []
  Name: []
  NumAssets: []
  AssetList: []
  InitPort: []
AInequality: []
bInequality: []
  AEquality: []
  bEquality: []
LowerBound: []
UpperBound: []
LowerBudget: []
UpperBudget: []
GroupMatrix: []
  LowerGroup: []
  UpperGroup: []
    GroupA: []
    GroupB: []
LowerRatio: []
UpperRatio: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the MAD portfolio constructor. The custom set methods offer additional ways to set and modify collections of properties in the PortfolioMAD object.

Using the Constructor for a Single-Step Setup

You can use the PortfolioMAD constructor to directly set up a “standard” portfolio optimization problem. Given scenarios of asset returns in the variable AssetScenarios, this problem is completely specified as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

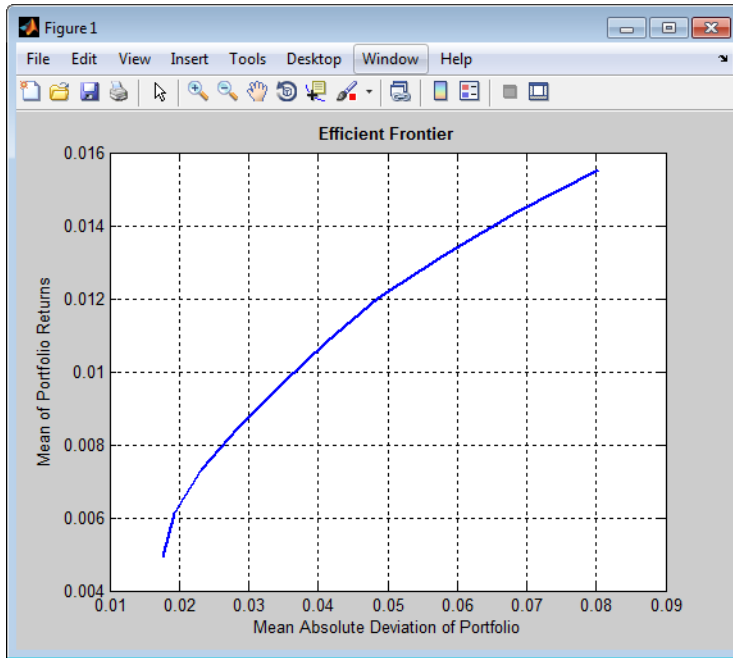
AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('Scenarios', AssetScenarios, ...
  'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1);
```

Note that the LowerBound property value undergoes scalar expansion since AssetScenarios provides the dimensions of the problem.

You can use dot notation with the method plotFrontier.

```
plotFrontier(p);
```



Using the Constructor with a Sequence of Steps

An alternative way to accomplish the same task of setting up a “standard” MAD portfolio optimization problem, given `AssetScenarios` variable is:

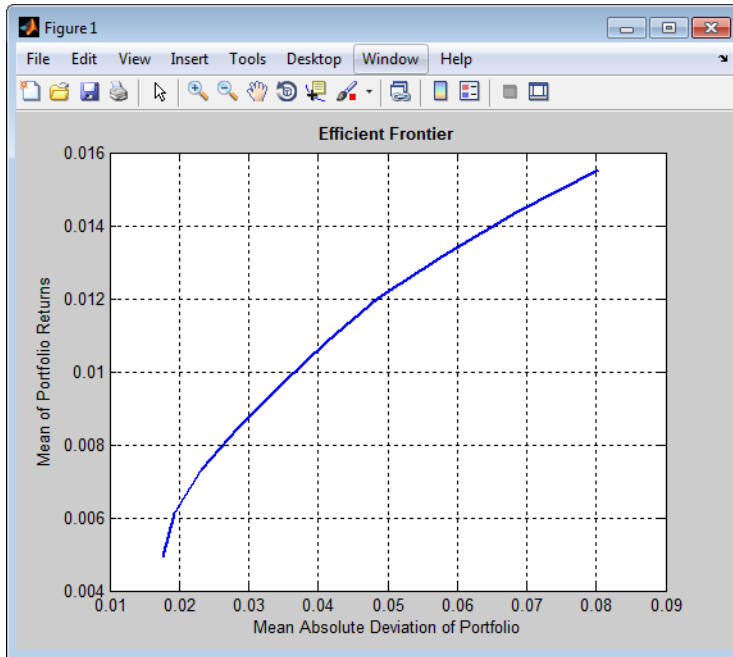
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = PortfolioMAD(p, 'LowerBound', 0);
```

```
p = PortfolioMAD(p, 'LowerBudget', 1, 'UpperBudget', 1);
plotFrontier(p);
```



This way works because the calls to the constructor are in this particular order. In this case, the call to initialize `AssetScenarios` provides the dimensions for the problem. If you were to do this step last, you would have to explicitly dimension the `LowerBound` property as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```

p = PortfolioMAD;
p = PortfolioMAD(p, 'LowerBound', zeros(size(m)));
p = PortfolioMAD(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setScenarios(p, AssetScenarios);

```

Note: If you did not specify the size of `LowerBound` but, instead, input a scalar argument, the constructor assumes that you are defining a single-asset problem and produces an error at the call to set asset scenarios with four assets.

Shortcuts for Property Names

The MAD portfolio constructor has shorter argument names that replace longer argument names associated with specific properties of the PortfolioMAD object. For example, rather than enter 'AInequality', the constructor accepts the case-insensitive name 'ai' to set the AInequality property in a PortfolioMAD object. Every shorter argument name corresponds with a single property in the MAD portfolio constructor. The one exception is the alternative argument name 'budget', which signifies both the LowerBudget and UpperBudget properties. When 'budget' is used, then the LowerBudget and UpperBudget properties are set to the same value to form an equality budget constraint.

Shortcut Argument Name	Equivalent Argument / Property Name
ae	AEquality
ai	AInequality
assetnames or assets	AssetList
be	bEquality
bi	bInequality
budget	UpperBudget and LowerBudget
group	GroupMatrix
lb	LowerBound
n or num	NumAssets
rfr	RiskFreeRate
scenario or assetscenarios	Scenarios

Shortcut Argument Name	Equivalent Argument / Property Name
ub	UpperBound

For example, this call to the constructor uses these shortcuts for properties:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('scenario', AssetScenarios, 'lb', 0, 'budget', 1);
plotFrontier(p);
```

Direct Setting of Portfolio Object Properties

Although not recommended, you can set properties directly using dot notation, however no error-checking is done on your inputs:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;

p = setScenarios(p, AssetScenarios);

p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));

plotFrontier(p);
```

Note: Scenarios cannot be assigned directly to a PortfolioMAD object. Scenarios must always be set through either the PortfolioMAD constructor, the setScenarios method, or any of the scenario simulation methods.

See Also

PortfolioMAD | estimateBounds

Related Examples

- “Common Operations on the PortfolioMAD Object” on page 6-34
- “Working with MAD Portfolio Constraints” on page 6-56

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Common Operations on the PortfolioMAD Object

In this section...

“Naming a PortfolioMAD Object” on page 6-34

“Configuring the Assets in the Asset Universe” on page 6-34

“Setting Up a List of Asset Identifiers” on page 6-35

“Truncating and Padding Asset Lists” on page 6-36

“Setting Up an Initial or Current Portfolio” on page 6-37

Naming a PortfolioMAD Object

To name a PortfolioMAD object, use the `Name` property. `Name` is informational and has no effect on any portfolio calculations. If the `Name` property is nonempty, `Name` is the title for the efficient frontier plot generated by `plotFrontier`. For example, if you set up an asset allocation fund, you could name the PortfolioMAD object `Asset Allocation Fund`:

```
p = PortfolioMAD('Name', 'Asset Allocation Fund');  
disp(p.Name);  
Asset Allocation Fund
```

Configuring the Assets in the Asset Universe

The fundamental quantity in the PortfolioMAD object is the number of assets in the asset universe. This quantity is maintained in the `NumAssets` property. Although you can set this property directly, it is usually derived from other properties such as the number of assets in the scenarios or the initial portfolio. In some instances, the number of assets may need to be set directly. This example shows how to set up a PortfolioMAD object that has four assets:

```
p = PortfolioMAD('NumAssets', 4);  
disp(p.NumAssets);  
4
```

After setting the `NumAssets` property, you cannot modify it (unless no other properties are set that depend on `NumAssets`). The only way to change the number of assets in an existing PortfolioMAD object with a known number of assets is to create a new PortfolioMAD object.

Setting Up a List of Asset Identifiers

When working with portfolios, you must specify a universe of assets. Although you can perform a complete analysis without naming the assets in your universe, it is helpful to have an identifier associated with each asset as you create and work with portfolios. You can create a list of asset identifiers as a cell vector of strings in the property `AssetList`. You can set up the list using the next two methods.

Setting Up Asset Lists Using the Constructor

Suppose you have a PortfolioMAD object, `p`, with assets with symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR'. You can create a list of these asset symbols in the object using the constructor:

```
p = PortfolioMAD('assetlist', { 'AA', 'BA', 'CAT', 'DD', 'ETR' });
disp(p.AssetList);
```

```
'AA'    'BA'    'CAT'    'DD'    'ETR'
```

Notice that the property `AssetList` is maintained as a cell array that contains strings, and that it is necessary to pass a cell array into the constructor to set `AssetList`. In addition, notice that the property `NumAssets` is set to 5 based on the number of symbols used to create the asset list:

```
disp(p.NumAssets);
```

```
5
```

Setting Up Asset Lists Using the setAssetList Method

You can also specify a list of assets using the method `setAssetList`. Given the list of asset symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR', you can use `setAssetList` with:

```
p = PortfolioMAD;
p = setAssetList(p, { 'AA', 'BA', 'CAT', 'DD', 'ETR' });
disp(p.AssetList);
```

```
'AA'    'BA'    'CAT'    'DD'    'ETR'
```

`setAssetList` also enables you enter symbols directly as a comma-separated list without creating a cell array of strings. For example, given the list of assets symbols 'AA', 'BA', 'CAT', 'DD', and 'ETR', use `setAssetList`:

```
p = PortfolioMAD;
```

```
p = setAssetList(p, 'AA', 'BA', 'CAT', 'DD', 'ETR');
disp(p.AssetList);

'AA'      'BA'      'CAT'      'DD'      'ETR'
```

`setAssetList` has many additional features to create lists of asset identifiers. If you use `setAssetList` with just a `PortfolioMAD` object, it creates a default asset list according to the name specified in the hidden public property `defaultforAssetList` (which is `'Asset'` by default). The number of asset names created depends on the number of assets in the property `NumAssets`. If `NumAssets` is not set, then `NumAssets` is assumed to be 1.

For example, if a `PortfolioMAD` object `p` is created with `NumAssets = 5`, then this code fragment shows the default naming behavior:

```
p = PortfolioMAD('numassets',5);
p = setAssetList(p);
disp(p.AssetList);

'Asset1'      'Asset2'      'Asset3'      'Asset4'      'Asset5'
```

Suppose that your assets are, for example, ETFs and you change the hidden property `defaultforAssetList` to `'ETF'`, you can then create a default list for ETFs:

```
p = PortfolioMAD('numassets',5);
p.defaultforAssetList = 'ETF';
p = setAssetList(p);
disp(p.AssetList);

'ETF1'      'ETF2'      'ETF3'      'ETF4'      'ETF5'
```

Truncating and Padding Asset Lists

If the `NumAssets` property is already set and you pass in too many or too few identifiers, the `MAD` portfolio constructor, and the `setAssetList` method truncate or pad the list with numbered default asset names that use the name specified in the hidden public property `defaultforAssetList`. If the list is truncated or padded, a warning message indicates the discrepancy. For example, assume you have a `PortfolioMAD` object with five ETFs and you only know the first three CUSIPs `'921937835'`, `'922908769'`, and `'922042775'`. Use this syntax to create an asset list that pads the remaining asset identifiers with numbered `'UnknownCUSIP'` placeholders:

```
p = PortfolioMAD('numassets',5);
p.defaultforAssetList = 'UnknownCUSIP';
```

```
p = setAssetList(p, '921937835', '922908769', '922042775');
disp(p.AssetList);

Warning: Input list of assets has 2 too few identifiers. Padding with numbered
assets.
> In PortfolioMAD.setAssetList at 121
   Columns 1 through 4

       '921937835'    '922908769'    '922042775'    'UnknownCUSIP4'

Column 5

       'UnknownCUSIP5'
```

Alternatively, suppose you have too many identifiers and need only the first four assets. This example illustrates truncation of the asset list using the `PortfolioMAD` constructor:

```
p = PortfolioMAD('numassets',4);
p = PortfolioMAD(p, 'assetlist', { 'AGG', 'EEM', 'MDY', 'SPY', 'VEU' });
disp(p.AssetList);

Warning: AssetList has 1 too many identifiers. Using first 4 assets.
> In PortfolioMAD.checkarguments at 410
   In PortfolioMAD.PortfolioMAD>PortfolioMAD.PortfolioMAD at 187
       'AGG'    'EEM'    'MDY'    'SPY'
```

The hidden public property `uppercaseAssetList` is a Boolean flag to specify whether to convert asset names to uppercase letters. The default value for `uppercaseAssetList` is `false`. This example shows how to use the `uppercaseAssetList` flag to force identifiers to be uppercase letters:

```
p = PortfolioMAD;
p.uppercaseAssetList = true;
p = setAssetList(p, { 'aa', 'ba', 'cat', 'dd', 'etr' });
disp(p.AssetList);

'AA'    'BA'    'CAT'    'DD'    'ETR'
```

Setting Up an Initial or Current Portfolio

In many applications, creating a new optimal portfolio requires comparing the new portfolio with an initial or current portfolio to form lists of purchases and sales. The `PortfolioMAD` object property `InitPort` lets you identify an initial or current portfolio. The initial portfolio also plays an essential role if you have either transaction costs or turnover constraints. The initial portfolio need not be feasible within the constraints of the problem. This can happen if the weights in a portfolio have shifted such that some

constraints become violated. To check if your initial portfolio is feasible, use the method `checkFeasibility` described in “Validating MAD Portfolios” on page 6-80. Suppose you have an initial portfolio in `x0`, then use the `PortfolioMAD` object constructor to set up an initial portfolio:

```
x0 = [ 0.3; 0.2; 0.2; 0.0 ];  
p = PortfolioMAD('InitPort', x0);  
disp(p.InitPort);
```

```
0.3000  
0.2000  
0.2000  
0
```

As with all array properties, you can set `InitPort` with scalar expansion. This is helpful to set up an equally weighted initial portfolio of, for example, 10 assets:

```
p = PortfolioMAD('NumAssets', 10, 'InitPort', 1/10);  
disp(p.InitPort);
```

```
0.1000  
0.1000  
0.1000  
0.1000  
0.1000  
0.1000  
0.1000  
0.1000  
0.1000  
0.1000
```

To clear an initial portfolio from your `PortfolioMAD` object, use either the constructor or the `setInitPort` method with an empty input for the `InitPort` property. If transaction costs or turnover constraints are set, it is not possible to clear the `InitPort` property in this way. In this case, to clear `InitPort`, first clear the dependent properties and then clear the `InitPort` property.

The `InitPort` property can also be set with `setInitPort` which lets you specify the number of assets if you want to use scalar expansion. For example, given an initial portfolio in `x0`, use `setInitPort` to set the `InitPort` property:

```
p = PortfolioMAD;  
x0 = [ 0.3; 0.2; 0.2; 0.0 ];  
p = setInitPort(p, x0);
```

```
disp(p.InitPort);

0.3000
0.2000
0.2000
0
```

To create an equally weighted portfolio of four assets, use `setInitPort`:

```
p = PortfolioMAD;
p = setInitPort(p, 1/4, 4);
disp(p.InitPort);

0.2500
0.2500
0.2500
0.2500
```

PortfolioMAD object methods that work with either transaction costs or turnover constraints also depend on the `InitPort` property. Consequently, the set methods for transaction costs or turnover constraints permit the assignment of a value for the `InitPort` property as part of their implementation. For details, see “Working with Average Turnover Constraints” on page 6-72, “Working with One-Way Turnover Constraints” on page 6-74, and “Working with Transaction Costs” on page 6-51. If either transaction costs or turnover constraints are used, then the `InitPort` property must have a nonempty value. Absent a specific value assigned through the constructor or various set methods, the PortfolioMAD object sets `InitPort` to 0 and warns if `BuyCost`, `SellCost`, or `Turnover` properties are set. This example shows what happens if you specify an average turnover constraint with an initial portfolio:

```
p = PortfolioMAD('Turnover', 0.3, 'InitPort', [ 0.3; 0.2; 0.2; 0.0 ]);
disp(p.InitPort);

0.3000
0.2000
0.2000
0
```

In contrast, this example shows what happens if an average turnover constraint is specified without an initial portfolio:

```
p = PortfolioMAD('Turnover', 0.3);
disp(p.InitPort);
```

```
Warning: InitPort and NumAssets are empty and either transaction costs or
turnover constraints specified. Will set NumAssets = 1 and InitPort = 0.
> In PortfolioMAD.checkarguments at 446
```

```
In PortfolioMAD.PortfolioMAD>PortfolioMAD.PortfolioMAD at 190  
0
```

See Also

PortfolioMAD | checkFeasibility | estimateBounds | setAssetList |
setInitPort

Related Examples

- “Working with MAD Portfolio Constraints” on page 6-56
- “Asset Returns and Scenarios” on page 6-41

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Asset Returns and Scenarios

In this section...

“How Stochastic Optimization Works” on page 6-41
 “What Are Scenarios?” on page 6-42
 “Setting Scenarios Using the PortfolioMAD Constructor” on page 6-42
 “Setting Scenarios Using the setScenarios Method” on page 6-44
 “Estimating the Mean and Covariance of Scenarios” on page 6-44
 “Simulating Normal Scenarios” on page 6-45
 “Simulating Normal Scenarios from Returns or Prices” on page 6-45
 “Simulating Normal Scenarios with Missing Data” on page 6-47
 “Simulating Normal Scenarios from Time Series Data” on page 6-49
 “Simulating Normal Scenarios for Mean and Covariance” on page 6-50
 “Working with a Riskless Asset” on page 6-51
 “Working with Transaction Costs” on page 6-51

How Stochastic Optimization Works

The MAD of a portfolio is mean-absolute deviation. For the definition of the MAD function, see “Risk Proxy” on page 6-5. Although analytic solutions for MAD exist for a few probability distributions, an alternative is to compute the expectation for MAD with samples from the probability distribution of asset returns. These samples are called scenarios and, given a collection of scenarios, the portfolio optimization problem becomes a stochastic optimization problem.

As a function of the portfolio weights, the MAD of the portfolio is a convex non-smooth function (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10). The PortfolioMAD object computes MAD as this nonlinear function which can be handled by the solver `fmincon` Optimization Toolbox. The nonlinear programming solver `fmincon` has several algorithms that can be selected with the `setSolver` method, the two algorithms that work best in practice are `'sqp'` and `'active-set'`.

There are reformulations of the MAD portfolio optimization problem (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-10) that result in a linear programming problem, which can be solved either with standard linear programming

techniques or with stochastic programming solvers. The PortfolioMAD object, however, does not reformulate the problem in such a manner. The PortfolioMAD object computes the MAD as a nonlinear function. The convexity of the MAD, as a function of the portfolio weights and the somewhat dull edges when the number of scenarios is large, make the MAD portfolio optimization problem tractable, in practice, for certain nonlinear programming solvers, such as `fmincon` from Optimization Toolbox. To learn more about the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

What Are Scenarios?

Since mean absolute deviation portfolio optimization works with scenarios of asset returns to perform the optimization, several ways exist to specify and simulate scenarios. In many applications with MAD portfolio optimization, asset returns may have distinctly nonnormal probability distributions with either multiple modes, binning of returns, truncation of distributions, and so forth. In other applications, asset returns are modeled as the result of various simulation methods that might include Monte-Carlo simulation, quasi-random simulation, and so forth. In many cases, the underlying probability distribution for risk factors may be multivariate normal but the resultant transformations are sufficiently nonlinear to result in distinctively nonnormal asset returns.

For example, this occurs with bonds and derivatives. In the case of bonds with a nonzero probability of default, such scenarios would likely include asset returns that are -100% to indicate default and some values slightly greater than -100% to indicate recovery rates.

Although the PortfolioMAD object has methods to simulate multivariate normal scenarios from either data or moments (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`), the usual approach is to specify scenarios directly from your own simulation methods. These scenarios are entered directly as a matrix with a sample for all assets across each row of the matrix and with samples for an asset down each column of the matrix. The architecture of the MAD portfolio optimization tools references the scenarios through a function handle so scenarios that have been set cannot be accessed directly as a property of the PortfolioMAD object.

Setting Scenarios Using the PortfolioMAD Constructor

Suppose you have a matrix of scenarios in the `AssetScenarios` variable. The scenarios are set through the PortfolioMAD constructor with:


```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('Scenarios', AssetScenarios);

disp(p.NumAssets);
disp(p.NumScenarios);

4

20000

```

Notice that the `PortfolioMAD` object determines and fixes the number of assets in `NumAssets` and the number of scenarios in `NumScenarios` based on the scenario's matrix. You can change the number of scenarios by calling the `PortfolioMAD` constructor with a different scenario matrix. However, once the `NumAssets` property has been set in the object, you cannot enter a scenario matrix with a different number of assets. The `getScenarios` method lets you recover scenarios from a `PortfolioMAD` object. You can also obtain the mean and covariance of your scenarios using `estimateScenarioMoments`.

Although not recommended for the casual user, an alternative way exists to recover scenarios by working with the function handle that points to scenarios in the `PortfolioMAD` object. To access some or all of the scenarios from a `PortfolioMAD` object, the hidden property `localScenarioHandle` is a function handle that points to a function to obtain scenarios that have already been set. To get scenarios directly from a `PortfolioMAD` object `p`, use

```

scenarios = p.localScenarioHandle([], []);
and to obtain a subset of scenarios from rows startrow to endrow, use

scenarios = p.localScenarioHandle(startrow, endrow);
where 1 ≤ startrow ≤ endrow ≤ numScenarios.

```

Setting Scenarios Using the setScenarios Method

You can also set scenarios using `setScenarios`. For example, given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);

disp(p.NumAssets);
disp(p.NumScenarios);

4

20000
```

Estimating the Mean and Covariance of Scenarios

The `estimateScenarioMoments` method obtains estimates for the mean and covariance of scenarios in a `PortfolioMAD` object.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
[mean, covar] = estimateScenarioMoments(p)

mean =

    0.0044
    0.0084
    0.0108
    0.0155

covar =

    0.0005    0.0003    0.0002   -0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0047    0.0028
   -0.0000    0.0010    0.0028    0.0103

```

Simulating Normal Scenarios

As a convenience, the two methods (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`) exist to simulate scenarios from data or moments under an assumption that they are distributed as multivariate normal random asset returns.

Simulating Normal Scenarios from Returns or Prices

Given either return or price data, use the method `simulateNormalScenariosByData` to simulate multivariate normal scenarios. Either returns or prices are stored as matrices with samples going down the rows and assets going across the columns. In addition, returns or prices can be stored in a financial time series `fints` object (see “Simulating Normal Scenarios from Time Series Data” on page 6-49). To illustrate using `simulateNormalScenariosByData`, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with `portsim`. The default behavior of `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by `portsim` in the variable `X`, a price series is created in the variable `Y`:

```

m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;

```

```
0.00034 0.002408 0.0017 0.000992;  
0.00016 0.0017 0.0048 0.0028;  
0 0.000992 0.0028 0.010208 ];
```

```
X = portsim(m', C, 120);  
Y = ret2tick(X);
```

Note: Portfolio optimization requires that you use total returns and not just price returns. Consequently, “returns” should be total returns and “prices” should be total return prices.

Given asset returns and prices in variables X and Y from above, this sequence of examples demonstrates equivalent ways to simulate multivariate normal scenarios for the PortfolioMAD object. Assume a PortfolioMAD object created in p that uses the asset returns in X uses `simulateNormalScenariosByData`:

```
p = PortfolioMAD;  
p = simulateNormalScenariosByData(p, X, 20000);  
  
[passetmean, passetcovar] = estimateScenarioMoments(p)  
  
passetmean =  
  
    0.0033  
    0.0085  
    0.0095  
    0.1503  
  
passetcovar =  
  
    0.0055    0.0004    0.0002    0.0001  
    0.0004    0.0024    0.0017    0.0010  
    0.0002    0.0017    0.0049    0.0028  
    0.0001    0.0010    0.0028    0.0102
```

The moments that you obtain from this simulation will likely differ from the moments listed here because the scenarios are random samples from the estimated multivariate normal probability distribution of the input returns X .

The default behavior of `simulateNormalScenariosByData` is to work with asset returns. If, instead, you have asset prices as in the variable Y , `simulateNormalScenariosByData` accepts an argument name `'DataFormat'` with

a corresponding value set to 'prices' to indicate that the input to the method is in the form of asset prices and not returns (the default value for the 'DataFormat' argument is 'returns'). This example simulates scenarios with the asset price data in Y for the PortfolioMAD object q:

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0043
    0.0083
    0.0099
    0.1500

passetcovar =

    0.0053    0.0003    0.0001    0.0002
    0.0003    0.0024    0.0017    0.0010
    0.0001    0.0017    0.0047    0.0027
    0.0002    0.0010    0.0027    0.0100
```

Simulating Normal Scenarios with Missing Data

Often when working with multiple assets, you have missing data indicated by NaN values in your return or price data. Although “Multivariate Normal Regression” on page 9-2 goes into detail about regression with missing data, the `simulateNormalScenariosByData` method has an argument name 'MissingData' that indicates with a Boolean value whether to use the missing data capabilities of Financial Toolbox. The default value for 'MissingData' is `false` which removes all samples with NaN values. If, however, 'MissingData' is set to `true`, `simulateNormalScenariosByData` uses the ECM algorithm to estimate asset moments. This example shows how this works on price data with missing values:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
      0.00034 0.002408 0.0017 0.000992;
      0.00016 0.0017 0.0048 0.0028;
      0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);
```

```
Y = ret2tick(X);
Y(1:20,1) = NaN;
Y(1:12,4) = NaN;
```

Notice that the prices above in `Y` have missing values in the first and fourth series.

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

q = PortfolioMAD;
q = simulateNormalScenariosByData(q, Y, 20000, 'dataformat', 'prices', 'missingdata', true);

[passetmean, passetcovar] = estimateScenarioMoments(p)
[qassetmean, qassetcovar] = estimateScenarioMoments(q)
```

```
passetmean =
```

```
0.0095
0.0103
0.0124
0.1505
```

```
passetcovar =
```

```
0.0054    0.0000   -0.0005   -0.0006
0.0000    0.0021    0.0015    0.0010
-0.0005    0.0015    0.0046    0.0026
-0.0006    0.0010    0.0026    0.0100
```

```
qassetmean =
```

```
0.0092
0.0082
0.0094
0.1463
```

```
qassetcovar =
```

```
0.0071   -0.0000   -0.0006   -0.0006
-0.0000    0.0032    0.0023    0.0015
-0.0006    0.0023    0.0064    0.0036
-0.0006    0.0015    0.0036    0.0133
```

The first `PortfolioMAD` object, `p`, contains scenarios obtained from price data in `Y` where `NaN` values are discarded and the second `PortfolioMAD` object, `q`, contains scenarios

obtained from price data in Y that accommodate missing values. Each time you run this example, you will get different estimates for the moments in p and q .

Simulating Normal Scenarios from Time Series Data

The `simulateNormalScenariosByData` method also accepts asset returns or prices stored in financial time series (`fints`) objects. The method implicitly works with matrices of data or data in a `fints` object using the same rules for whether the data are returns or prices. To illustrate, use `fints` to create the `fints` object `Xfts` that contains asset returns generated with `fints` (see “Simulating Normal Scenarios from Returns or Prices” on page 6-45) and add series labels:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
      0.00034 0.002408 0.0017 0.000992;
      0.00016 0.0017 0.0048 0.0028;
      0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);

d = (datenum('31-jan-2001'):datenum('31-dec-2010'))';
Xfts = fints(d, zeros(numel(d),4), {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'});
Xfts = tomonthly(Xfts);

Xfts.Bonds = X(:,1);
Xfts.LargeCap = X(:,2);
Xfts.SmallCap = X(:,3);
Xfts.Emerging = X(:,4);

p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Xfts, 20000);

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0038
    0.0078
    0.0102
    0.1492

passetcovar =

    0.0053    0.0004    0.0001   -0.0000
    0.0004    0.0024    0.0017    0.0010
    0.0001    0.0017    0.0048    0.0028
   -0.0000    0.0010    0.0028    0.0103
```

The name-value inputs 'DataFormat' to handle return or price data and 'MissingData' to ignore or use samples with missing values also work for `fints` data. In addition, `simulateNormalScenariosByData` extracts asset names or identifiers from a `fints` object if the argument name 'GetAssetList' is set to `true` (the default value is `false`). If the 'GetAssetList' value is `true`, the identifiers are used to set the `AssetList` property of the `PortfolioMAD` object. Thus, repeating the formation of the `PortfolioMAD` object `q` from the previous example with the 'GetAssetList' flag set to `true` extracts the series labels from the `fints` object:

```
p = simulateNormalScenariosByData(p, Xfts, 20000, 'getassetlist', true);
disp(p.AssetList)

'Bonds'      'LargeCap'      'SmallCap'      'Emerging'
```

If you set the 'GetAssetList' flag set to `true` and your input data is in a matrix, `simulateNormalScenariosByData` uses the default labeling scheme from `setAssetList` as described in “Setting Up a List of Asset Identifiers” on page 6-35.

Simulating Normal Scenarios for Mean and Covariance

Given the mean and covariance of asset returns, use the `simulateNormalScenariosByMoments` method to simulate multivariate normal scenarios. The mean can be either a row or column vector and the covariance matrix must be a symmetric positive-semidefinite matrix. Various rules for scalar expansion apply. To illustrate using `simulateNormalScenariosByMoments`, start with moments in `m` and `C` and generate 20,000 scenarios:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
      0.00034 0.002408 0.0017 0.000992;
      0.00016 0.0017 0.0048 0.0028;
      0 0.000992 0.0028 0.010208 ];

p = PortfolioMAD;
p = simulateNormalScenariosByMoments(p, m, C, 20000);
[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0040
    0.0084
    0.0105
    0.1513
```



```

passetcovar =
    0.0053    0.0003    0.0002    0.0001
    0.0003    0.0024    0.0017    0.0009
    0.0002    0.0017    0.0048    0.0028
    0.0001    0.0009    0.0028    0.0102

```

`simulateNormalScenariosByMoments` performs scalar expansion on arguments for the moments of asset returns. If `NumAssets` has not already been set, a scalar argument is interpreted as a scalar with `NumAssets` set to 1. `simulateNormalScenariosByMoments` provides an additional optional argument to specify the number of assets so that scalar expansion works with the correct number of assets. In addition, if either a scalar or vector is input for the covariance of asset returns, a diagonal matrix is formed such that a scalar expands along the diagonal and a vector becomes the diagonal.

Working with a Riskless Asset

The `PortfolioMAD` object has a separate `RiskFreeRate` property that stores the rate of return of a riskless asset. Thus, you can separate your universe into a riskless asset and a collection of risky assets. For example, assume that your riskless asset has a return in the scalar variable `r0`, then the property for the `RiskFreeRate` is set using the constructor:

```

r0 = 0.01/12;

p = PortfolioMAD;
p = PortfolioMAD('RiskFreeRate', r0);
disp(p.RiskFreeRate);

8.3333e-04

```

Note: If your portfolio problem has a budget constraint such that your portfolio weights must sum to 1, then the riskless asset is irrelevant.

Working with Transaction Costs

The difference between net and gross portfolio returns is transaction costs. The net portfolio return proxy has distinct proportional costs to purchase and to sell assets

which are maintained in the PortfolioMAD object properties `BuyCost` and `SellCost`. Transaction costs are in units of total return and, as such, are proportional to the price of an asset so that they enter the model for net portfolio returns in return form. For example, suppose you have a stock currently priced \$40 and your usual transaction costs are 5 cents per share. Then the transaction cost for the stock is $0.05/40 = 0.00125$ (as defined in “Net Portfolio Returns” on page 6-4). Costs are entered as positive values and credits are entered as negative values.

Setting Transaction Costs Using the Constructor

To set up transaction costs, you must specify an initial or current portfolio in the `InitPort` property. If the initial portfolio is not set when you set up the transaction cost properties, `InitPort` is 0. The properties for transaction costs can be set using the constructor `PortfolioMAD`. For example, assume that purchase and sale transaction costs are in the variables `bc` and `sc` and an initial portfolio is in the variable `x0`, then transaction costs are set:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioMAD('BuyCost', bc, 'SellCost', sc, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5
```

```
0.0013
0.0013
0.0013
0.0013
0.0013
```

```
0.0013
0.0070
0.0013
0.0013
0.0024
```

```
0.4000
0.2000
0.2000
0.1000
```

```
0.1000
```

Setting Transaction Costs Using the setCosts Method

You can also set the properties for transaction costs using `setCosts`. Assume that you have the same costs and initial portfolio as in the previous example. Given a `PortfolioMAD` object `p` with an initial portfolio already set, use `setCosts` to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = PortfolioMAD('InitPort', x0);
p = setCosts(p, bc, sc);
```

```
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5
```

```
0.0013
0.0013
0.0013
0.0013
0.0013
```

```
0.0013
0.0070
0.0013
0.0013
0.0024
```

```
0.4000
0.2000
0.2000
0.1000
0.1000
```

You can also set up the initial portfolio's `InitPort` value as an optional argument to `setCosts` so that the following is an equivalent way to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
```

```
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];  
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = PortfolioMAD;  
p = setCosts(p, bc, sc, x0);
```

```
disp(p.NumAssets);  
disp(p.BuyCost);  
disp(p.SellCost);  
disp(p.InitPort);
```

```
5
```

```
0.0013  
0.0013  
0.0013  
0.0013  
0.0013
```

```
0.0013  
0.0070  
0.0013  
0.0013  
0.0024
```

```
0.4000  
0.2000  
0.2000  
0.1000  
0.1000
```

Setting Transaction Costs with Scalar Expansion

Both the constructor `PortfolioMAD` and `setCosts` method implement scalar expansion on the arguments for transaction costs and the initial portfolio. If the `NumAssets` property is already set in the `PortfolioMAD` object, scalar arguments for these properties are expanded to have the same value across all dimensions. In addition, `setCosts` lets you specify `NumAssets` as an optional final argument. For example, assume that you have an initial portfolio `x0` and you want to set common transaction costs on all assets in your universe. You can set these costs in any of these equivalent ways:

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];  
p = PortfolioMAD('InitPort', x0, 'BuyCost', 0.002, 'SellCost', 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];  
p = PortfolioMAD('InitPort', x0);  
p = setCosts(p, 0.002, 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];  
p = PortfolioMAD;  
p = setCosts(p, 0.002, 0.002, x0);
```

To clear costs from your PortfolioMAD object, use either the constructor or `setCosts` with empty inputs for the properties to be cleared. For example, you can clear sales costs from the PortfolioMAD object `p` in the previous example:

```
p = PortfolioMAD(p, 'SellCost', []);
```

See Also

[PortfolioMAD](#) | [setCosts](#) | [setScenarios](#) | [simulateNormalScenariosByData](#) | [simulateNormalScenariosByMoments](#)

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56
- “Validate the MAD Portfolio Problem” on page 6-78
- “Estimate Efficient Portfolios” on page 6-83
- “Estimate Efficient Frontiers” on page 6-94
- “Asset Returns and Scenarios” on page 6-41

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Working with MAD Portfolio Constraints

In this section...

“Setting Default Constraints for Portfolio Weights” on page 6-56

“Working with Bound Constraints” on page 6-59

“Working with Budget Constraints” on page 6-61

“Working with Group Constraints” on page 6-62

“Working with Group Ratio Constraints” on page 6-65

“Working with Linear Equality Constraints” on page 6-68

“Working with Linear Inequality Constraints” on page 6-70

“Working with Average Turnover Constraints” on page 6-72

“Working with One-Way Turnover Constraints” on page 6-74

The final element for a complete specification of a portfolio optimization problem is the set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset \mathcal{R}^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or “default” portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to 1 (using the budget constraint). For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

Setting Default Constraints for Portfolio Weights

The “default” MAD portfolio problem has two constraints on portfolio weights:

- Portfolio weights must be nonnegative.
- Portfolio weights must sum to 1.

Implicitly, these constraints imply that portfolio weights are no greater than 1, although this is a superfluous constraint to impose on the problem.

Setting Default Constraints Using the Constructor

Given a portfolio optimization problem with `NumAssets = 20` assets, use the `PortfolioMAD` constructor to set up a default problem and explicitly set bounds and budget constraints:

```
p = PortfolioMAD('NumAssets', 20, 'LowerBound', 0, 'Budget', 1);
disp(p);
```

PortfolioMAD with properties:

```
    BuyCost: []
    SellCost: []
    RiskFreeRate: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
    Name: []
    NumAssets: 20
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: [20x1 double]
    UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
    GroupA: []
    GroupB: []
    LowerRatio: []
    UpperRatio: []
```

Setting Default Constraints Using the setDefaultConstraints Method

An alternative approach is to use the `setDefaultConstraints` method. If the number of assets is already known in a `PortfolioMAD` object, use `setDefaultConstraints` with no arguments to set up the necessary bound and budget constraints. Suppose you have 20 assets to set up the portfolio set for a default problem:

```
p = PortfolioMAD('NumAssets', 20);  
p = setDefaultConstraints(p);  
disp(p);
```

PortfolioMAD with properties:

```
    BuyCost: []  
    SellCost: []  
RiskFreeRate: []  
    Turnover: []  
BuyTurnover: []  
SellTurnover: []  
NumScenarios: []  
    Name: []  
    NumAssets: 20  
    AssetList: []  
    InitPort: []  
AInequality: []  
bInequality: []  
    AEquality: []  
    bEquality: []  
LowerBound: [20x1 double]  
UpperBound: []  
LowerBudget: 1  
UpperBudget: 1  
GroupMatrix: []  
LowerGroup: []  
UpperGroup: []  
    GroupA: []  
    GroupB: []  
LowerRatio: []  
UpperRatio: []
```

If the number of assets is unknown, `setDefaultConstraints` accepts `NumAssets` as an optional argument to form a portfolio set for a default problem. Suppose you have 20 assets:

```
p = PortfolioMAD;  
p = setDefaultConstraints(p, 20);  
disp(p);
```

PortfolioMAD with properties:

```
    BuyCost: []  
    SellCost: []
```



```

RiskFreeRate: []
  Turnover: []
  BuyTurnover: []
  SellTurnover: []
NumScenarios: []
  Name: []
  NumAssets: 20
  AssetList: []
  InitPort: []
AInequality: []
bInequality: []
  AEquality: []
  bEquality: []
  LowerBound: [20x1 double]
  UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
  LowerGroup: []
  UpperGroup: []
    GroupA: []
    GroupB: []
  LowerRatio: []
  UpperRatio: []

```

Working with Bound Constraints

Bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “Bound Constraints” on page 6-9). Although every MAD portfolio set must be bounded, it is not necessary to specify a MAD portfolio set with explicit bound constraints. For example, you can create a MAD portfolio set with an implicit upper bound constraint or a MAD portfolio set with just average turnover constraints. The bound constraints have properties `LowerBound` for the lower-bound constraint and `UpperBound` for the upper-bound constraint. Set default values for these constraints using the `setDefaultConstraints` method (see “Setting Default Constraints for Portfolio Weights” on page 6-56).

Setting Bounds Using the Constructor

The properties for bound constraints are set through the constructor `PortfolioMAD`. Suppose you have a balanced fund with stocks that can range from 50 to 75% of your portfolio and bonds that can range from 25 to 50% of your portfolio. The bound constraints for a balanced fund are set with:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioMAD('LowerBound', lb, 'UpperBound', ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);

2

0.5000
0.2500

0.7500
0.5000
```

To continue with this example, you must set up a budget constraint. For details, see “Budget Constraints” on page 6-10.

Setting Bounds Using the `setBounds` Method

You can also set the properties for bound constraints using `setBounds`. Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. Given a `PortfolioMAD` object `p`, use `setBounds` to set the bound constraints:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioMAD;
p = setBounds(p, lb, ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);

2

0.5000
0.2500

0.7500
0.5000
```

Setting Bounds Using the Constructor or `setBounds` Method

Both the constructor `PortfolioMAD` and `setBounds` method implement scalar expansion on either the `LowerBound` or `UpperBound` properties. If the `NumAssets`

property is already set in the PortfolioMAD object, scalar arguments for either property expand to have the same value across all dimensions. In addition, `setBounds` lets you specify `NumAssets` as an optional argument. Suppose you have a universe of 500 assets and you want to set common bound constraints on all assets in your universe. Specifically, you are a long-only investor and want to hold no more than 5% of your portfolio in any single asset. You can set these bound constraints in any of these equivalent ways:

```
p = PortfolioMAD('NumAssets', 500, 'LowerBound', 0, 'UpperBound', 0.05);
```

or

```
p = PortfolioMAD('NumAssets', 500);
p = setBounds(p, 0, 0.05);
```

or

```
p = PortfolioMAD;
p = setBounds(p, 0, 0.05, 500);
```

To clear bound constraints from your PortfolioMAD object, use either the PortfolioMAD constructor or `setBounds` with empty inputs for the properties to be cleared. For example, to clear the upper-bound constraint from the PortfolioMAD object `p` in the previous example:

```
p = PortfolioMAD(p, 'UpperBound', []);
```

Working with Budget Constraints

The budget constraint is an optional linear constraint that maintains upper and lower bounds on the sum of portfolio weights (see “Budget Constraints” on page 5-10). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a MAD portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to 1 using `setDefaultConstraints` (see “Setting Default Constraints for Portfolio Weights” on page 6-56).

Setting Budget Constraints Using the Constructor

The properties for the budget constraint can also be set using the constructor `PortfolioMAD`. Suppose you have an asset universe with many risky assets and a riskless asset and you want to ensure that your portfolio never holds more than 1% cash,

that is, you want to ensure that you are 99 to 100% invested in risky assets. The budget constraint for this portfolio can be set with:

```
p = PortfolioMAD('LowerBudget', 0.99, 'UpperBudget', 1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
0.9900
```

```
1
```

Setting Budget Constraints Using the `setBudget` Method

You can also set the properties for a budget constraint using `setBudget`. Suppose you have a fund that permits up to 10% leverage which means that your portfolio can be from 100 to 110% invested in risky assets. Given a `PortfolioMAD` object `p`, use `setBudget` to set the budget constraints:

```
p = PortfolioMAD;  
p = setBudget(p, 1, 1.1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
1
```

```
1.1000
```

If you were to continue with this example, then set the `RiskFreeRate` property to the borrowing rate to finance possible leveraged positions. For details on the `RiskFreeRate` property, see “Working with a Riskless Asset” on page 6-51. To clear either bound for the budget constraint from your `PortfolioMAD` object, use either the constructor `PortfolioMAD` or `setBudget` with empty inputs for the properties to be cleared. For example, clear the upper-budget constraint from the `PortfolioMAD` object `p` in the previous example with:

```
p = PortfolioMAD(p, 'UpperBudget', []);
```

Working with Group Constraints

Group constraints are optional linear constraints that group assets together and enforce bounds on the group weights (see “Group Constraints” on page 6-11). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with 1 or 0) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership

matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the Constructor

The properties for group constraints are set through the constructor `PortfolioMAD`. Suppose you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio, then you can set group constraints:

```
G = [ 1 1 1 0 0 ];
p = PortfolioMAD('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);

5
1     1     1     0     0
0.3000
```

The group matrix `G` can also be a logical matrix so that the following code achieves the same result.

```
G = [ true true true false false ];
p = PortfolioMAD('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);

5
1     1     1     0     0
0.3000
```

Setting Group Constraints Using the `setGroups` and `addGroups` Methods

You can also set the properties for group constraints using `setGroups`. Suppose you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a `PortfolioMAD` object `p`, use `setGroups` to set the group constraints:

```
G = [ true true true false false ];
p = PortfolioMAD;
```

```
p = setGroups(p, G, [], 0.3);  
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.UpperGroup);
```

```
5
```

```
1     1     1     0     0
```

```
0.3000
```

In this example, you would set the `LowerGroup` property to be empty (`[]`).

Suppose you want to add another group constraint to make odd-numbered assets constitute at least 20% of your portfolio. Set up an augmented group matrix and introduce infinite bounds for unconstrained group bounds or use the `addGroups` method to build up group constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioMAD;  
G = [ true true true false false ]; % group matrix for first group constraint  
p = setGroups(p, G, [], 0.3);  
G = [ true false true false true ]; % group matrix for second group constraint  
p = addGroups(p, G, 0.2);  
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.LowerGroup);  
disp(p.UpperGroup);
```

```
5
```

```
1     1     1     0     0  
1     0     1     0     1
```

```
-Inf
```

```
0.2000
```

```
0.3000
```

```
Inf
```

`addGroups` determines which bounds are unbounded so you only need to focus on the constraints that you want to set.

The constructor `PortfolioMAD` and `setGroups` and `addGroups` implement scalar expansion on either the `LowerGroup` or `UpperGroup` properties based on the dimension of the group matrix in the property `GroupMatrix`. Suppose you have a universe of 30 assets with 6 asset classes such that assets 1-5, assets 6-12, assets 13-18, assets 19-22, assets 23-27, and assets 28-30 constitute each of your asset classes and you want each

asset class to fall from 0 to 25% of your portfolio. Let the following group matrix define your groups and scalar expansion define the common bounds on each group:

```
p = PortfolioMAD;
G = blkdiag(true(1,5), true(1,7), true(1,6), true(1,4), true(1,5), true(1,3));
p = setGroups(p, G, 0, 0.25);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

30

Columns 1 through 13

```
 1   1   1   1   1   0   0   0   0   0   0   0   0
 0   0   0   0   0   1   1   1   1   1   1   1   0
 0   0   0   0   0   0   0   0   0   0   0   0   1
 0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0   0   0   0
```

Columns 14 through 26

```
 0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0   0   0   0
 1   1   1   1   1   0   0   0   0   0   0   0   0
 0   0   0   0   0   1   1   1   1   1   0   0   0
 0   0   0   0   0   0   0   0   0   0   1   1   1
 0   0   0   0   0   0   0   0   0   0   0   0   0
```

Columns 27 through 30

```
 0   0   0   0
 0   0   0   0
 0   0   0   0
 0   0   0   0
 1   0   0   0
 0   1   1   1
```

```
 0
 0
 0
 0
 0
 0
```

```
0.2500
0.2500
0.2500
0.2500
0.2500
0.2500
```

Working with Group Ratio Constraints

Group ratio constraints are optional linear constraints that maintain bounds on proportional relationships among groups of assets (see “Group Ratio Constraints” on

page 6-12). Although the constraints are implemented as general constraints, the usual convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with 1 or 0) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group to a comparison group fall within specified bounds. Group ratio constraints have properties:

- `GroupA` for the base membership matrix
- `GroupB` for the comparison membership matrix
- `LowerRatio` for the lower-bound constraint on the ratio of groups
- `UpperRatio` for the upper-bound constraint on the ratio of groups

Setting Group Ratio Constraints Using the Constructor

The properties for group ratio constraints are set using constructor `PortfolioMAD`. For example, assume you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ]; % financial companies
GB = [ 0 0 0 1 1 1 ]; % nonfinancial companies
p = PortfolioMAD('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

6

```
1    1    1    0    0    0
```

```
0    0    0    1    1    1
```

0.5000

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = PortfolioMAD('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
```



```
disp(p.UpperRatio);
6
1     1     1     0     0     0
0     0     0     1     1     1
0.5000
```

Setting Group Ratio Constraints Using the `setGroupRatio` and `addGroupRatio` Methods

You can also set the properties for group ratio constraints using `setGroupRatio`. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). Given a `PortfolioMAD` object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false true true true ]; % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
6
1     1     1     0     0     0
0     0     0     1     1     1
0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose you want to add another group ratio constraint to ensure that the weights in odd-numbered assets constitute at least 20% of the weights in nonfinancial assets your portfolio. You can set up augmented group ratio matrices and introduce infinite bounds for unconstrained group ratio bounds, or you can use the `addGroupRatio` method to build up group ratio constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioMAD;
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);
```

```
GA = [ true false true false true false ]; % odd-numbered companies
GB = [ false false false true true true ]; % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);

6

1      1      1      0      0      0
1      0      1      0      1      0

0      0      0      1      1      1
0      0      0      1      1      1

-Inf
0.2000

0.5000
Inf
```

Notice that `addGroupRatio` determines which bounds are unbounded so you only need to focus on the constraints you want to set.

The constructor `PortfolioMAD`, `setGroupRatio`, and `addGroupRatio` implement scalar expansion on either the `LowerRatio` or `UpperRatio` properties based on the dimension of the group matrices in `GroupA` and `GroupB` properties.

Working with Linear Equality Constraints

Linear equality constraints are optional linear constraints that impose systems of equalities on portfolio weights (see “Linear Equality Constraints” on page 6-9). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `bEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the Constructor

The properties for linear equality constraints are set using the constructor `PortfolioMAD`. Suppose you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. To set this constraint:

```
A = [ 1 1 1 0 0 ];
```

```

b = 0.5;
p = PortfolioMAD('AEquality', A, 'bEquality', b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

```

```
5
```

```
1    1    1    0    0
```

```
0.5000
```

Setting Linear Equality Constraints Using the `setEquality` and `addEquality` Methods

You can also set the properties for linear equality constraints using `setEquality`. Suppose you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. Given a `PortfolioMAD` object `p`, use `setEquality` to set the linear equality constraints:

```

A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setEquality(p, A, b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

```

```
5
```

```
1    1    1    0    0
```

```
0.5000
```

Suppose you want to add another linear equality constraint to ensure that the last three assets also constitute 50% of your portfolio. You can set up an augmented system of linear equalities or use `addEquality` to build up linear equality constraints. For this example, create another system of equalities:

```

p = PortfolioMAD;
A = [ 1 1 1 0 0 ];    % first equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % second equality constraint

```

```
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000
```

The constructor `PortfolioMAD` and `setEquality` and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

Working with Linear Inequality Constraints

Linear inequality constraints are optional linear constraints that impose systems of inequalities on portfolio weights (see “Linear Inequality Constraints” on page 6-8). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the Constructor

The properties for linear inequality constraints are set using the constructor `PortfolioMAD`. Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. To set up these constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD('AInequality', A, 'bInequality', b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);

5

1     1     1     0     0
```

```
0.5000
```

Setting Linear Inequality Constraints Using the `setInequality` and `addInequality` Methods

You can also set the properties for linear inequality constraints using `setInequality`. Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 50% of your portfolio. Given a `PortfolioMAD` object `p`, use `setInequality` to set the linear inequality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setInequality(p, A, b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5
```

```
1    1    1    0    0
```

```
0.5000
```

Suppose you want to add another linear inequality constraint to ensure that the last three assets constitute at least 50% of your portfolio. You can set up an augmented system of linear inequalities or use the `addInequality` method to build up linear inequality constraints. For this example, create another system of inequalities:

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ]; % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5
```

```
1    1    1    0    0
```

```
0      0      -1      -1      -1

0.5000
-0.5000
```

The `PortfolioMAD` constructor and `setInequality` and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

Working with Average Turnover Constraints

The turnover constraint is an optional linear absolute value constraint (see “Average Turnover Constraints” on page 6-12) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `PortfolioMAD` constructor or the `setTurnover` method. The turnover constraint depends on an initial or current portfolio, which is assumed to be zero if not set when the turnover constraint is set. The turnover constraint has properties `Turnover`, for the upper bound on average turnover, and `InitPort`, for the portfolio against which turnover is computed.

Setting Average Turnover Constraints Using the Constructor

The properties for the turnover constraints are set using the `PortfolioMAD` constructor. Suppose you have an initial portfolio of 10 assets in a variable `x0` and you want to ensure that average turnover is no more than 30%. To set this turnover constraint:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('Turnover', 0.3, 'InitPort', x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);

10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
```

```
0.1000
```

Note if the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 6-37).

Setting Average Turnover Constraints Using the `setTurnover` Method

You can also set properties for portfolio turnover using `setTurnover` to specify both the upper bound for average turnover and an initial portfolio. Suppose you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that average turnover is no more than 30%. Given a `PortfolioMAD` object `p`, use `setTurnover` to set the turnover constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setTurnover(p, 0.3);
```

```
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
```

```
0.3000
```

```
0.1200
```

```
0.0900
```

```
0.0800
```

```
0.0700
```

```
0.1000
```

```
0.1000
```

```
0.1500
```

```
0.1100
```

```
0.0800
```

```
0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD;
p = setTurnover(p, 0.3, x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10
```

```
0.3000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

`setTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the `PortfolioMAD` object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setTurnover` lets you specify `NumAssets` as an optional argument. To clear turnover from your `PortfolioMAD` object, use the constructor `PortfolioMAD` or `setTurnover` with empty inputs for the properties to be cleared.

Working with One-Way Turnover Constraints

One-way turnover constraints are optional constraints (see “One-Way Turnover Constraints” on page 6-13) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the `PortfolioMAD` constructor or the `setOneWayTurnover` method. One-way turnover constraints depend upon an initial or current portfolio, which is assumed to be zero if not set when the turnover constraints are set. One-way turnover constraints have properties `BuyTurnover`, for the upper bound on net purchases, `SellTurnover`, for the upper bound on net sales, and `InitPort`, for the portfolio against which turnover is computed.

Setting One-Way Turnover Constraints Using the Constructor

The Properties for the one-way turnover constraints are set using the `PortfolioMAD` constructor. Suppose you have an initial portfolio with 10 assets in a variable `x0` and you want to ensure that turnover on purchases is no more than 30% and turnover on sales is no more than 20% of the initial portfolio. To set these turnover constraints:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('BuyTurnover', 0.3, 'SellTurnover', 0.2, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
```



```

disp(p.InitPort);
10
0.3000
0.2000
0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000

```

If the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 6-37).

Setting Turnover Constraints Using the `setOneWayTurnover` Method

You can also set properties for portfolio turnover using `setOneWayTurnover` to specify to the upper bounds for turnover on purchases (`BuyTurnover`) and sales (`SellTurnover`) and an initial portfolio. Suppose you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that turnover on purchases is no more than 30% and that turnover on sales is no more than 20% of the initial portfolio. Given a `PortfolioMAD` object `p`, use `setOneWayTurnover` to set the turnover constraints with and without the initial portfolio being set previously:

```

x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);

disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
or
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD;
p = setOneWayTurnover(p, 0.3, 0.2, x0);
disp(p.NumAssets);

```

```
disp(p.BuyTurnover);  
disp(p.SellTurnover);  
disp(p.InitPort);
```

```
10
```

```
0.3000
```

```
0.2000
```

```
0.1200
```

```
0.0900
```

```
0.0800
```

```
0.0700
```

```
0.1000
```

```
0.1000
```

```
0.1500
```

```
0.1100
```

```
0.0800
```

```
0.1000
```

`setOneWayTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the `PortfolioMAD` object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setOneWayTurnover` lets you specify `NumAssets` as an optional argument. To remove one-way turnover from your `PortfolioMAD` object, use the constructor `PortfolioMAD` or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints`
| `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` |
`setOneWayTurnover` | `setTurnover`

Related Examples

- “Constructing the `PortfolioMAD` Object” on page 6-26
- “Validate the MAD Portfolio Problem” on page 6-78
- “Estimate Efficient Portfolios” on page 6-83
- “Estimate Efficient Frontiers” on page 6-94
- “Asset Returns and Scenarios” on page 6-41

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Validate the MAD Portfolio Problem

In this section...

“Validating a MAD Portfolio Set” on page 6-78

“Validating MAD Portfolios” on page 6-80

In some cases, you may want to validate either your inputs to, or outputs from, a portfolio optimization problem. Although most error checking that occurs during the problem setup phase catches most difficulties with a portfolio optimization problem, the processes to validate MAD portfolio sets and portfolios are time consuming and are best done offline. Consequently, the portfolio optimization tools have specialized methods to validate MAD portfolio sets and portfolios. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

Validating a MAD Portfolio Set

Since it is necessary and sufficient that your MAD portfolio set must be a nonempty, closed, and bounded set to have a valid portfolio optimization problem, the method `estimateBounds` lets you examine your portfolio set to determine if it is nonempty and, if nonempty, whether it is bounded. Suppose you have the following MAD portfolio set which is an empty set because the initial portfolio at 0 is too far from a portfolio that satisfies the budget and turnover constraint:

```
p = PortfolioMAD('NumAssets', 3, 'Budget', 1);
p = setTurnover(p, 0.3, 0);
```

If a MAD portfolio set is empty, `estimateBounds` returns NaN bounds and sets the `isbounded` flag to `[]`:

```
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
NaN
NaN
NaN
```

```
ub =
```

```
NaN
NaN
```

```

NaN
isbounded =
    []

```

Suppose you create an unbounded MAD portfolio set as follows:

```

p = PortfolioMAD('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

    -Inf
    -Inf

ub =

    1.0e-008 *
    -0.3712
         Inf

isbounded =

    0

```

In this case, `estimateBounds` returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Finally, suppose you created a MAD that is both nonempty and bounded. `estimateBounds` not only validates the set, but also obtains tighter bounds which is useful if you are concerned with the actual range of portfolio choices for individual assets in your :

```

p = PortfolioMAD;
p = setBudget(p, 1,1);
p = setBounds(p, [ -0.1; 0.2; 0.3; 0.2 ], [ 0.5; 0.3; 0.9; 0.8 ]);

[lb, ub, isbounded] = estimateBounds(p)

lb =

    -0.1000
     0.2000
     0.3000

```

```
0.2000
ub =
0.3000
0.3000
0.7000
0.6000
isbounded =
1
```

In this example, all but the second asset have tighter upper bounds than the input upper bound implies.

Validating MAD Portfolios

Given a MAD portfolio set specified in a PortfolioMAD object, you often want to check if specific portfolios are feasible with respect to the portfolio set. This can occur with, for example, initial portfolios and with portfolios obtained from other procedures. The `checkFeasibility` method determines whether a collection of portfolios is feasible. Suppose you perform the following portfolio optimization and want to determine if the resultant efficient portfolios are feasible relative to a modified problem.

First, set up a problem in the PortfolioMAD object `p`, estimate efficient portfolios in `pwgt`, and then confirm that these portfolios are feasible relative to the initial problem:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p);
```

```
checkFeasibility(p, pwgt)
```

```
ans =
```

```
1 1 1 1 1 1 1 1 1 1
```

Next, set up a different portfolio problem that starts with the initial problem with an additional a turnover constraint and an equally weighted initial portfolio:

```
q = setTurnover(p, 0.3, 0.25);
checkFeasibility(q, pwgt)
```

```
ans =
```

```
0 0 1 1 1 0 0 0 0 0
```

In this case, only two of the ten efficient portfolios from the initial problem are feasible relative to the new problem in PortfolioMAD object `q`. Solving the second problem using `checkFeasibility` demonstrates that the efficient portfolio for PortfolioMAD object `q` is feasible relative to the initial problem:

```
qwgt = estimateFrontier(q);
checkFeasibility(p, qwgt)
```

```
ans =
```

```
1 1 1 1 1 1 1 1 1 1
```

See Also

PortfolioMAD | checkFeasibility | estimateBounds

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56
- “Estimate Efficient Portfolios” on page 6-83
- “Estimate Efficient Frontiers” on page 6-94
- “Asset Returns and Scenarios” on page 6-41

More About

- “PortfolioMAD Object” on page 6-17

- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Estimate Efficient Portfolios

In this section...

“Obtaining Portfolios Along the Entire Efficient Frontier” on page 6-83

“Obtaining Endpoints of the Efficient Frontier” on page 6-85

“Obtaining Efficient Portfolios for Target Returns” on page 6-87

“Obtaining Efficient Portfolios for Target Risks” on page 6-89

“Choosing and Controlling the Solver” on page 6-92

There are two ways to look at a portfolio optimization problem that depends on what you are trying to do. One goal is to estimate efficient portfolios and the other is to estimate efficient frontiers. This section focuses on the former goal and “Estimate Efficient Frontiers” on page 6-94 focuses on the latter goal. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

Obtaining Portfolios Along the Entire Efficient Frontier

The most basic way to obtain optimal portfolios is to obtain points over the entire range of the efficient frontier. Given a portfolio optimization problem in a PortfolioMAD object, the `estimateFrontier` method computes efficient portfolios spaced evenly according to the return proxy from the minimum to maximum return efficient portfolios. The number of portfolios estimated is controlled by the hidden property `defaultNumPorts` which is set to 10. A different value for the number of portfolios estimated is specified as input to `estimateFrontier`. This example shows the default number of efficient portfolios over the entire range of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
disp(pwgt);
```

Columns 1 through 8

```

0.8907  0.7289  0.5614  0.3946  0.2257  0.0612  0  0.0000
0.0330  0.1163  0.2119  0.3042  0.3998  0.4876  0.4400  0.3125
0.0420  0.0469  0.0472  0.0505  0.0534  0.0580  0.0374  0.0018
0.0343  0.1079  0.1794  0.2507  0.3211  0.3933  0.5226  0.6857

```

Columns 9 through 10

```

0.0000  0.0000
0.1570  0.0000
0.0000  0.0000
0.8430  1.0000

```

If you want only four portfolios in the previous example:

```
pwgt = estimateFrontier(p, 4);
```

```
disp(pwgt);
```

```

0.8907  0.3946  0  0.0000
0.0330  0.3042  0.4401  0.0000
0.0420  0.0505  0.0373  0.0000
0.0343  0.2507  0.5227  1.0000

```

Starting from the initial portfolio, `estimateFrontier` also returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

```

```

display(pwgt);
display(pbuy);
display(psell);

```

pwgt =

Columns 1 through 8

```

0.8907  0.7289  0.5614  0.3946  0.2257  0.0612  0  0.0000
0.0330  0.1163  0.2119  0.3042  0.3998  0.4876  0.4400  0.3125
0.0420  0.0469  0.0472  0.0505  0.0534  0.0580  0.0374  0.0018
0.0343  0.1079  0.1794  0.2507  0.3211  0.3933  0.5226  0.6857

```

Columns 9 through 10

```

0.0000  0.0000
0.1570  0.0000
0.0000  0.0000
0.8430  1.0000

```

pbuy =

Columns 1 through 8

```

0.5907  0.4289  0.2614  0.0946  0  0  0  0

```

```

0      0      0      0.0042  0.0998  0.1876  0.1400  0.0125
0      0      0      0      0      0      0      0
0      0.0079  0.0794  0.1507  0.2211  0.2933  0.4226  0.5857

```

Columns 9 through 10

```

0      0
0      0
0      0
0.7430  0.9000

```

psell =

Columns 1 through 8

```

0      0      0      0      0.0743  0.2388  0.3000  0.3000
0.2670  0.1837  0.0881  0      0      0      0      0
0.1580  0.1531  0.1528  0.1495  0.1466  0.1420  0.1626  0.1982
0.0657      0      0      0      0      0      0      0

```

Columns 9 through 10

```

0.3000  0.3000
0.1430  0.3000
0.2000  0.2000
0      0

```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Endpoints of the Efficient Frontier

In many cases, you might be interested in the endpoint portfolios for the efficient frontier. Suppose you want to determine the range of returns from minimum to maximum to refine a search for a portfolio with a specific target return. Use the `estimateFrontierLimits` method to obtain the endpoint portfolios:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);

```

```
disp(pwgt);  
  
    0.8875    0.0000  
    0.0373    0.0000  
    0.0386    0.0000  
    0.0366    1.0000
```

Note: The endpoints of the efficient frontier depend upon the Scenarios in the PortfolioMAD object. If you change the Scenarios, you are likely to obtain different endpoints.

Starting from an initial portfolio, `estimateFrontierLimits` also returns purchases and sales to get from the initial portfolio to the endpoint portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];  
C = [ 0.0064 0.00408 0.00192 0;  
      0.00408 0.0289 0.0204 0.0119;  
      0.00192 0.0204 0.0576 0.0336;  
      0 0.0119 0.0336 0.1225 ];  
m = m/12;  
C = C/12;  
  
AssetScenarios = mvnrnd(m, C, 20000);  
  
p = PortfolioMAD;  
p = setScenarios(p, AssetScenarios);  
p = setDefaultConstraints(p);  
  
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];  
p = setInitPort(p, pwgt0);  
[pwgt, pbuy, psell] = estimateFrontierLimits(p);  
  
display(pwgt);  
display(pbuy);  
display(psell);  
  
pwgt =  
  
    0.8927    0.0000
```

```

0.0334    0.0000
0.0422    0.0000
0.0317    1.0000

```

pbuy =

```

0.5927    0
0         0
0         0
0         0.9000

```

psell =

```

0         0.3000
0.2666    0.3000
0.1578    0.2000
0.0683    0

```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Efficient Portfolios for Target Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` method accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. For example, assume that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 12%:

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByReturn(p, [0.07, 0.10, .12]);

```

```
display(pwgt);  
  
pwgt =  
  
    0.7537    0.3899    0.1478  
    0.1113    0.2934    0.4136  
    0.0545    0.1006    0.1319  
    0.0805    0.2161    0.3066
```

In some cases, you can request a return for which no efficient portfolio exists. Based on the previous example, suppose you want a portfolio with a 4% return (which is the return of the first asset). A portfolio that is fully invested in the first asset, however, is inefficient. `estimateFrontierByReturn` warns if your target returns are outside the range of efficient portfolio returns and replaces it with the endpoint portfolio of the efficient frontier closest to your target return:

```
pwgt = estimateFrontierByReturn(p, [0.04]);  
  
Warning: One or more target return values are outside the feasible range [  
0.0591121, 0.182542 ].  
  Will return portfolios associated with endpoints of the range for these values.  
> In PortfolioMAD.estimateFrontierByReturn at 90
```

The best way to avoid this situation is to bracket your target portfolio returns with `estimateFrontierLimits` and `estimatePortReturn` (see “Obtaining Endpoints of the Efficient Frontier” on page 6-85 and “Obtaining MAD Portfolio Risks and Returns” on page 6-94).

```
pret = estimatePortReturn(p, p.estimateFrontierLimits);  
  
display(pret);  
  
pret =  
  
    0.0591  
    0.1825
```

This result indicates that efficient portfolios have returns that range from 6.5 to 17.8%. Note, your results for these examples may be different due to the random generation of scenarios.

If you have an initial portfolio, `estimateFrontierByReturn` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, to obtain purchases and sales with target returns of 7%, 10%, and 12%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];  
p = setInitPort(p, pwgt0);
```

```
[pwgt, pbuy, psell] = estimateFrontierByReturn(p, [0.07, 0.10, .12]);
```

```
display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =
```

```
    0.7537    0.3899    0.1478
    0.1113    0.2934    0.4136
    0.0545    0.1006    0.1319
    0.0805    0.2161    0.3066
```

```
pbuy =
```

```
    0.4537    0.0899         0
         0         0    0.1136
         0         0         0
         0    0.1161    0.2066
```

```
psell =
```

```
         0         0    0.1522
    0.1887    0.0066         0
    0.1455    0.0994    0.0681
    0.0195         0         0
```

If you do not have an initial portfolio, the purchase and sale weights assume that your initial portfolio is 0.

Obtaining Efficient Portfolios for Target Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` method accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Suppose you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
```

```
0 0.0119 0.0336 0.1225 ];  
  
AssetScenarios = mvnrnd(m, C, 20000);  
  
p = PortfolioMAD;  
p = setScenarios(p, AssetScenarios);  
p = setDefaultConstraints(p);  
  
pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);  
  
display(pwgt);  
  
pwgt =  
  
    0.2102    0.0621         0  
    0.3957    0.4723    0.4305  
    0.1051    0.1204    0.1291  
    0.2889    0.3452    0.4404
```

In some cases, you can request a risk for which no efficient portfolio exists. Based on the previous example, suppose you want a portfolio with 6% risk (individual assets in this universe have risks ranging from 7 to 42.5%). It turns out that a portfolio with 6% risk cannot be formed with these four assets. `estimateFrontierByRisk` warns if your target risks are outside the range of efficient portfolio risks and replaces it with the endpoint of the efficient frontier closest to your target risk:

```
pwgt = estimateFrontierByRisk(p, 0.06)  
  
Warning: One or more target risk values are outside the feasible range [  
0.0610574, 0.278711 ].  
Will return portfolios associated with endpoints of the range for these values.  
> In PortfolioMAD.estimateFrontierByRisk at 82  
  
pwgt =  
  
    0.8867  
    0.0396  
    0.0404  
    0.0332
```

The best way to avoid this situation is to bracket your target portfolio risks with `estimateFrontierLimits` and `estimatePortRisk` (see “Obtaining Endpoints of the Efficient Frontier” on page 6-85 and “Estimate Efficient Frontiers” on page 6-94).

```
prsk = estimatePortRisk(p, p.estimateFrontierLimits);  
  
display(prsk);  
  
prsk =
```



```
0.0611
0.2787
```

This result indicates that efficient portfolios have risks that range from 7 to 42.5%. Note, your results for these examples may be different due to the random generation of scenarios.

Starting with an initial portfolio, `estimateFrontierByRisk` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the example with target risks of 12%, 14%, and 16%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);
```

```
display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =
```

```
0.2102    0.0621    0
0.3957    0.4723    0.4305
0.1051    0.1204    0.1291
0.2889    0.3452    0.4404
```

```
pbuy =
```

```
0    0    0
0.0957    0.1723    0.1305
0    0    0
0.1889    0.2452    0.3404
```

```
psell =
```

```
0.0898    0.2379    0.3000
0    0    0
0.0949    0.0796    0.0709
0    0    0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

Choosing and Controlling the Solver

When solving portfolio optimizations for a PortfolioMAD object, while all variations of `fmincon` from Optimization Toolbox are supported, using `'sqp'` and `'active-set'` algorithms for `fmincon` is recommended and the use of `'interior-point'` algorithm is not recommended for MAD portfolio optimization.

Unlike Optimization Toolbox which uses the `'trust-region-reflective'` algorithm as the default algorithm for `fmincon`, the portfolio optimization for a PortfolioMAD object uses the `'active-set'` algorithm. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see “Constrained Nonlinear Optimization Algorithms”.

To modify `fmincon` options for MAD portfolio optimizations, use `setSolver` to set the hidden properties `solverType` and `solverOptions` to specify and control the solver. Since these solver properties are hidden, you cannot set them using the PortfolioMAD constructor. The default solver is `fmincon` with the `'sqp'` algorithm and no displayed output, so you do not need to use `setSolver` to specify this.

If you want to specify additional options associated with the `fmincon` solver, `setSolver` accepts these options as name-value pair arguments. For example, if you want to use `fmincon` with the `sqp` algorithm and with displayed output, use `setSolver` with:

```
p = PortfolioMAD;  
p = setSolver(p, 'fmincon', 'Algorithm', 'sqp', 'Display', 'final');  
display(p.solverOptions.Algorithm);  
display(p.solverOptions.Display);
```

```
sqp  
final
```

Alternatively, `setSolver` accepts an `optimoptions` object as the second argument. For example, you can change the algorithm to `trust-region-reflective` with no displayed output as follows:

```
p = PortfolioMAD;  
options = optimoptions('fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'off');  
p = setSolver(p, 'fmincon', options);  
display(p.solverOptions.Algorithm);  
display(p.solverOptions.Display);
```

```
trust-region-reflective  
off
```

See Also

PortfolioMAD | estimateFrontier | estimateFrontierByReturn
| estimateFrontierByRisk | estimateFrontierByRisk |
estimateFrontierLimits | estimatePortReturn | estimatePortRisk |
setSolver

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56
- “Estimate Efficient Frontiers” on page 6-94
- “Asset Returns and Scenarios” on page 6-41

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Estimate Efficient Frontiers

In this section...

“Obtaining MAD Portfolio Risks and Returns” on page 6-94

“Obtaining Portfolio Standard Deviation” on page 6-96

“Plotting the Efficient Frontier” on page 6-97

Whereas “Estimate Efficient Portfolios” on page 6-83 focused on estimation of efficient portfolios, this section focuses on the estimation of efficient frontiers. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

Obtaining MAD Portfolio Risks and Returns

Given any portfolio and, in particular, efficient portfolios, the methods `estimatePortReturn` and `estimatePortRisk` provide estimates for the return (or return proxy), risk (or the risk proxy). Each method has the same input syntax but with different combinations of outputs. Suppose you have this following portfolio optimization problem that gave you a collection of portfolios along the efficient frontier in `pwgt`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = setInitPort(p, pwgt0);
pwgt = estimateFrontier(p)

pwgt =
```

Columns 1 through 8

0.8954	0.7264	0.5573	0.3877	0.2176	0.0495	0.0000	0
0.0310	0.1239	0.2154	0.3081	0.4028	0.4924	0.4069	0.2386
0.0409	0.0524	0.0660	0.0792	0.0907	0.1047	0.1054	0.1132
0.0328	0.0973	0.1613	0.2250	0.2890	0.3534	0.4877	0.6482

Columns 9 through 10

0	0.0000
0.0694	0.0000
0.1221	0.0000
0.8084	1.0000

Note: Remember that the risk proxy for MAD portfolio optimization is mean-absolute deviation.

Given `pwgt0` and `pwgt`, use the portfolio risk and return estimation methods to obtain risks and returns for your initial portfolio and the portfolios on the efficient frontier:

```
prsk0 = estimatePortRisk(p, pwgt0);
pret0 = estimatePortReturn(p, pwgt0);
prsk = estimatePortRisk(p, pwgt);
pret = estimatePortReturn(p, pwgt);
display(prsk0);
display(pret0);
display(prsk);
display(pret);
```

You obtain these risks and returns:

```
prsk0 =
    0.0256
```

```
pret0 =
    0.0072
```

```
prsk =
    0.0178
    0.0193
    0.0233
    0.0286
```

```
0.0348  
0.0414  
0.0489  
0.0584  
0.0692  
0.0809
```

```
pret =
```

```
0.0047  
0.0059  
0.0072  
0.0084  
0.0096  
0.0108  
0.0120  
0.0133  
0.0145  
0.0157
```

Obtaining Portfolio Standard Deviation

The PortfolioMAD object has a method to compute standard deviations of portfolio returns, `estimatePortStd`. This methods work with any portfolios, not necessarily efficient portfolios. For example, the following example obtains five portfolios (`pwgt`) on the efficient frontier and also has an initial portfolio in `pwgt0`. Various portfolio statistics are computed that include the return, risk, and standard deviation. The listed estimates are for the initial portfolio in the first row followed by estimates for each of the five efficient portfolios in subsequent rows.

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];  
C = [ 0.005333 0.00034 0.00016 0;  
0.00034 0.002408 0.0017 0.000992;  
0.00016 0.0017 0.0048 0.0028;  
0 0.000992 0.0028 0.010208 ];  
  
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];  
  
p = PortfolioMAD('initport', pwgt0);  
p = simulateNormalScenariosByMoments(p, m, C, 20000);  
p = setDefaultConstraints(p);
```

```

pwgt = estimateFrontier(p, 5);

pret = estimatePortReturn(p, [pwgt0, pwgt]);
prsk = estimatePortRisk(p, [pwgt0, pwgt]);
pstd = estimatePortStd(p, [pwgt0, pwgt]);

[pret, prsk, pstd]

ans =

    0.0212    0.0305    0.0381
    0.0187    0.0326    0.0407
    0.0514    0.0369    0.0462
    0.0841    0.0484    0.0607
    0.1168    0.0637    0.0796
    0.1495    0.0807    0.1009

```

Plotting the Efficient Frontier

The `plotFrontier` method creates a plot of the efficient frontier for a given portfolio optimization problem. This method accepts several types of inputs and generates a plot with an optional possibility to output the estimates for portfolio risks and returns along the efficient frontier. `plotFrontier` has four different ways that it can be used. In addition to a plot of the efficient frontier, if you have an initial portfolio in the `InitPort` property, `plotFrontier` also displays the return versus risk of the initial portfolio on the same plot. If you have a well-posed portfolio optimization problem set up in a `PortfolioMAD` object and you use `plotFrontier`, you will get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is currently 10 and is maintained in the hidden property `defaultNumPorts`). This example illustrates a typical use of `plotFrontier` to create a new plot:

```

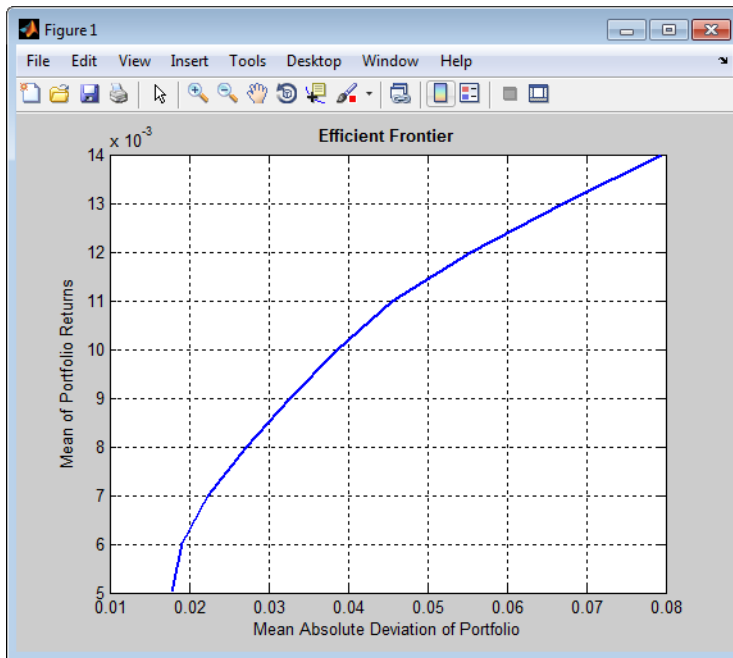
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

plotFrontier(p);

```



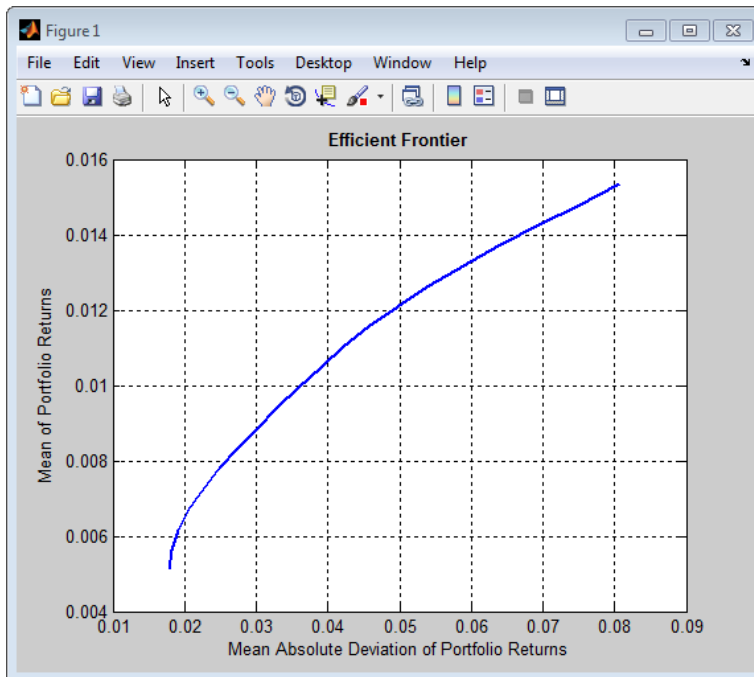
The `Name` property appears as the title of the efficient frontier plot if you set it in the `PortfolioMAD` object. Without an explicit name, the title on the plot would be “Efficient Frontier.” If you want to obtain a specific number of portfolios along the efficient frontier, use `plotFrontier` with the number of portfolios that you want. Suppose you have the `PortfolioMAD` object from the previous example and you want to plot 20 portfolios along the efficient frontier and to obtain 20 risk and return values for each portfolio:

```
[prsk, pret] = plotFrontier(p, 20);
display([pret, prsk]);
```

```
ans =
```

```
0.0049    0.0176
0.0054    0.0179
0.0058    0.0189
0.0063    0.0205
0.0068    0.0225
0.0073    0.0248
0.0078    0.0274
0.0083    0.0302
```


0.0088	0.0331
0.0093	0.0361
0.0098	0.0392
0.0103	0.0423
0.0108	0.0457
0.0112	0.0496
0.0117	0.0539
0.0122	0.0586
0.0127	0.0635
0.0132	0.0687
0.0137	0.0744
0.0142	0.0806



Plotting Existing Efficient Portfolios

If you already have efficient portfolios from any of the "estimateFrontier" methods (see "Estimate Efficient Frontiers" on page 6-94), pass them into plotFrontier directly to plot the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
```

```

C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

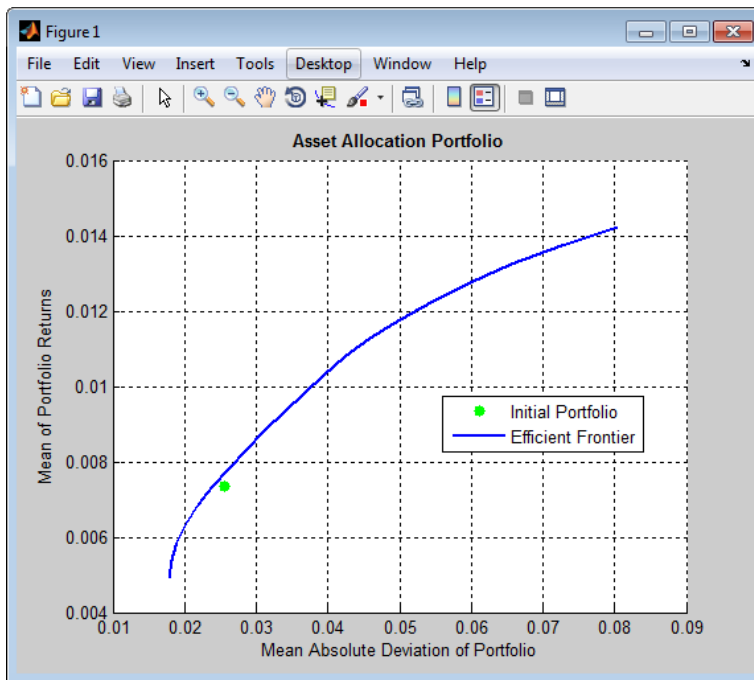
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p, 20);
plotFrontier(p, pwgt)

```



Plotting Existing Efficient Portfolio Risks and Returns

If you already have efficient portfolio risks and returns, you can use the interface to `plotFrontier` to pass them into `plotFrontier` to obtain a plot of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

AssetScenarios = mvnrnd(m, C, 20000);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p);

pret= estimatePortReturn(p, pwgt)
prsk = estimatePortRisk(p, pwgt)

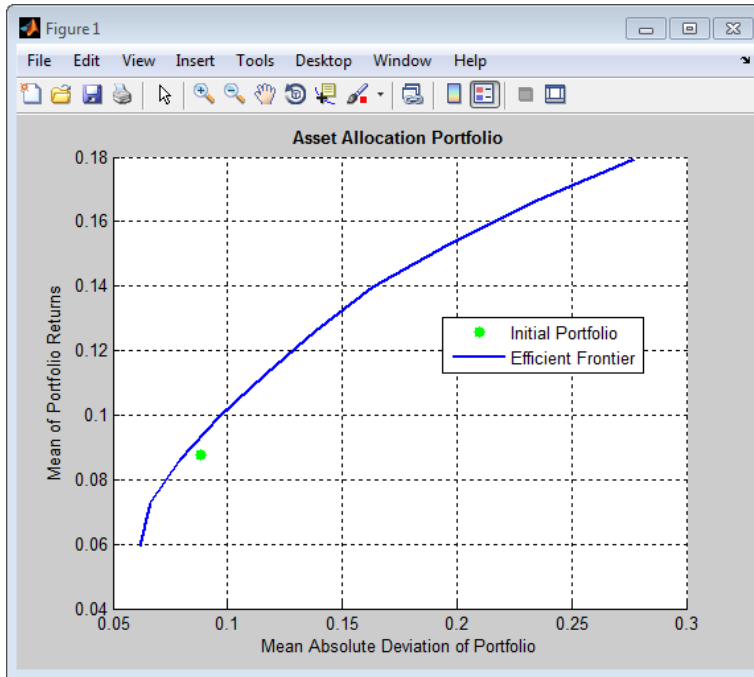
plotFrontier(p, prsk, pret)

pret =

    0.0590
    0.0723
    0.0857
    0.0991
    0.1124
    0.1258
    0.1391
    0.1525
    0.1658
    0.1792

prsk =

    0.0615
    0.0664
    0.0795
    0.0976
    0.1184
    0.1408
    0.1663
    0.1992
    0.2368
    0.2787
```



See Also

`PortfolioMAD` | `estimatePortReturn` | `estimatePortStd` | `plotFrontier`

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56
- “Estimate Efficient Portfolios” on page 6-83
- “Asset Returns and Scenarios” on page 6-41
- “Postprocessing Results” on page 6-104

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2

- “PortfolioMAD Object Workflow”

Postprocessing Results

In this section...

“Setting Up Tradable Portfolios” on page 6-104

“Working with Other Portfolio Objects” on page 6-105

“Troubleshooting MAD Portfolio Optimization Results” on page 6-108

After obtaining efficient portfolios or estimates for expected portfolio risks and returns, use your results to set up trades to move toward an efficient portfolio. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow”.

Setting Up Tradable Portfolios

Suppose you set up a portfolio optimization problem and obtained portfolios on the efficient frontier. Use the `dataset` object from Statistics Toolbox to form a blotter that lists your portfolios with the names for each asset. For example, suppose you want to obtain five portfolios along the efficient frontier. You can set up a blotter with weights multiplied by 100 to view the allocations for each portfolio:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD;
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([100*pwgt], pnames, 'obsnames', p.AssetList);
display(Blotter);

Blotter =
```

	Port1	Port2	Port3	Port4	Port5
Bonds	88.154	50.867	13.611	0	1.0609e-12
Large-Cap Equities	4.0454	22.571	41.276	23.38	7.9362e-13
Small-Cap Equities	4.2804	9.3108	14.028	17.878	6.4823e-14

Emerging Equities 3.5202 17.252 31.084 58.743 100

Note: Your results may differ from this result due to the simulation of scenarios.

This result indicates that you would invest primarily in bonds at the minimum-risk/minimum-return end of the efficient frontier (Port1), and that you would invest completely in emerging equity at the maximum-risk/maximum-return end of the efficient frontier (Port5). You can also select a particular efficient portfolio, for example, suppose you want a portfolio with 15% risk and you add purchase and sale weights outputs obtained from the “estimateFrontier” methods to set up a trade blotter:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD;
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');

p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = p.setDefaultConstraints;

[pwgt, pbuy, psell] = estimateFrontierByRisk(p, 0.15);

Blotter = dataset([100*[pwgt0, pwgt, pbuy, psell]], ...
{'Initial', 'Weight', 'Purchases', 'Sales'}], 'obsnames', p.AssetList);
display(Blotter);

Blotter =
```

	Initial	Weight	Purchases	Sales
Bonds	30	6.0364e-18	0	30
Large-Cap Equities	30	50.179	20.179	0
Small-Cap Equities	20	13.43	0	6.5696
Emerging Equities	10	36.391	26.391	0

If you have prices for each asset (in this example, they can be ETFs), add them to your blotter and then use the tools of the `dataset` object to obtain shares and shares to be traded.

Working with Other Portfolio Objects

The `PortfolioMAD` object is for MAD portfolio optimization. The `PortfolioCVaR` object is for CVaR portfolio optimization. The `Portfolio` object is for mean-variance portfolio optimization. In some cases, you might want to examine portfolio optimization problems according to different combinations of return and risk proxies. A common example is

that you want to do a MAD portfolio optimization and then want to work primarily with moments of portfolio returns. Suppose you set up a MAD portfolio optimization problem with:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD;
p = setAssetList(p, 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
```

To work with the same problem in a mean-variance framework, you can use the scenarios from the PortfolioMAD object to set up a Portfolio object so that `p` contains a MAD optimization problem and `q` contains a mean-variance optimization problem based on the same data.

```
q = Portfolio('AssetList', p.AssetList);
q = estimateAssetMoments(q, p.getScenarios);
q = setDefaultConstraints(q);

pwgt = estimateFrontier(p);
qwgt = estimateFrontier(q);
```

Note that since each object has a different risk proxy, it is not possible to compare results side by side. To obtain means and standard deviations of portfolio returns, you can use the methods associated with each object to obtain:

```
pret = estimatePortReturn(p, pwgt);
pstd = estimatePortStd(p, pwgt);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);
```

```
[pret, qret]
[pstd, qstd]
```

```
ans =
```

```
0.0592    0.0590
0.0730    0.0728
0.0868    0.0867
0.1006    0.1005
```



```
0.1145 0.1143
0.1283 0.1282
0.1421 0.1420
0.1559 0.1558
0.1697 0.1697
0.1835 0.1835
```

```
ans =
```

```
0.0767 0.0767
0.0829 0.0828
0.0989 0.0987
0.1208 0.1206
0.1461 0.1459
0.1732 0.1730
0.2042 0.2040
0.2453 0.2452
0.2929 0.2928
0.3458 0.3458
```

To produce comparable results, you can use the returns or risks from one portfolio optimization as target returns or risks for the other portfolio optimization.

```
qwgt = estimateFrontierByReturn(q, pret);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);
```

```
[pret, qret]
[pstd, qstd]
```

```
ans =
```

```
0.0592 0.0592
0.0730 0.0730
0.0868 0.0868
0.1006 0.1006
0.1145 0.1145
0.1283 0.1283
0.1421 0.1421
0.1559 0.1559
0.1697 0.1697
0.1835 0.1835
```

```
ans =  
  
    0.0767    0.0767  
    0.0829    0.0829  
    0.0989    0.0989  
    0.1208    0.1208  
    0.1461    0.1461  
    0.1732    0.1732  
    0.2042    0.2042  
    0.2453    0.2453  
    0.2929    0.2929  
    0.3458    0.3458
```

Now it is possible to compare standard deviations of portfolio returns from either type of portfolio optimization.

Troubleshooting MAD Portfolio Optimization Results

PortfolioMAD Object Destroyed When Modifying

If a PortfolioMAD object is destroyed when modifying, remember to pass an existing object into the constructor if you want to modify it, otherwise it creates a new object. See “Constructing the PortfolioMAD Object” on page 6-26 for details.

Matrix Incompatibility and "Non-Conformable" Errors

If you get matrix incompatibility or "non-conformable" errors, the representation of data in the tools follows a specific set of basic rules described in “Conventions for Representation of Data” on page 6-24.

Missing Data Estimation Fails

If asset return data has missing or NaN values, the method `simulateNormalScenariosByData` with the `'missingdata'` flag set to `true` may fail with either too many iterations or a singular covariance. To correct this problem, consider this:

- If you have asset return data with no missing or NaN values, you can compute a covariance matrix that may be singular without difficulties. If you have missing or NaN values in your data, the supported missing data feature requires that your covariance matrix must be positive-definite, i.e., nonsingular.
- `simulateNormalScenariosByData` uses default settings for the missing data estimation procedure that might not be appropriate for all problems.

In either case, you might want to estimate the moments of asset returns separately with either the ECM estimation functions such as `ecmmle` or with your own methods.

mad_optim_transform Errors

If you obtain optimization errors such as:

```
Error using mad_optim_transform (line 276)
Portfolio set appears to be either empty or unbounded. Check constraints.
```

```
Error in PortfolioMAD/estimateFrontier (line 64)
[AI, bI, AE, bE, lB, uB, f0, f, x0] = mad_optim_transform(obj);
```

or

```
Error using mad_optim_transform (line 281)
Cannot obtain finite lower bounds for specified portfolio set.
```

```
Error in PortfolioMAD/estimateFrontier (line 64)
[AI, bI, AE, bE, lB, uB, f0, f, x0] = mad_optim_transform(obj);
```

Since the portfolio optimization tools require a bounded portfolio set, these errors (and similar errors) can occur if your portfolio set is either empty and, if nonempty, unbounded. Specifically, the portfolio optimization algorithm requires that your portfolio set have at least a finite lower bound. The best way to deal with these problems is to use the validation methods in “Validate the MAD Portfolio Problem” on page 6-78. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

If you obtain efficient portfolios that do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between 0 and 1 and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the PortfolioMAD object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for

your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

See Also

`PortfolioMAD` | `checkFeasibility` | `estimateScenarioMoments`

Related Examples

- “Constructing the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints” on page 6-56
- “Estimate Efficient Portfolios” on page 6-83
- “Estimate Efficient Frontiers” on page 6-94
- “Asset Returns and Scenarios” on page 6-41

More About

- “PortfolioMAD Object” on page 6-17
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow”

Investment Performance Metrics

- “Overview of Performance Metrics” on page 7-2
- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Tracking Error” on page 7-9
- “Risk-Adjusted Return” on page 7-10
- “Sample and Expected Lower Partial Moments” on page 7-12
- “Maximum and Expected Maximum Drawdown” on page 7-15

Overview of Performance Metrics

In this section...
“Performance Metrics Types” on page 7-2
“Performance Metrics” on page 7-2

Performance Metrics Types

Sharpe first proposed a ratio of excess return to total risk as an investment performance metric. Subsequent work by Sharpe, Lintner, and Mossin extended these ideas to entire asset markets in what is called the Capital Asset Pricing Model (CAPM). Since the development of the CAPM, a variety of investment performance metrics has evolved.

This section presents four type of investment performance metrics:

- The first type of metrics are absolute investment performance metrics that are called “classic” metrics since they are based on the CAPM. They include the Sharpe ratio, the information ratio, and tracking error. To compute the Sharpe ratio from data, use `sharpe` to calculate the ratio for one or more asset return series. To compute the information ratio and associated tracking error, use `inforatio` to calculate these quantities for one or more asset return series.
- The second type of metrics are relative investment performance metrics to compute risk-adjusted returns. These metrics are also based on the CAPM and include Beta, Jensen's Alpha, the Security Market Line (SML), Modigliani and Modigliani Risk-Adjusted Return, and the Graham-Harvey measures. To calculate risk-adjusted alpha and return, use `portalpha`.
- The third type of metrics are alternative investment performance metrics based on lower partial moments. To calculate lower partial moments, use `lpm` for sample lower partial moments and `elpm` for expected lower partial moments.
- The fourth type of metrics are performance metrics based on maximum drawdown and expected maximum drawdown. To calculate maximum or expected maximum drawdowns, use `maxdrawdown` and `emaxdrawdown`.

Performance Metrics

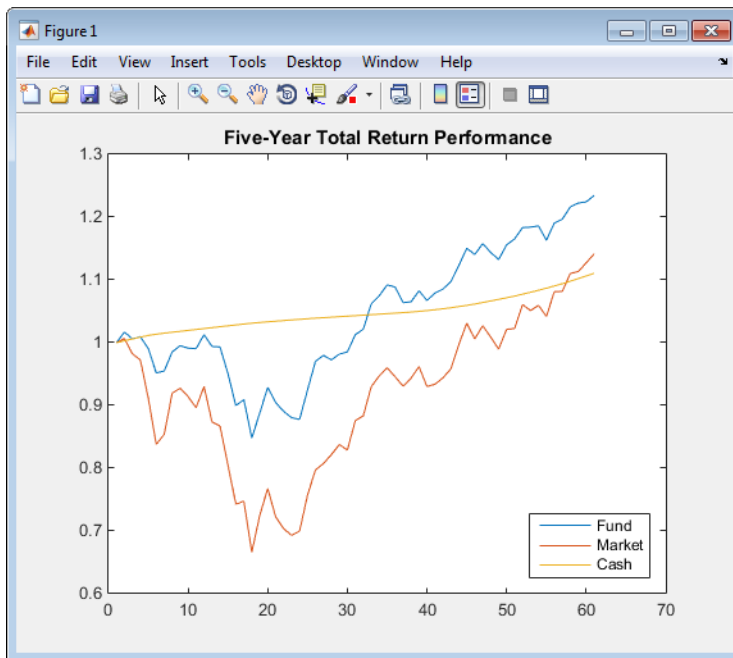
To illustrate the functions for investment performance metrics, you will work with three financial time series objects using performance data for:

- An actively managed, large-cap value mutual fund
- A large-cap market index
- 90-day Treasury bills

The data is monthly total return prices that cover a span of 5 years.

The following plot illustrates the performance of each series in terms of total returns to an initial \$1 invested at the start of this 5-year period:

```
load FundMarketCash
plot(TestData)
hold on
title('\bfive-Year Total Return Performance');
legend('Fund', 'Market', 'Cash', 'Location', 'SouthEast');
hold off
```



The mean (Mean) and standard deviation (Sigma) of returns for each series are

```
Returns = tick2ret(TestData);
Assets
```

```
Mean = mean>Returns)
Sigma = std>Returns, 1)
```

which gives the following result:

```
Assets =
  'Fund'   'Market'   'Cash'
Mean =
  0.0038   0.0030   0.0017
Sigma =
  0.0229   0.0389   0.0009
```

Note: Functions for investment performance metrics use total return price and total returns. To convert between total return price and total returns, use `ret2tick` and `tick2ret`.

Using the Sharpe Ratio

In this section...

“Introduction” on page 7-5

“Sharpe Ratio” on page 7-5

Introduction

The Sharpe ratio is the ratio of the excess return of an asset divided by the asset's standard deviation of returns. The Sharpe ratio has the form:

$$(\text{Mean} - \text{Riskless}) / \text{Sigma}$$

Here **Mean** is the mean of asset returns, **Riskless** is the return of a riskless asset, and **Sigma** is the standard deviation of asset returns. A higher Sharpe ratio is better than a lower Sharpe ratio. A negative Sharpe ratio indicates “anti-skill” since the performance of the riskless asset is superior. For more information, see [sharpe](#).

Sharpe Ratio

To compute the Sharpe ratio, the mean return of the cash asset is used as the return for the riskless asset. Thus, given asset return data and the riskless asset return, the Sharpe ratio is calculated with

```
load FundMarketCash
Returns = tick2ret(TestData);
Riskless = mean>Returns(:,3))
Sharpe = sharpe>Returns, Riskless)
```

which gives the following result:

```
Riskless =
    0.0017
Sharpe =
    0.0886    0.0315    0
```

The Sharpe ratio of the example fund is significantly higher than the Sharpe ratio of the market. As will be demonstrated with [portalalpha](#), this translates into a strong risk-adjusted return. Since the **Cash** asset is the same as **Riskless**, it makes sense that its

Sharpe ratio is 0. The Sharpe ratio was calculated with the mean of cash returns. It can also be calculated with the cash return series as input for the riskless asset

```
Sharpe = sharpe>Returns, Returns(:,3))
```

which gives the following result:

```
Sharpe =  
    0.0886    0.0315         0
```

When using the `Portfolio` object, you can use the `estimateMaxSharpeRatio` method to estimate an efficient portfolio that maximizes the Sharpe ratio. For more information, see “Efficient Portfolio that Maximizes Sharpe Ratio” on page 4-92.

Using the Information Ratio

In this section...

“Introduction” on page 7-7

“Information Ratio” on page 7-7

Introduction

Although originally called the “appraisal ratio” by Treynor and Black, the information ratio is the ratio of relative return to relative risk (known as “tracking error”). Whereas the Sharpe ratio looks at returns relative to a riskless asset, the information ratio is based on returns relative to a risky benchmark which is known colloquially as a “bogey.” Given an asset or portfolio of assets with random returns designated by `Asset` and a benchmark with random returns designated by `Benchmark`, the information ratio has the form:

$$\text{Mean}(\text{Asset} - \text{Benchmark}) / \text{Sigma}(\text{Asset} - \text{Benchmark})$$

Here `Mean(Asset - Benchmark)` is the mean of `Asset` minus `Benchmark` returns, and `Sigma(Asset - Benchmark)` is the standard deviation of `Asset` minus `Benchmark` returns. A higher information ratio is considered better than a lower information ratio. For more information, see `inforatio`.

Information Ratio

To calculate the information ratio using the example data, the mean return of the market series is used as the return of the benchmark. Thus, given asset return data and the riskless asset return, compute the information ratio with

```
load FundMarketCash
Returns = tick2ret(TestData);
Benchmark = Returns(:,2);
InfoRatio = inforatio>Returns, Benchmark)
```

which gives the following result:

```
InfoRatio =
    0.0432      NaN    -0.0315
```

Since the market series has no risk relative to itself, the information ratio for the second series is undefined (which is represented as NaN in MATLAB software). Its standard deviation of relative returns in the denominator is 0.

Tracking Error

In this section...

“Introduction” on page 7-9

“Tracking Error” on page 7-9

Introduction

Given an asset or portfolio of assets and a benchmark, the relative standard deviation of returns between the asset or portfolio of assets and the benchmark is called tracking error.

Tracking Error

The function `inforatio` computes tracking error and returns it as a second argument

```
load FundMarketCash
Returns = tick2ret(TestData);
Benchmark = Returns(:,2);
[InfoRatio, TrackingError] = inforatio>Returns, Benchmark)
```

which gives the following results:

```
InfoRatio =
    0.0432      NaN   -0.0315
TrackingError =
    0.0187         0    0.0390
```

Tracking error is a useful measure of performance relative to a benchmark since it is in units of asset returns. For example, the tracking error of 1.87% for the fund relative to the market in this example is reasonable for an actively managed, large-cap value fund.

Risk-Adjusted Return

In this section...

“Introduction” on page 7-10

“Risk-Adjusted Return” on page 7-10

Introduction

Risk-adjusted return either shifts the risk (which is the standard deviation of returns) of a portfolio to match the risk of a market portfolio or shifts the risk of a market portfolio to match the risk of a fund. According to the Capital Asset Pricing Model (CAPM), the market portfolio and a riskless asset are points on a Security Market Line (SML). The return of the resultant shifted portfolio, levered or unlevered, to match the risk of the market portfolio, is the risk-adjusted return. The SML provides another measure of risk-adjusted return, since the difference in return between the fund and the SML, return at the same level of risk.

Risk-Adjusted Return

Given our example data with a fund, a market, and a cash series, you can calculate the risk-adjusted return and compare it with the fund and market's mean returns

```
load FundMarketCash
Returns = tick2ret(TestData);
Fund = Returns(:,1);
Market = Returns(:,2);
Cash = Returns(:,3);
MeanFund = mean(Fund)
MeanMarket = mean(Market)

[MM, aMM] = portalpha(Fund, Market, Cash, 'MM')
[GH1, aGH1] = portalpha(Fund, Market, Cash, 'gh1')
[GH2, aGH2] = portalpha(Fund, Market, Cash, 'gh2')
[SML, aSML] = portalpha(Fund, Market, Cash, 'sm1')
```

which gives the following results:

```
MeanFund =
    0.0038
```

MeanMarket =

0.0030

MM =

0.0022

aMM =

0.0052

GH1 =

0.0013

aGH1 =

0.0025

GH2 =

0.0022

aGH2 =

0.0052

SML =

0.0013

aSML =

0.0025

Since the fund's risk is much less than the market's risk, the risk-adjusted return of the fund is much higher than both the nominal fund and market returns.

Sample and Expected Lower Partial Moments

In this section...

“Introduction” on page 7-12

“Sample Lower Partial Moments” on page 7-12

“Expected Lower Partial Moments” on page 7-13

Introduction

Use lower partial moments to examine what is colloquially known as “downside risk.” The main idea of the lower partial moment framework is to model moments of asset returns that fall below a minimum acceptable level of return. To compute lower partial moments from data, use `lpm` to calculate lower partial moments for multiple asset return series and for multiple moment orders. To compute expected values for lower partial moments under several assumptions about the distribution of asset returns, use `elpm` to calculate lower partial moments for multiple assets and for multiple orders.

Sample Lower Partial Moments

The following example demonstrates `lpm` to compute the zero-order, first-order, and second-order lower partial moments for the three time series, where the mean of the third time series is used to compute MAR (with the so-called risk-free rate).

```
load FundMarketCash
Returns = tick2ret(TestData);
Assets
MAR = mean>Returns(:,3))
LPM = lpm>Returns, MAR, [0 1 2])
```

which gives the following results:

```
Assets =
    'Fund'    'Market'    'Cash'
MAR =
    0.0017
LPM =
    0.4333    0.4167    0.6167
    0.0075    0.0140    0.0004
```



```
0.0003    0.0008    0.0000
```

The first row of LPM contains zero-order lower partial moments of the three series. The fund and market index fall below MAR about 40% of the time and cash returns fall below its own mean about 60% of the time.

The second row contains first-order lower partial moments of the three series. The fund and market have large expected shortfall returns relative to MAR by 75 and 140 basis points per month. On the other hand, cash underperforms MAR by about only 4 basis points per month on the downside.

The third row contains second-order lower partial moments of the three series. The square root of these quantities provides an idea of the dispersion of returns that fall below the MAR. The market index has a much larger variation on the downside when compared to the fund.

Expected Lower Partial Moments

To compare realized values with expected values, use `elpm` to compute expected lower partial moments based on the mean and standard deviations of normally distributed asset returns. The `elpm` function works with the mean and standard deviations for multiple assets and multiple orders.

```
load FundMarketCash
Returns = tick2ret(TestData);
MAR = mean>Returns(:,3))
Mean = mean>Returns)
Sigma = std>Returns, 1)
Assets
ELPM = elpm(Mean, Sigma, MAR, [0 1 2])
```

which gives the following results:

```
Assets =
    'Fund'    'Market'    'Cash'
ELPM =
    0.4647    0.4874    0.5000
    0.0082    0.0149    0.0004
    0.0002    0.0007    0.0000
```

Based on the moments of each asset, the expected values for lower partial moments imply better than expected performance for the fund and market and worse than

expected performance for cash. Note that this function works with either degenerate or nondegenerate normal random variables. For example, if cash were truly riskless, its standard deviation would be 0. You can examine the difference in expected shortfall.

```
RisklessCash = elpm(Mean(3), 0, MAR, 1)
```

which gives the following result:

```
RisklessCash =  
0
```

Maximum and Expected Maximum Drawdown

In this section...

“Introduction” on page 7-15

“Maximum Drawdown” on page 7-15

“Expected Maximum Drawdown” on page 7-18

Introduction

Maximum drawdown is the maximum decline of a series, measured as return, from a peak to a nadir over a period of time. Although additional metrics exist that are used in the hedge fund and commodity trading communities (see Pederson and Rudholm-Alfvén [20] in Appendix A), the original definition and subsequent implementation of these metrics is not yet standardized.

It is possible to compute analytically the expected maximum drawdown for a Brownian motion with drift (see Magdon-Ismail, Atiya, Pratap, and Abu-Mostafa [16] Appendix A). These results are used to estimate the expected maximum drawdown for a series that approximately follows a geometric Brownian motion.

Use `maxdrawdown` and `emaxdrawdown` to calculate the maximum and expected maximum drawdowns.

Maximum Drawdown

This example demonstrates how to compute the maximum drawdown (MaxDD) using our example data with a fund, a market, and a cash series:

```
load FundMarketCash
MaxDD = maxdrawdown(TestData)
```

which gives the following results:

```
MaxDD =
    0.1658    0.3381    0
```

The maximum drop in the given time period was of 16.58% for the fund series, and 33.81% for the market. There was no decline in the cash series, as expected, because the cash account never loses value.

`maxdrawdown` can also return the indices (`MaxDDIndex`) of the maximum drawdown intervals for each series in an optional output argument:

```
[MaxDD, MaxDDIndex] = maxdrawdown(TestData)
```

which gives the following results:

```
MaxDD =
```

```
    0.1658    0.3381         0
```

```
MaxDDIndex =
```

```
     2     2   NaN  
    18    18   NaN
```

The first two series experience their maximum drawdowns from the 2nd to the 18th month in the data. The indices for the third series are NaNs because it never has a drawdown.

The 16.58% value loss from month 2 to month 18 for the fund series is verified using the reported indices:

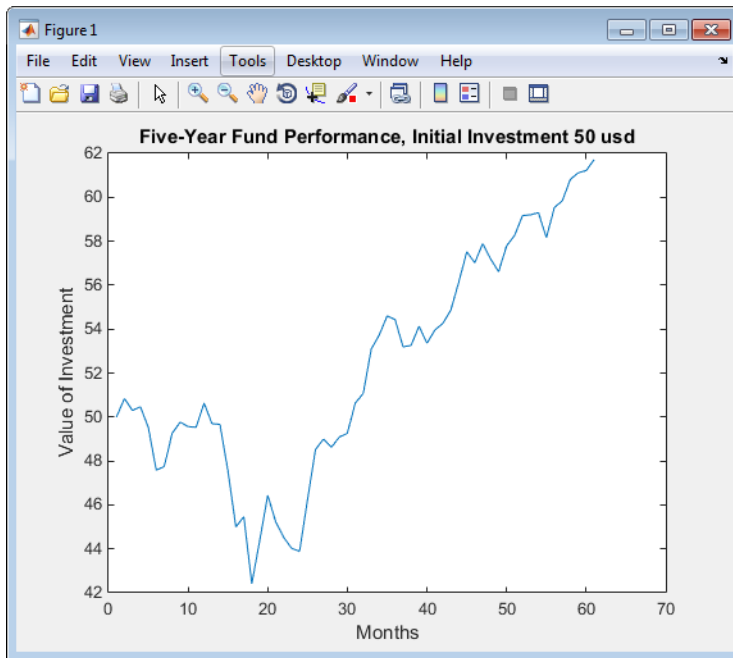
```
Start = MaxDDIndex(1,:);  
End = MaxDDIndex(2,:);  
(TestData(Start(1),1) - TestData(End(1),1))/TestData(Start(1),1)
```

```
ans =
```

```
    0.1658
```

Although the maximum drawdown is measured in terms of returns, `maxdrawdown` can measure the drawdown in terms of absolute drop in value, or in terms of log-returns. To contrast these alternatives more clearly, we work with the fund series assuming, an initial investment of 50 dollars:

```
Fund50 = 50*TestData(:,1);  
plot(Fund50);  
title('\bfive-Year Fund Performance, Initial Investment 50 usd');  
xlabel('Months');  
ylabel('Value of Investment');
```



First, we compute the standard maximum drawdown, which coincides with the results above because returns are independent of the initial amounts invested:

```
MaxDD50Ret = maxdrawdown(Fund50)
```

```
MaxDD50Ret =
```

```
0.1658
```

Next, we compute the maximum drop in value, using the `arithmetic` argument:

```
[MaxDD50Arith, Ind50Arith] = maxdrawdown(Fund50, 'arithmetic')
```

```
MaxDD50Arith =
```

```
8.4285
```

```
Ind50Arith =
```

```
2
```

18

The value of this investment was \$50.84 in month 2, but by month 18 the value was down to \$42.41, a drop of \$8.43. This is the largest loss in dollar value from a previous high in the given time period. In this case, the maximum drawdown period, 2nd to 18th month, is the same independently of whether drawdown is measured as return or as dollar value loss.

Last, we compute the maximum decline based on log-returns using the `geometric` argument. In this example, the log-returns result in a maximum drop of 18.13%, again from the 2nd to the 18th month, not far from the 16.58% obtained using standard returns.

```
[MaxDD50LogRet, Ind50LogRet] = maxdrawdown(Fund50, 'geometric')
```

```
MaxDD50LogRet =
```

```
0.1813
```

```
Ind50LogRet =
```

```
2  
18
```

Note, the last measure is equivalent to finding the arithmetic maximum drawdown for the log of the series:

```
MaxDD50LogRet2 = maxdrawdown(log(Fund50), 'arithmetic')
```

```
MaxDD50LogRet2 =
```

```
0.1813
```

Expected Maximum Drawdown

This example demonstrates using the log-return moments of the fund to compute the expected maximum drawdown (EMaxDD) and then compare it with the realized maximum drawdown (MaxDD).

```
load FundMarketCash  
logReturns = log(TestData(2:end,:) ./ TestData(1:end - 1,:));  
Mu = mean(logReturns(:,1));
```

```
Sigma = std(logReturns(:,1),1);  
T = size(logReturns,1);  
  
MaxDD = maxdrawdown(TestData(:,1),'geometric')  
EMaxDD = emaxdrawdown(Mu-0.5*Sigma^2, Sigma, T)
```

which gives the following results:

```
MaxDD =  
  
    0.1813  
  
EMaxDD =  
  
    0.1588
```

The drawdown observed in this time period is above the expected maximum drawdown. There is no contradiction here. The expected maximum drawdown is not an upper bound on the maximum losses from a peak, but an estimate of their average, based on a geometric Brownian motion assumption.

Credit Risk Analysis

- “Credit Rating by Bagging Decision Trees” on page 8-2
- “Estimation of Transition Probabilities” on page 8-21
- “Forecasting Corporate Default Rates” on page 8-39
- “Credit Quality Thresholds” on page 8-69
- “About Credit Scorecards” on page 8-73
- “Credit Scorecard Modeling Workflow” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-80
- “Case Study for a Credit Scorecard Analysis” on page 8-87

Credit Rating by Bagging Decision Trees

This example shows how to build an automated credit rating tool.

One of the fundamental tasks in credit risk management is to assign a credit grade to a borrower. Grades are used to rank customers according to their perceived creditworthiness: better grades mean less risky customers; similar grades mean similar level of risk. Grades come in two categories: credit ratings and credit scores. Credit ratings are a small number of discrete classes, usually labeled with letters, such as 'AAA', 'BB-', etc. Credit scores are numeric grades such as '640' or '720'. Credit grades are one of the key elements in regulatory frameworks, such as Basel II (see Basel Committee on Banking Supervision [3]).

Assigning a credit grade involves analyzing information on the borrower. If the borrower is an individual, information of interest could be the individual's income, outstanding debt (mortgage, credit cards), household size, residential status, etc. For corporate borrowers, one may consider certain financial ratios (e.g., sales divided by total assets), industry, etc. Here, we refer to these pieces of information about a borrower as *features* or *predictors*. Different institutions use different predictors, and they may also have different rating classes or score ranges to rank their customers. For relatively small loans offered to a large market of potential borrowers (e.g., credit cards), it is common to use credit scores, and the process of grading a borrower is usually automated. For larger loans, accessible to small- to medium-sized companies and larger corporations, credit ratings are usually used, and the grading process may involve a combination of automated algorithms and expert analysis.

There are rating agencies that keep track of the creditworthiness of companies. Yet, most banks develop an internal methodology to assign credit grades for their customers. Rating a customer internally can be a necessity if the customer has not been rated by a rating agency, but even if a third-party rating exists, an internal rating offers a complementary assessment of a customer's risk profile.

This example shows how MATLAB® can help with the automated stage of a credit rating process. In particular, we take advantage of one of the statistical learning tools readily available in Statistics Toolbox™, a classification algorithm known as a *bagged decision tree*.

We assume that historical information is available in the form of a data set where each record contains the features of a borrower and the credit rating that was assigned to it. These may be internal ratings, assigned by a committee that followed policies and

procedures already in place. Alternatively, the ratings may come from a rating agency, whose ratings are being used to "jump start" a new internal credit rating system.

The existing historical data is the starting point, and it is used to *train* the bagged decision tree that will automate the credit rating. In the vocabulary of statistical learning, this training process falls in the category of *supervised learning*. The classifier is then used to assign ratings to new customers. In practice, these automated or *predicted* ratings would most likely be regarded as tentative, until a credit committee of experts reviews them. The type of classifier we use here can also facilitate the revision of these ratings, because it provides a measure of certainty for the predicted ratings, a *classification score*.

In practice, one needs to train a classifier first, then use it to assign a credit rating to new customers, and finally one also needs to *profile* or *evaluate the quality* or accuracy of the classifier, a process also known as *validation* or *back-testing*. We discuss some readily available back-testing tools, as well.

Loading the Existing Credit Rating Data

We load the historical data from the comma-delimited text file `CreditRating_Historical.dat`. We choose to work with text files here, but users with access to Database Toolbox™ can certainly load this information directly from a database.

The data set contains financial ratios, industry sector, and credit ratings for a list of corporate customers. This is simulated, not real data. The first column is a customer ID. Then we have five columns of financial ratios. These are the same ratios used in Altman's z-score (see Altman [1]; see also Loeffler and Posch [4] for a related analysis).

- Working capital / Total Assets (WC_TA)
- Retained Earnings / Total Assets (RE_TA)
- Earnings Before Interests and Taxes / Total Assets (EBIT_TA)
- Market Value of Equity / Book Value of Total Debt (MVE_BVTD)
- Sales / Total Assets (S_TA)

Next, we have an industry sector label, an integer value ranging from 1 to 12. The last column has the credit rating assigned to the customer. We load the data into a `table` array.

```
creditDS = readtable('CreditRating_Historical.dat');
```

We copy the features into a matrix X , and the corresponding classes, the ratings, into a vector Y . This is not a required step, since we could access this information directly from the `dataset` or `table` array, but we do it here to simplify some repeated function calls below.

The features to be stored in the matrix X are the five financial ratios, and the industry label. `Industry` is a categorical variable, *nominal* in fact, because there is no ordering in the industry sectors. The response variable, the credit ratings, is also categorical, though this is an *ordinal* variable, because, by definition, ratings imply a *ranking* of creditworthiness. We can use this variable "as is" to train our classifier. Here we choose to copy it into an *ordinal array* because this way the outputs come out in the natural order of the ratings and are easier to read. The ordering of the ratings is established by the cell array we pass as a third argument in the definition of Y . The credit ratings can also be mapped into numeric values, which can be useful to try alternative methods to analyze the data (e.g., regression). It is always recommended to try different methods in practice.

```
X = [creditDS.WC_TA creditDS.RE_TA creditDS.EBIT_TA creditDS.MVE_BVTD...  
      creditDS.S_TA creditDS.Industry];  
Y = ordinal(creditDS.Rating);
```

We use the predictors X and the response Y to fit a particular type of classification ensemble called a *bagged decision tree*. "Bagging," in this context, stands for "bootstrap aggregation." The methodology consists in generating a number of sub-samples, or *bootstrap replicas*, from the data set. These sub-samples are randomly generated, sampling with replacement from the list of customers in the data set. For each replica, a decision tree is grown. Each decision tree is a trained classifier on its own, and could be used in isolation to classify new customers. The predictions of two trees grown from two different bootstrap replicas may be different, though. The ensemble *aggregates* the predictions of all the decision trees that are grown for all the bootstrap replicas. If the majority of the trees predict one particular class for a new customer, it is reasonable to consider that prediction to be more robust than the prediction of any single tree alone. Moreover, if a different class is predicted by a smaller set of trees, that information is useful, too. In fact, the proportion of trees that predict different classes is the basis for the *classification scores* that are reported by the ensemble when classifying new data.

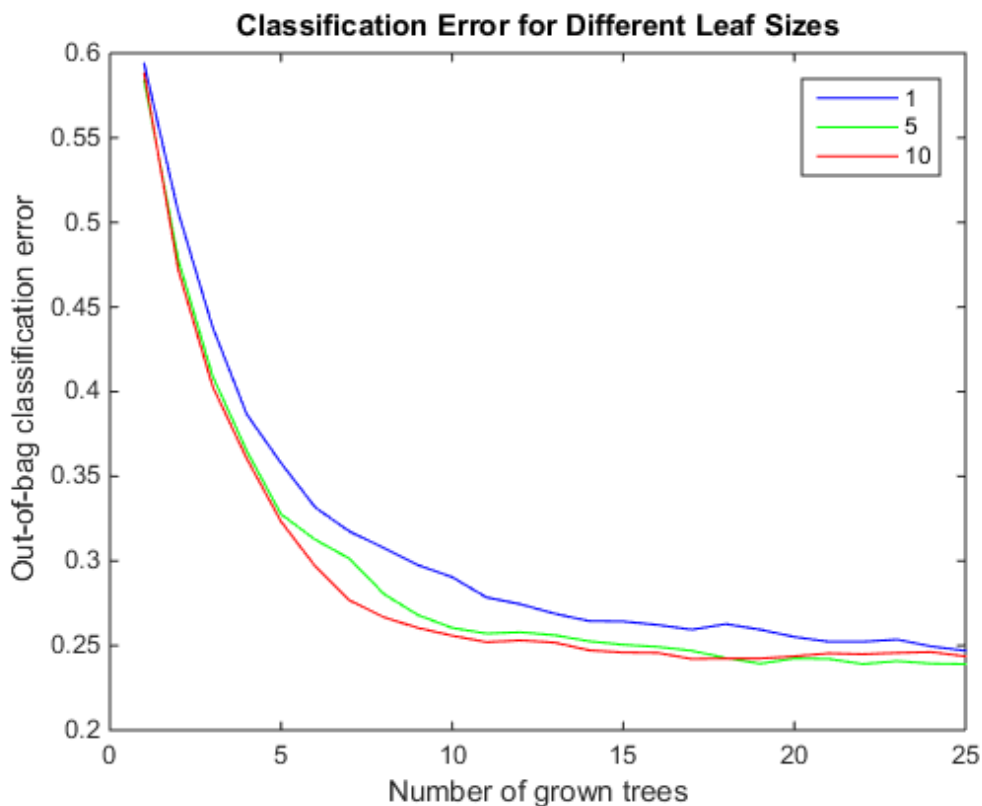
Constructing the Tree Bagger

The first step to construct our classification ensemble will be to find a good leaf size for the individual trees; here we try sizes of 1, 5 and 10. (See Statistics Toolbox

documentation for more on `TreeBagger`.) We start with a small number of trees, 25 only, because we mostly want to compare the initial trend in the classification error for different leaf sizes. For reproducibility and fair comparisons, we reinitialize the random number generator, which is used to sample with replacement from the data, each time we build a classifier.

```
leaf = [1 5 10];
nTrees = 25;
rng(9876, 'twister');
savedRng = rng; % save the current RNG settings

color = 'bgr';
for ii = 1:length(leaf)
    % Reinitialize the random number generator, so that the
    % random samples are the same for each leaf size
    rng(savedRng);
    % Create a bagged decision tree for each leaf size and plot out-of-bag
    % error 'oobError'
    b = TreeBagger(nTrees,X,Y, 'OOBPred','on',...
                  'CategoricalPredictors',6,...
                  'MinLeaf',leaf(ii));
    plot(b.oobError,color(ii));
    hold on;
end
xlabel('Number of grown trees');
ylabel('Out-of-bag classification error');
legend({'1', '5', '10'}, 'Location', 'NorthEast');
title('Classification Error for Different Leaf Sizes');
hold off;
```

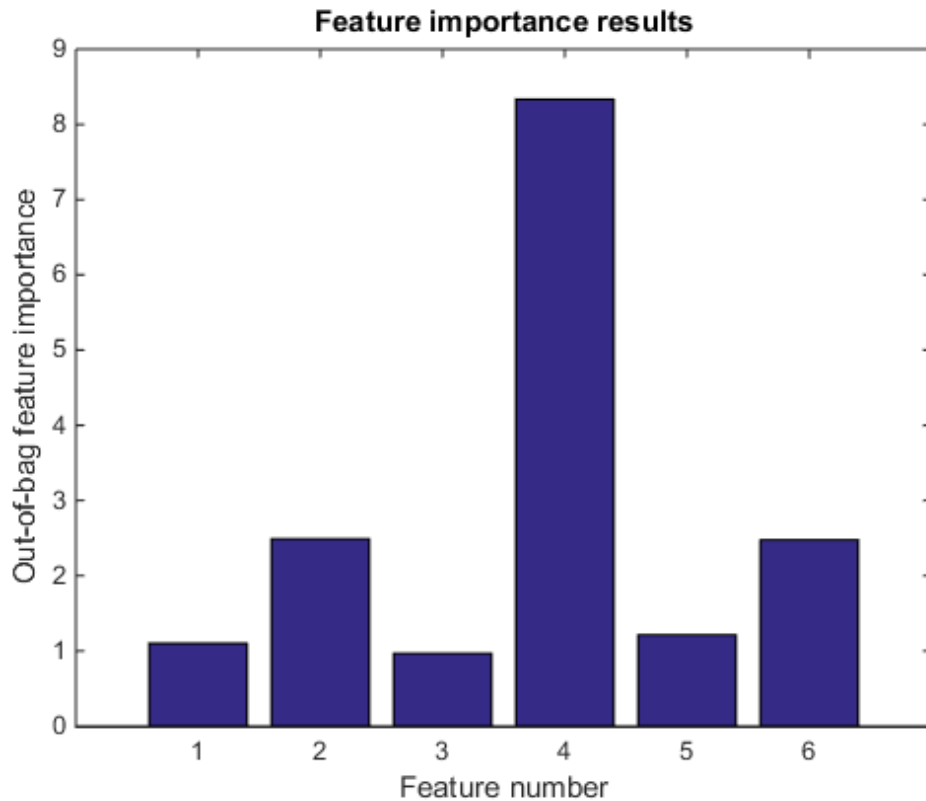


The errors are comparable for the three leaf-size options. We will therefore work with a leaf size of 10, because it results in leaner trees and more efficient computations.

Note that we did not have to split the data into *training* and *test* subsets. This is done internally, it is implicit in the sampling procedure that underlies the method. At each bootstrap iteration, the bootstrap replica is the training set, and any customers left out ("out-of-bag") are used as test points to estimate the out-of-bag classification error reported above.

Next, we want to find out whether all the features are important for the accuracy of our classifier. We do this by turning on the *feature importance* measure (`oobvarimp`), and plot the results to visually find the most important features. We also try a larger number of trees now, and store the classification error, for further comparisons below.

```
nTrees = 50;  
leaf = 10;  
rng(savedRng);  
b = TreeBagger(nTrees,X,Y,'OOBVarImp','on',...  
               'CategoricalPredictors',6,...  
               'MinLeaf',leaf);  
  
bar(b.OOBPermutedVarDeltaError);  
xlabel('Feature number');  
ylabel('Out-of-bag feature importance');  
title('Feature importance results');  
  
oobErrorFullX = b.oobError;
```



Features 2, 4 and 6 stand out from the rest. Feature 4, market value of equity / book value of total debt (MVE_BVTD), is the most important predictor for this data set. This ratio is closely related to the predictors of creditworthiness in structural models, such as Merton's model [5], where the value of the firm's equity is compared to its outstanding debt to determine the default probability.

Information on the industry sector, feature 6 (Industry), is also relatively more important than other variables to assess the creditworthiness of a firm for this data set.

Although not as important as MVE_BVTD, feature 2, retained earnings / total assets (RE_TA), stands out from the rest. There is a correlation between retained earnings and the age of a firm (the longer a firm has existed, the more earnings it can accumulate, in general), and in turn the age of a firm is correlated to its creditworthiness (older firms tend to be more likely to survive in tough times).

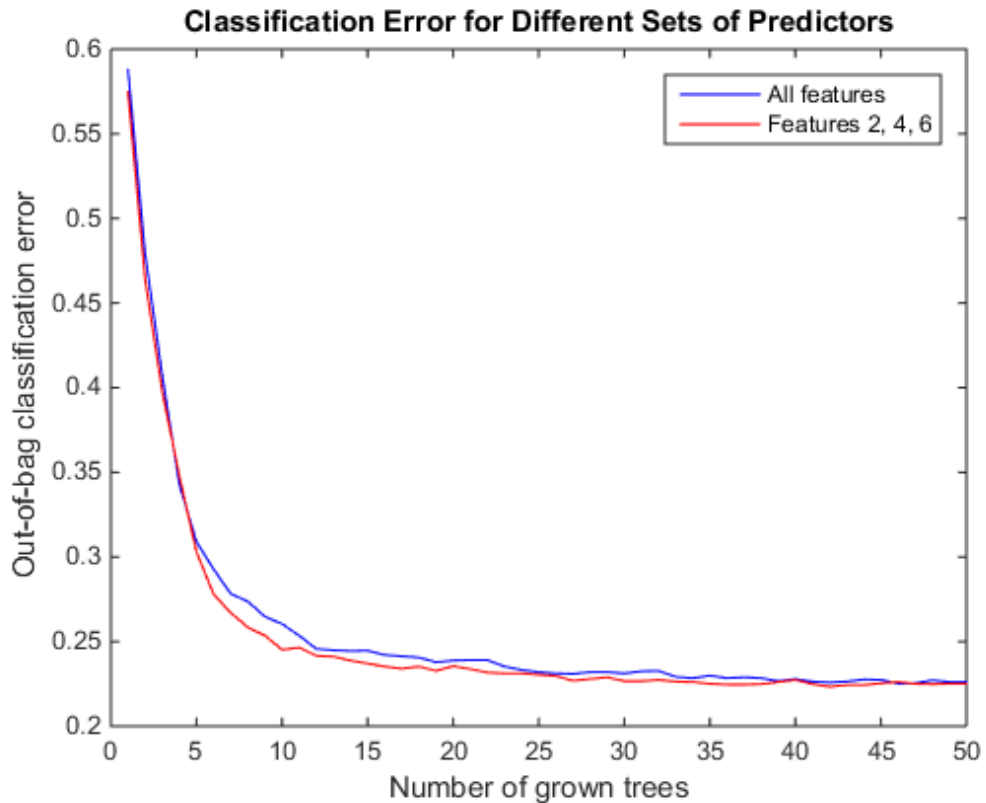
Let us fit a new classification ensemble using only predictors RE_TA, MVE_BVTD, and Industry. We compare its classification error with the previous classifier, which uses all features.

```
X = [creditDS.RE_TA creditDS.MVE_BVTD creditDS.Industry];

rng(savedRng);
b = TreeBagger(nTrees,X,Y, 'OOBPred', 'on', ...
               'CategoricalPredictors', 3, ...
               'MinLeaf', leaf);

oobErrorX246 = b.oobError;

plot(oobErrorFullX, 'b');
hold on;
plot(oobErrorX246, 'r');
xlabel('Number of grown trees');
ylabel('Out-of-bag classification error');
legend({'All features', 'Features 2, 4, 6'}, 'Location', 'NorthEast');
title('Classification Error for Different Sets of Predictors');
hold off;
```

The accuracy of the classification does not deteriorate significantly when we remove the features with relatively low importance (1, 3, and 5), so we will use the more parsimonious classification ensemble for our predictions.

In this example, we have started with a set of six features only, and used the feature importance measure of the classifier, and the out-of-bag classification error as criteria to screen out three of the variables. Feature selection can be a time consuming process when the initial set of potential predictors contains dozens of variables. Besides the tools we have used here (variable importance and a "visual" comparison of out-of-bag errors), other variable selection tools in Statistics Toolbox can be helpful for these types of analyses (see documentation). However, in the end, a successful feature selection process requires a combination of quantitative tools and an analyst's judgement.

For example, the variable importance measure we used here is a ranking mechanism that estimates the relative impact of a feature by measuring how much the predictive accuracy of the classifier deteriorates when this feature's values are randomly permuted. The idea is that when the feature in question adds little to the predictive power of the classifier, using altered (in this case permuted) values should not impact the classification results. Relevant information, on the other hand, cannot be randomly swapped without degrading the predictions. Now, if two highly correlated features are important, they will both rank high in this analysis. In that case, keeping one of these features should suffice for accurate classifications, but one would not know that from the ranking results alone. One would have to check the correlations separately, or use an expert's judgement. That is to say, tools like variable importance or `sequentialfs` can greatly help for feature selection, but an analyst's judgment is a key piece in this process.

At this point, the classifier could be saved (e.g., `save classifier.mat b`), to be loaded in a future session (`load classifier`) to classify new customers. For efficiency, it is recommended to keep a compact version of the classifier once the training process is finished.

```
b = b.compact;
```

Classifying New Data

Here we use the previously constructed classification ensemble to assign credit ratings to new customers. Because the ratings of existing customers need to be reviewed, too, on a regular basis, especially when their financial information has substantially changed, the data set could also contain a list of existing customers under review. We start by loading the new data.

```
newDS = readtable('CreditRating_NewCompanies.dat');
```

To predict the credit rating for this new data, we call the `predict` method on the classifier. The method returns two arguments, the predicted class and the classification score. We certainly want to get both output arguments, since the classification scores contain information on how certain the predicted ratings seem to be. We could copy variables `RE_TA`, `MVE_BVTD` and `Industry` into a matrix `X`, as before, but since we will make only one call to `predict`, we can skip this step and use `newDS` directly.

```
[predClass,classifScore] = b.predict([newDS.RE_TA newDS.MVE_BVTD...  
    newDS.Industry]);
```

At this point, we can create a report. Here we only display on the screen a small report for the first three customers, for illustration purposes, but MATLAB's deployment tools could greatly improve the workflow here. For example, using MATLAB Builder™ JA,

credit analysts could run this classification remotely, using a web browser, and get a report, without even having MATLAB on their desktops.

```

for i = 1:3
    fprintf('Customer %d:\n',newDS.ID(i));
    fprintf('    RE/TA    = %5.2f\n',newDS.RE_TA(i));
    fprintf('    MVE/BVTD = %5.2f\n',newDS.MVE_BVTD(i));
    fprintf('    Industry = %2d\n',newDS.Industry(i));
    fprintf('    Predicted Rating : %s\n',predClass{i});
    fprintf('    Classification score : \n');
    for j = 1:length(b.ClassNames)
        if (classifScore(i,j)>0)
            fprintf('        %s : %5.4f \n',b.ClassNames{j},classifScore(i,j));
        end
    end
end
end

```

```

Customer 60644:
    RE/TA    = 0.22
    MVE/BVTD = 2.40
    Industry = 6
    Predicted Rating : AA
    Classification score :
        A : 0.2397
        AA : 0.7367
        AAA : 0.0203
        BBB : 0.0032

```

```

Customer 33083:
    RE/TA    = 0.24
    MVE/BVTD = 1.51
    Industry = 4
    Predicted Rating : BBB
    Classification score :
        A : 0.0707
        BB : 0.0038
        BBB : 0.9255

```

```

Customer 63830:
    RE/TA    = 0.18
    MVE/BVTD = 1.69
    Industry = 7
    Predicted Rating : A
    Classification score :
        A : 0.6635
        AA : 0.0124
        B : 0.0024

```

```
BBB : 0.3218
```

Keeping records of the predicted ratings and corresponding scores can be useful for periodic assessments of the quality of the classifier. We store this information here in the table array `predDS`.

```
classnames = b.ClassNames;  
predDS = [table(newDS.ID,predClass),array2table(classifScore)];  
predDS.Properties.VariableNames = {'ID','PredRating',classnames{:}};
```

This information could be saved, for example, to a comma-delimited text file `PredictedRatings.dat` using the command

```
export(predDS,'file','PredictedRatings.dat','delimiter',';')
```

or written directly to a database using Database Toolbox.

Back-Testing: Profiling the Classification Process

Validation or *back-testing* is the process of profiling or assessing the quality of the credit ratings. There are many different measures and tests related to this task (see, for example, Basel Committee on Banking Supervision [2]). Here, we focus on the following two questions:

- How accurate are the predicted ratings, as compared to the actual ratings? Here "predicted ratings" refers to those obtained from the automated classification process, and "actual ratings" to those assigned by a credit committee that puts together the predicted ratings and their classification scores, and other pieces of information, such as news and the state of the economy to determine a final rating.
- How well do the actual ratings rank customers according to their creditworthiness? This is done in an *ex-post* analysis performed, for example, one year later, when it is known which companies defaulted during the year.

The file `CreditRating_ExPost.dat` contains "follow up" data on the same companies considered in the previous section. It contains the actual ratings that the committee assigned to these companies, as well as a "default flag" that indicates whether the corresponding company defaulted within one year of the rating process (if 1) or not (if 0).

```
exPostDS = readtable('CreditRating_ExPost.dat');
```

If this were a new MATLAB session, besides reading the follow up data from `CreditRating_ExPost.dat` we would need to load the predicted ratings information stored in `predDS`, for example, with the command

```
predDS = readtable('PredictedRatings.dat')
```

Comparing predicted ratings vs. actual ratings. The rationale to train an automated classifier is to expedite the work of the credit committee. The more accurate the predicted ratings are, the less time the committee has to spend reviewing the predicted ratings. So it is conceivable that the committee wants to have regular checks on how closely the predicted ratings match the final ratings they assign, and to recommend re-training the automated classifier (and maybe include new features, for example) if the mismatch seems concerning.

The first tool we can use to compare predicted vs. actual ratings is a *confusion matrix*, readily available in Statistics Toolbox:

```
C = confusionmat(exPostDS.Rating,predDS.PredRating,...
    'order',{'AAA' 'AA' 'A' 'BBB' 'BB' 'B' 'CCC'})
```

```
C =
```

203	13	0	0	0	0	0
8	110	20	0	0	0	0
0	18	155	31	0	0	0
0	1	26	246	50	0	0
0	0	0	37	229	19	0
0	0	0	0	46	45	2
0	0	0	0	0	9	43

The rows in **C** correspond to the actual ratings, and the columns to the predicted ratings. The amount in the position (i, j) in this matrix indicates how many customers received an actual rating i and were predicted as rating j . For example, position $(3, 2)$ tells us how many customers received a rating of 'A' by the credit committee, but were predicted as 'AA' with the automated classifier. One can also present this matrix in percentage form with a simple transformation:

```
Cperc = diag(sum(C,2))\C
```

```
Cperc =
```

0.9398	0.0602	0	0	0	0	0
0.0580	0.7971	0.1449	0	0	0	0
0	0.0882	0.7598	0.1520	0	0	0

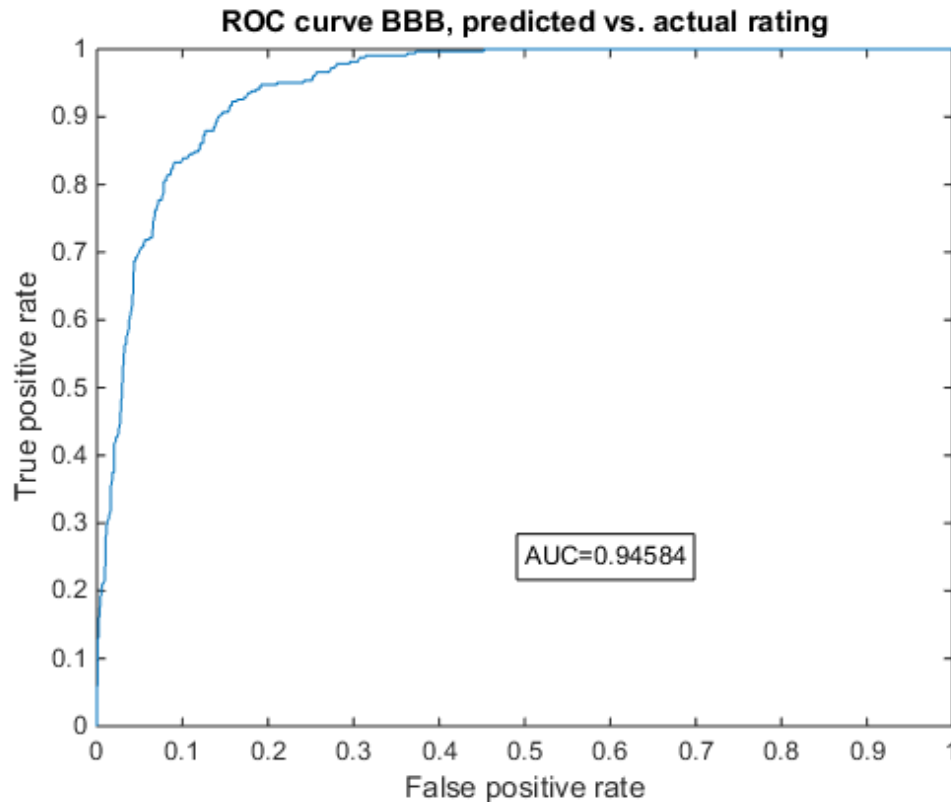
0	0.0031	0.0805	0.7616	0.1548	0	0
0	0	0	0.1298	0.8035	0.0667	0
0	0	0	0	0.4946	0.4839	0.0215
0	0	0	0	0	0.1731	0.8269

Good agreement between the predicted and the actual ratings would result in values in the main diagonal that dominate the rest of the values in a row, ideally values close to 1. In this case, we actually see an important disagreement for 'B,' since about half of the customers that were rated as 'B' by the credit committee had been predicted as 'BB' by the automated classifier. On the other hand, it is good to see that ratings differ in at most one notch in most cases, with the only exception of 'BBB.'

A confusion matrix could also be used to compare the internal ratings assigned by the institution against third-party ratings; this is often done in practice.

For each specific rating, we can compute yet another measure of agreement between predicted and actual ratings. We can build a *Receiver Operating Characteristic (ROC) curve* using the `perfcurve` function from Statistics Toolbox, and check the *area under the curve (AUC)*. The `perfcurve` function takes as an argument the actual ratings, which are our benchmark, the standard we are comparing against, and the 'BBB' classification scores determined by the automated process. Let us build a ROC and calculate the AUC for rating 'BBB' in our example.

```
[xVal,yVal,~,auc] = perfcurve(exPostDS.Rating,predDS.BBB, 'BBB');  
plot(xVal,yVal);  
xlabel('False positive rate');  
ylabel('True positive rate');  
text(0.5,0.25,strcat('AUC=',num2str(auc)), 'EdgeColor', 'k');  
title('ROC curve BBB, predicted vs. actual rating');
```



Here is an explanation of how the ROC is built. Recall that for each customer the automated classifier returns a classification score for each of the credit ratings, in particular, for 'BBB,' which can be interpreted as how likely it is that this particular customer should be rated 'BBB.' In order to build the ROC curve, one needs to vary the *classification threshold*. That is, the minimum score to classify a customer as 'BBB.' In other words, if the threshold is t , we only classify customers as 'BBB' if their 'BBB' score is greater than or equal to t . For example, suppose that company XYZ had a 'BBB' score of 0.87. If the actual rating of XYZ (the information in `exPostDS.Rating`) is 'BBB,' then XYZ would be correctly classified as 'BBB' for any threshold of up to 0.87. This would be a *true positive*, and it would increase what is called the *sensitivity* of the classifier. For any threshold greater than 0.87, this company would not receive a 'BBB' rating, and we would have a *false negative* case. To complete the description, suppose now that XYZ's actual rating is 'BB.' Then it would be correctly rejected as a 'BBB' for thresholds of

more than 0.87, becoming a *true negative*, and thus increasing the so called *specificity* of the classifier. However, for thresholds of up to 0.87, it would become a *false positive* (it would be classified as 'BBB,' when it actually is a 'BB'). The ROC curve is constructed by plotting the proportion of true positives (sensitivity), versus false positives (1-specificity), as the threshold varies from 0 to 1.

The AUC, as its name indicates, is the area under the ROC curve. The closer the AUC is to 1, the more accurate the classifier (a perfect classifier would have an AUC of 1). In this example, the AUC seems high enough, but it would be up to the committee to decide which level of AUC for the ratings should trigger a recommendation to improve the automated classifier.

Comparing actual ratings vs. defaults in the following year. A common tool used to assess the ranking of customers implicit in the credit ratings is the *Cumulative Accuracy Profile (CAP)*, and the associated *accuracy ratio* measure. The idea is to measure the relationship between the credit ratings assigned and the number of defaults observed in the following year. One would expect that fewer defaults are observed for better rating classes. If the default rate were the same for all ratings, the rating system would be no different from a naive (and useless) classification system in which customers were randomly assigned a rating, independently of their creditworthiness.

It is not hard to see that the `perfcurve` function can also be used to construct the CAP. The standard we compare against is not a rating, as before, but the default flag that we loaded from the `CreditRating_ExPost.dat` file. The score we use is a "dummy score" that indicates the ranking in creditworthiness implicit in the list of ratings. The dummy score only needs to satisfy that better ratings get lower dummy scores (they are "less likely to have a default flag of 1"), and that any two customers with the same rating get the same dummy score. A default probability could be passed as a score, of course, but we do not have default probabilities here, and in fact *we do not need to have estimates of the default probabilities to construct the CAP*, because we are not validating default probabilities. All we are assessing with this tool is how well the ratings *rank* customers according to their creditworthiness.

Usually, the CAP of the rating system under consideration is plotted together with the CAP of the "perfect rating system." The latter is a hypothetical credit rating system for which the lowest rating includes all the defaulters, and no other customers. The area under this perfect curve is the maximum possible AUC attainable by a rating system. By convention, the AUC is adjusted for CAPs to subtract the area under the *naive system's* CAP, that is, the CAP of the system that randomly assigns ratings to customers. The naive system's CAP is simply a straight line from the origin to (1,1), with an AUC of 0.5. The *accuracy ratio* for a rating system is then defined as the ratio of the adjusted AUC

(AUC of the system in consideration minus AUC of the naive system) to the maximum accuracy (AUC of the perfect system minus AUC of the naive system).

```

ratingsList = {'AAA' 'AA' 'A' 'BBB' 'BB' 'B' 'CCC'};
Nratings = length(ratingsList);
dummyDelta = 1/(Nratings+1);
dummyRank = linspace(dummyDelta,1-dummyDelta,Nratings)';

D = exPostDS.Def_tplus1;
fracTotDef = sum(D)/length(D);
maxAcc = 0.5 - 0.5 * fracTotDef;

R = double(ordinal(exPostDS.Rating,[],ratingsList));
S = dummyRank(R);
[xVal,yVal,~,auc] = perfcurve(D,S,1);

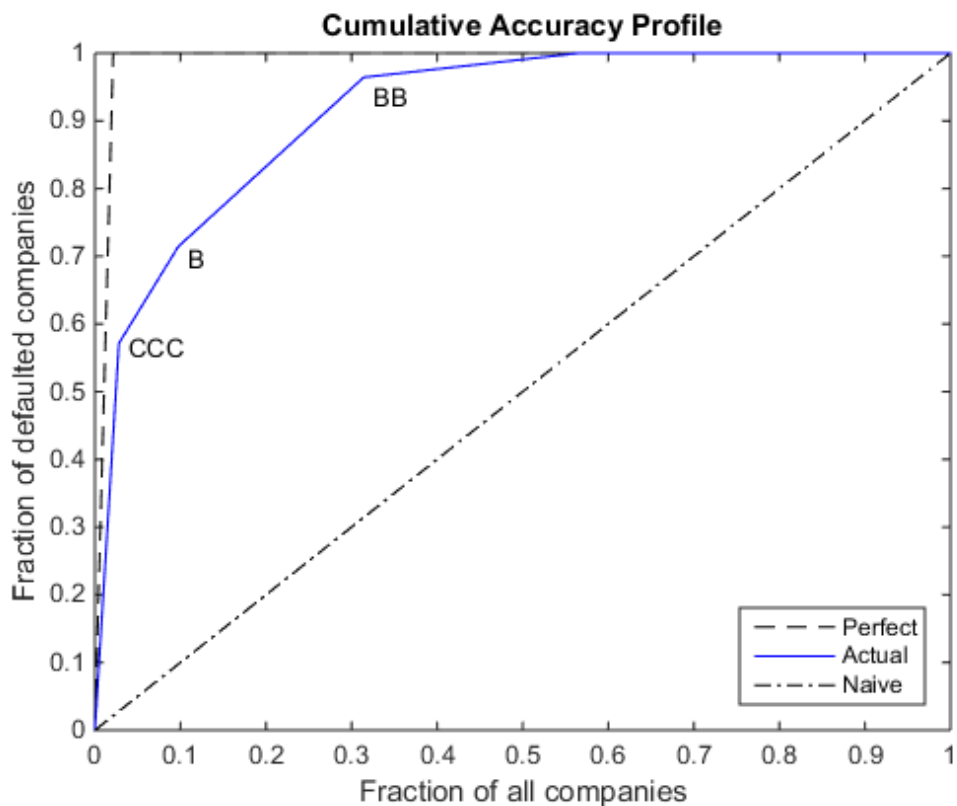
accRatio = (auc-0.5)/maxAcc;
fprintf('Accuracy ratio for actual ratings: %5.3f\n',accRatio);

xPerfect(1) = 0; xPerfect(2) = fracTotDef; xPerfect(3) = 1;
yPerfect(1) = 0; yPerfect(2) = 1; yPerfect(3) = 1;
xNaive(1) = 0; xNaive(2) = 1;
yNaive(1) = 0; yNaive(2) = 1;

plot(xPerfect,yPerfect,'--k',xVal,yVal,'b',xNaive,yNaive,'-.k');
xlabel('Fraction of all companies');
ylabel('Fraction of defaulted companies');
title('Cumulative Accuracy Profile');
legend({'Perfect','Actual','Naive'},'Location','SouthEast');
text(xVal(2)+0.01,yVal(2)-0.01,'CCC')
text(xVal(3)+0.01,yVal(3)-0.02,'B')
text(xVal(4)+0.01,yVal(4)-0.03,'BB')

Accuracy ratio for actual ratings: 0.850

```



The key to reading the information of the CAP is in the "kinks," labeled in the plot for ratings 'CCC,' 'B,' and 'BB.' For example, the second kink is associated with the second lowest rating, 'B,' and it is located at (0.097, 0.714). This means that 9.7% of the customers were ranked 'B' or lower, and they account for 71.4% of the defaults observed.

In general, the accuracy ratio should be treated as a relative, rather than an absolute measure. For example, we can add the CAP of the predicted ratings in the same plot, and compute its accuracy ratio to compare it with the accuracy ratio of the actual ratings.

```
Rpred = double(ordinal(predDS.PredRating,[],ratingsList));
Spred = dummyRank(Rpred);
[xValPred,yValPred,~,aucPred] = perfcurve(D,Spred,1);

accRatioPred = (aucPred-0.5)/maxAcc;
```

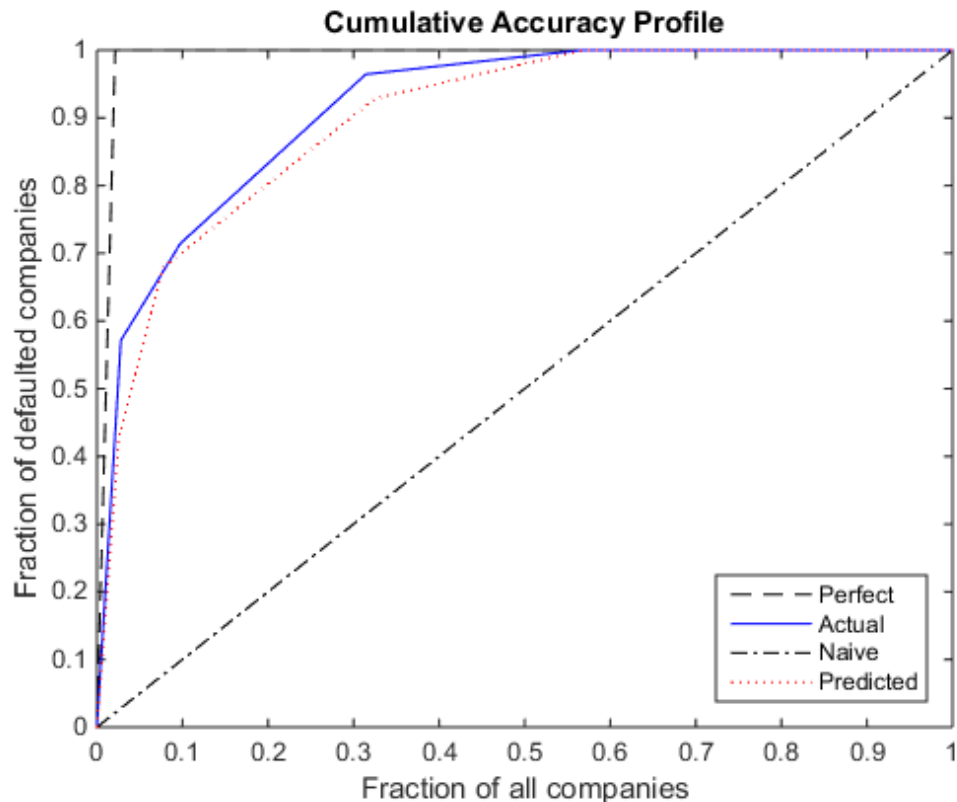
```

fprintf('Accuracy ratio for predicted ratings: %5.3f\n',accRatioPred);

plot(xPerfect,yPerfect,'--k',xVal,yVal,'b',xNaive,yNaive,'-.k',...
     xValPred,yValPred,':r');
xlabel('Fraction of all companies');
ylabel('Fraction of defaulted companies');
title('Cumulative Accuracy Profile');
legend({'Perfect','Actual','Naive','Predicted'},'Location','SouthEast');

Accuracy ratio for predicted ratings: 0.816

```



The accuracy ratio of the predicted rating is smaller, and its CAP is mostly below the CAP of the actual rating. This is reasonable, since the actual ratings are assigned by the credit committees that take into consideration the predicted ratings *and* extra information that can be important to fine-tune the ratings.

Final Remarks

MATLAB offers a wide range of machine learning tools, besides bagged decision trees, that can be used in the context of credit rating. In Statistics Toolbox you can find classification tools such as discriminant analysis and naive Bayes classifiers. MATLAB also offers Neural Networks Toolbox™. Also, Database Toolbox and MATLAB's deployment tools such as MATLAB Builder JA may provide you with more flexibility to adapt the workflow presented here to your own preferences and needs.

No probabilities of default have been computed here. For credit ratings, the probabilities of default are usually computed based on credit-rating migration history. See the transprob reference page in Financial Toolbox™ for more information.

Bibliography

[1] Altman, E., "Financial Ratios, Discriminant Analysis and the Prediction of Corporate Bankruptcy," *Journal of Finance*, Vol. 23, No. 4, (Sep., 1968), pp. 589-609.

[2] Basel Committee on Banking Supervision, "Studies on the Validation of Internal Rating Systems," Bank for International Settlements (BIS), Working Papers No. 14, revised version, May 2005. Available at: http://www.bis.org/publ/bcbs_wp14.htm.

[3] Basel Committee on Banking Supervision, "International Convergence of Capital Measurement and Capital Standards: A Revised Framework," Bank for International Settlements (BIS), comprehensive version, June 2006. Available at: <http://www.bis.org/publ/bcbsca.htm>.

[4] Loeffler, G., and P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.

[5] Merton, R., "On the Pricing of Corporate Debt: The Risk Structure of Interest Rates," *Journal of Finance*, Vol. 29, No. 2, (May, 1974), pp. 449-70.

Estimation of Transition Probabilities

In this section...

“Introduction” on page 8-21

“Estimate Transition Probabilities” on page 8-21

“Estimate Transition Probabilities for Different Rating Scales” on page 8-24

“Estimate Point-in-Time and Through-the-Cycle Probabilities” on page 8-25

“Estimate t -Year Default Probabilities” on page 8-28

“Estimate Bootstrap Confidence Intervals” on page 8-29

“Group Credit Ratings” on page 8-30

“Work with Nonsquare Matrices” on page 8-32

“Remove Outliers” on page 8-34

“Estimate Probabilities for Different Segments” on page 8-35

“Work with Large Datasets” on page 8-36

Introduction

Credit ratings rank borrowers according to their credit worthiness. Though this ranking is, in itself, useful, institutions are also interested in knowing how likely it is that borrowers in a particular rating category will be upgraded or downgraded to a different rating, and especially, how likely it is that they will default.

Transition probabilities offer one way to characterize the past changes in credit quality of obligors (typically firms), and are cardinal inputs to many risk management applications. Financial Toolbox software supports the estimation of transition probabilities using both cohort and duration (also known as hazard rate or intensity) approaches using `transprob` and related functions.

Note: The sample dataset used throughout this section is simulated using a single transition matrix. No attempt is made to match historical trends in transition rates.

Estimate Transition Probabilities

The `Data_TransProb.mat` file contains sample credit ratings data.

```

load Data_TransProb
data(1:10,:)

ans =

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '12-May-1986'    'B'
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'
    '00013326'    '09-Feb-1985'    'A'
    '00013326'    '24-Feb-1994'    'AA'
    '00013326'    '10-Nov-2000'    'BBB'
    '00014413'    '23-Dec-1982'    'B'
    '00014413'    '20-Apr-1988'    'BB'
    '00014413'    '16-Jan-1998'    'B'

```

The sample data is formatted as a cell array with three columns. Each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). The assigned credit rating corresponds to the associated ID on the associated date. All information corresponding to the same ID must be stored in contiguous rows. In this example, IDs, dates, and ratings are stored in string format, but you also can enter them in numeric format.

In this example, the simplest calling syntax for `transprob` passes the `nRecords-by-3` cell array as the only input argument. The default `startDate` and `endDate` are the earliest and latest dates in the data. The default estimation algorithm is the duration method and 1-year transition probabilities are estimated:

```

transMat0 = transprob(data)

transMat0 =

93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001    0.0017
1.6166    93.1518    4.3632    0.6602    0.1626    0.0055    0.0004    0.0396
0.1237    2.9003    92.2197    4.0756    0.5365    0.0661    0.0028    0.0753
0.0236    0.2312    5.0059    90.1846    3.7979    0.4733    0.0642    0.2193
0.0216    0.1134    0.6357    5.7960    88.9866    3.4497    0.2919    0.7050
0.0010    0.0062    0.1081    0.8697    7.3366    86.7215    2.5169    2.4399
0.0002    0.0011    0.0120    0.2582    1.4294    4.2898    81.2927    12.7167
0         0         0         0         0         0         0         100.0000

```

It is recommended to provide explicit start and end dates. Otherwise the estimation window for two different datasets can differ, and the estimates might not be comparable. From this point, assume that the time window of interest is the 5-year period from the end of 1995 to the end of 2000. For comparisons, compute the estimates for this time window. First use the `duration` algorithm (default option), and then the `cohort` algorithm explicitly set.

```

startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
transMat1 = transprob(data,startDate,startDate,endDate,endDate)
transMat2 = transprob(data,startDate,startDate,endDate,endDate,...
'algorithm','cohort')

transMat1 =

90.6236    7.9051    1.0314    0.4123    0.0210    0.0020    0.0003    0.0043
4.4780    89.5558    4.5298    1.1225    0.2284    0.0094    0.0009    0.0754
0.3983    6.1164    87.0641    5.4801    0.7637    0.0892    0.0050    0.0832
0.1029    0.8572    10.7918    83.0204    3.9971    0.7001    0.1313    0.3992
0.1043    0.3745    2.2962    14.0954    78.9840    3.0013    0.0463    1.0980
0.0113    0.0544    0.7055    3.2925    15.4350    75.5988    1.8166    3.0860
0.0044    0.0189    0.1903    1.9743    6.2320    10.2334    75.9983    5.3484
      0          0          0          0          0          0          0    100.0000

transMat2 =

90.1554    8.5492    0.9067    0.3886         0         0         0         0
4.9512    88.5221    5.1763    1.0503    0.2251         0         0    0.0750
0.2770    6.6482    86.2188    6.0942    0.6233    0.0693         0    0.0693
0.0794    0.8737    11.6759    81.6521    4.3685    0.7943    0.1589    0.3971
0.1002    0.4008    1.9038    15.4309    77.8557    3.4068         0    0.9018
      0          0    0.2262    2.4887    17.4208    74.2081    2.2624    3.3937
      0          0    0.7576    1.5152    6.0606    10.6061    75.0000    6.0606
      0          0          0          0          0          0          0    100.0000

```

By default, the `cohort` algorithm internally gets yearly snapshots of the credit ratings, but the number of snapshots per year is definable using the parameter/value pair `snapsPerYear`. To get the estimates using quarterly snapshots:

```

transMat3 = transprob(data,startDate,startDate,endDate,endDate,...
'algorithm','cohort','snapsPerYear',4)

transMat3 =

90.4765    8.0881    1.0072    0.4069    0.0164    0.0015    0.0002    0.0032
4.5949    89.3216    4.6489    1.1239    0.2276    0.0074    0.0007    0.0751
0.3747    6.3158    86.7380    5.6344    0.7675    0.0856    0.0040    0.0800
0.0958    0.7967    11.0441    82.6138    4.1906    0.7230    0.1372    0.3987
0.1028    0.3571    2.3312    14.4954    78.4276    3.1489    0.0383    1.0987
0.0084    0.0399    0.6465    3.0962    16.0789    75.1300    1.9044    3.0956
0.0031    0.0125    0.1445    1.8759    6.2613    10.7022    75.6300    5.3705
      0          0          0          0          0          0          0    100.0000

```

Both `duration` and `cohort` compute 1-year transition probabilities by default, but the time interval for the transitions is definable using the parameter/value pair `transInterval`. For example, to get the 2-year transition probabilities using the `cohort` algorithm with the same snapshot periodicity and estimation window:

```

transMat4 = transprob(data,startDate,startDate,endDate,endDate,...
'algorithm','cohort','snapsPerYear',4,'transInterval',2)

transMat4 =

82.2358    14.6092    2.2062    0.8543    0.0711    0.0074    0.0011    0.0149
8.2803    80.4584    8.3606    2.2462    0.4665    0.0316    0.0030    0.1533
0.9604    11.1975    76.1729    9.7284    1.5322    0.2044    0.0162    0.1879

```

```

0.2483    2.0903    18.8440    69.5145    6.9601    1.2966    0.2329    0.8133
0.2129    0.8713    5.4893    23.5776    62.6438    4.9464    0.1390    2.1198
0.0378    0.1895    1.7679    7.2875    24.9444    57.1783    2.8816    5.7132
0.0154    0.0716    0.6576    4.2157    11.4465    16.3455    57.4078    9.8399
      0          0          0          0          0          0          0    100.0000

```

Estimate Transition Probabilities for Different Rating Scales

The dataset `data` from `Data_TransProb.mat` contains sample credit ratings using the default rating scale `{'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D'}`. It also contains the dataset `dataIGSG` with ratings investment grade ('IG'), speculative grade ('SG'), and default ('D'). To estimate the transition matrix for this dataset, use the `labels` argument.

```

load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
dataIGSG(1:10,:)
transMatIGSG = transprob(dataIGSG,'labels',{'IG','SG','D'},...
    'startDate',startDate,'endDate',endDate)

```

ans =

```

'00011253'    '04-Apr-1983'    'IG'
'00012751'    '17-Feb-1985'    'SG'
'00012751'    '19-May-1986'    'D'
'00014690'    '17-Jan-1983'    'IG'
'00012144'    '21-Nov-1984'    'IG'
'00012144'    '25-Mar-1992'    'SG'
'00012144'    '07-May-1994'    'IG'
'00012144'    '23-Jan-2000'    'SG'
'00012144'    '20-Aug-2001'    'IG'
'00012937'    '07-Feb-1984'    'IG'

```

transMatIGSG =

```

98.1986    1.5179    0.2835
 8.5396    89.4891    1.9713
      0          0    100.0000

```

There is another dataset, `dataIGSGnum`, with the same information as `dataIGSG`, except the ratings are mapped to a numeric scale where 'IG'=1, 'SG'=2, and 'D'=3. To estimate the transition matrix, use the `labels` optional argument specifying the numeric scale as a cell array.


```

dataIGSGnum(1:10,:)
% Note {1,2,3} and num2cell(1:3) are equivalent; num2cell is convenient
% when the number of ratings is larger
transMatIGSGnum = transprob(dataIGSGnum,'labels',{1,2,3},...
'startDate',startDate,'endDate',endDate)

```

```
ans =
```

```

'00011253'    '04-Apr-1983'    [1]
'00012751'    '17-Feb-1985'    [2]
'00012751'    '19-May-1986'    [3]
'00014690'    '17-Jan-1983'    [1]
'00012144'    '21-Nov-1984'    [1]
'00012144'    '25-Mar-1992'    [2]
'00012144'    '07-May-1994'    [1]
'00012144'    '23-Jan-2000'    [2]
'00012144'    '20-Aug-2001'    [1]
'00012937'    '07-Feb-1984'    [1]

```

```
transMatIGSGnum =
```

```

98.1986    1.5179    0.2835
 8.5396    89.4891    1.9713
 0          0    100.0000

```

Any time the input dataset contains ratings not included in the default rating scale {'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D'}, the full rating scale needs to be specified using the `labels` optional argument. For example, if the dataset contains ratings 'AAA', ..., 'CCC', 'D', and 'NR' (not rated), use `labels` with this cell array {'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D', 'NR'}.

Estimate Point-in-Time and Through-the-Cycle Probabilities

Transition probability estimates are sensitive to the length of the estimation window. When the estimation window is small, the estimates only capture recent credit events, and these can change significantly from one year to the next. These are called point-in-time (PIT) estimates. In contrast, a large time window yields fairly stable estimates that average transition rates over a longer period of time. These are called through-the-cycle (TTC) estimates.

The estimation of PIT probabilities requires repeated calls to `transprob` with a rolling estimation window. Use `transprobprep` every time repeated calls to `transprob` are required. `transprobprep` performs a preprocessing step on the raw dataset that is

independent of the estimation window. The benefits of `transprobprep` are greater as the number of repeated calls to `transprob` increases. Also, the performance gains from `transprobprep` are more significant for the `cohort` algorithm.

```
load Data_TransProb
prepData = transprobprep(data);

Years = 1991:2000;
nYears = length(Years);
nRatings = length(prepData.ratingsLabels);
transMatPIT = zeros(nRatings,nRatings,nYears);
algorithm = 'duration';
sampleTotals(nYears,1) = struct('totalsVec',[],'totalsMat',[],...
'algorithm',algorithm);
for t = 1:nYears
    startDate = ['31-Dec-' num2str(Years(t)-1)];
    endDate = ['31-Dec-' num2str(Years(t))];
    [transMatPIT(:, :, t), sampleTotals(t)] = transprob(prepData, ...
        'startDate', startDate, 'endDate', endDate, 'algorithm', algorithm);
end
```

Here is the PIT transition matrix for 1993. Recall that the sample dataset contains simulated credit migrations so the PIT estimates in this example do not match actual historical transition rates.

```
transMatPIT(:, :, Years==1993)

ans =

    95.3193    4.5999    0.0802    0.0004    0.0002    0.0000    0.0000    0.0000
    2.0631    94.5931    3.3057    0.0254    0.0126    0.0002    0.0000    0.0000
    0.0237    2.1748    95.5901    1.4700    0.7284    0.0131    0.0000    0.0000
    0.0003    0.0372    3.2585    95.2914    1.3876    0.0250    0.0001    0.0000
    0.0000    0.0005    0.0657    3.8292    92.7474    3.3459    0.0111    0.0001
    0.0000    0.0001    0.0128    0.7977    8.0926    90.4897    0.5958    0.0113
    0.0000    0.0000    0.0005    0.0459    0.5026    11.1621    84.9315    3.3574
         0         0         0         0         0         0         0    100.0000
```

A structure array stores the `sampleTotals` optional output from `transprob`. The `sampleTotals` structure contains summary information on the total time spent on each rating, and the number of transitions out of each rating, for each year under consideration. For more information on the `sampleTotals` structure, see “Algorithms” on page 18-1416.

As an example, the `sampleTotals` structure for 1993 is used here. The total time spent on each rating is stored in the `totalsVec` field of the structure. The total transitions out of each rating are stored in the `totalsMat` field. A third field, `algorithm`, indicates the algorithm used to generate the structure.

```
sampleTotals(Years==1993).totalsVec
sampleTotals(Years==1993).totalsMat
```

```

sampleTotals(Years==1993).algorithm
ans =
    144.4411  230.0356  262.2438  204.9671  246.1315  147.0767  54.9562  215.1479

ans =
    0  7  0  0  0  0  0  0
    5  0  8  0  0  0  0  0
    0  6  0  4  2  0  0  0
    0  0  7  0  3  0  0  0
    0  0  0  10  0  9  0  0
    0  0  0  1  13  0  1  0
    0  0  0  0  0  7  0  2
    0  0  0  0  0  0  0  0

ans =
duration

```

To get the TTC transition matrix, pass the `sampleTotals` structure array to `transprobybtotals`. Internally, `transprobybtotals` aggregates the information in the `sampleTotals` structures to get the total time spent on each rating over the 10 years considered in this example, and the total number of transitions out of each rating during the same period. `transprobybtotals` uses the aggregated information to get the TTC matrix, or average 1-year transition matrix.

```

transMatTTC = transprobybtotals(sampleTotals)
transMatTTC =
    92.8544  6.1068  0.7463  0.2761  0.0123  0.0009  0.0001  0.0032
    2.9399  92.2329  3.8394  0.7349  0.1676  0.0050  0.0004  0.0799
    0.2410  4.5963  90.3468  3.9572  0.6909  0.0521  0.0025  0.1133
    0.0530  0.4729  7.9221  87.2751  3.5075  0.4650  0.0791  0.2254
    0.0460  0.1636  1.1873  9.3442  85.4305  2.9520  0.1150  0.7615
    0.0031  0.0152  0.2608  1.5563  10.4468  83.8525  1.9771  1.8882
    0.0009  0.0041  0.0542  0.8378  2.9996  7.3614  82.4758  6.2662
    0  0  0  0  0  0  0  100.0000

```

The same TTC matrix could be obtained with a direct call to `transprob`, setting the estimation window to the 10 years under consideration. But it is much more efficient to use the `sampleTotals` structures, whenever they are available. (Note, for the `duration` algorithm, these alternative workflows can result in small numerical differences in the estimates whenever leap years are part of the sample.)

In “Estimate Transition Probabilities” on page 8-21, a 1-year transition matrix is estimated using the 5-year time window from 1996 through 2000. This is another example of a TTC matrix and this can also be computed using the `sampleTotals` structure array.

```

transprobytotals(sampleTotals(Years>=1996&Years<=2000))
ans =
    90.6239    7.9048    1.0313    0.4123    0.0210    0.0020    0.0003    0.0043
    4.4776    89.5565    4.5294    1.1224    0.2283    0.0094    0.0009    0.0754
    0.3982    6.1159    87.0651    5.4797    0.7636    0.0892    0.0050    0.0832
    0.1029    0.8571    10.7909    83.0218    3.9968    0.7001    0.1313    0.3991
    0.1043    0.3744    2.2960    14.0947    78.9851    3.0012    0.0463    1.0980
    0.0113    0.0544    0.7054    3.2922    15.4341    75.6004    1.8165    3.0858
    0.0044    0.0189    0.1903    1.9742    6.2318    10.2332    75.9990    5.3482
         0         0         0         0         0         0         0    100.0000

```

Estimate t -Year Default Probabilities

By varying the start and end dates, the amount of data considered for the estimation is changed, but the output still contains, by default, 1-year transition probabilities. You can change the default behavior by specifying the `transInterval` argument, as illustrated in “Estimate Transition Probabilities” on page 8-21.

However, when t -year transition probabilities are required for a whole range of values of t , for example, 1-year, 2-year, 3-year, 4-year, and 5-year transition probabilities, it is more efficient to call `transprob` once to get the optional output `sampleTotals`. You can use the same `sampleTotals` structure can be used to get the t -year transition matrix for any transition interval t . Given a `sampleTotals` structure and a transition interval, you can get the corresponding transition matrix by using `transprobytotals`.

```

load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';

[~,sampleTotals] = transprob(data,'startDate', ...
    startDate, 'endDate',endDate);

DefProb = zeros(7,5);
for t = 1:5
    transMatTemp = transprobytotals(sampleTotals,'transInterval',t);
    DefProb(:,t) = transMatTemp(1:7,8);
end
DefProb

DefProb =

```

```

    0.0043    0.0169    0.0377    0.0666    0.1033
    0.0754    0.1542    0.2377    0.3265    0.4213
    0.0832    0.1936    0.3276    0.4819    0.6536
    0.3992    0.8127    1.2336    1.6566    2.0779
    1.0980    2.1189    3.0668    3.9468    4.7644
    3.0860    5.6994    7.9281    9.8418    11.4963

```

5.3484 9.8053 13.5320 16.6599 19.2964

Estimate Bootstrap Confidence Intervals

`transprob` also returns the `idTotals` structure array which contains, for each ID, or company, the total time spent on each rating, and the total transitions out of each rating. For more information on the `idTotals` structure, see “Algorithms” on page 18-1416. The `idTotals` structure is similar to the `sampleTotals` structures (see “Estimate Point-in-Time and Through-the-Cycle Probabilities” on page 8-25), but `idTotals` has the information at an ID level. Because most companies only migrate between very few ratings, the numeric arrays in `idTotals` are stored as sparse arrays to reduce memory requirements.

You can use the `idTotals` structure array to estimate confidence intervals for the transition probabilities using a bootstrapping procedure, as the following example demonstrates. To do this, call `transprob` and keep the third output argument, `idTotals`. The `idTotals` fields are displayed for the last company in the sample. Within the estimation window, this company spends almost a year as 'AA' and it is then upgraded to 'AAA'.

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';

[transMat,-,idTotals] = transprob(data,...
    'startDate',startDate,'endDate',endDate);

% Total time spent on each rating
full(idTotals(end).totalsVec)
% Total transitions out of each rating
full(idTotals(end).totalsMat)
% Algorithm
idTotals(end).algorithm

ans =

    4.0820    0.9180         0         0         0         0         0         0

ans =

    0     0     0     0     0     0     0     0
    1     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0

ans =
```

```
duration
```

Next, use `bootstrp` from Statistics Toolbox with `transprobbytotals` as the bootstrap function and `idTotals` as the data to sample from. Each bootstrap sample corresponds to a dataset made of companies sampled with replacement from the original data. However, you do not have to draw companies from the original data, because a bootstrap `idTotals` sample contains all the information required to compute the transition probabilities. `transprobbytotals` aggregates all structures in each bootstrap `idTotals` sample and finds the corresponding transition matrix.

To estimate 95% confidence intervals for the transition matrix and display the probabilities of default together with its upper and lower confidence bounds:

```
PD = transMat(1:7,8);

bootstat = bootstrp(100,@(totals)transprobbytotals(totals),idTotals);
ci = prctile(bootstat,[2.5 97.5]); % 95% confidence
CIlower = reshape(ci(1,:),8,8);
CIupper = reshape(ci(2,:),8,8);
PD_LB = CIlower(1:7,8);
PD_UB = CIupper(1:7,8);

[PD_LB PD PD_UB]

ans =

    0.0004    0.0043    0.0106
    0.0028    0.0754    0.2192
    0.0126    0.0832    0.2180
    0.1659    0.3992    0.6617
    0.5703    1.0980    1.7260
    1.7264    3.0860    4.7602
    1.7678    5.3484    9.5055
```

Group Credit Ratings

Credit rating scales can be more or less granular. For example, there are ratings with qualifiers (such as 'AA+', 'BB-', etc.), whole ratings ('AA', 'BB', etc.), and investment or speculative grade ('IG', 'SG') categories. Given a dataset with credit ratings at a more granular level, transition probabilities for less granular categories can be of interest. For example, you might be interested in a transition matrix for investment and speculative grades given a dataset with whole ratings. Use `transprobgroupptotals` for this evaluation, as illustrated in the following examples. The sample dataset data has whole credit ratings:

```

load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
data(1:5,:)

ans =

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '12-May-1986'    'B'
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'
    '00013326'    '09-Feb-1985'    'A'

```

A call to `transprob` returns the transition matrix and totals structures for the eight ('AAA' to 'D') whole credit ratings. The array with number of transitions out of each credit rating is displayed after the call to `transprob`:

```

[transMat,sampleTotals,idTotals] = transprob(data,'startDate',startDate,...
'endDate',endDate);
sampleTotals.totalsMat

ans =

     0     67     7     3     0     0     0     0
    67     0    68    15     3     0     0     1
     4    101     0    93    11     1     0     1
     1     7   163     0    62    10     2     5
     1     3    16   168     0    37     0    11
     0     0     2    10    83     0    10    14
     0     0     0     2     8    16     0     7
     0     0     0     0     0     0     0     0

```

Next, use `transprobgroupptotals` to group whole ratings into investment and speculative grades. This function takes a totals structure as the first argument. The second argument indicates the edges between rating categories. In this case, ratings 1 through 4 ('AAA' through 'BBB') correspond to the first category ('IG'), ratings 5 through 7 ('BB' through 'CCC') to the second category ('SG'), and rating 8 ('D') is a category of its own. `transprobgroupptotals` adds up the total time spent on ratings that belong to the same category. For example, total times spent on 'AAA' through 'BBB' are added up as the total time spent on 'IG'. `transprobgroupptotals` also adds up the total number of transitions between any 'IG' rating and any 'SG' rating, for example, a credit migration from 'BBB' to 'BB'.

The grouped totals can then be passed to `transprobbytotals` to obtain the transition matrix for investment and speculative grades. Note that both `totalsMat` and the new

transition matrix are both 3-by-3, corresponding to the grouped categories 'IG', 'SG', and 'D'.

```
sampleTotalsIGSG = transprobgrouptotals(sampleTotals,[4 7 8])
transMatIGSG = transprobybtotals(sampleTotalsIGSG)
```

```
sampleTotalsIGSG =
```

```
totalsVec: [4.8591e+003 1.5034e+003 1.1621e+003]
totalsMat: [3x3 double]
algorithm: 'duration'
```

```
transMatIGSG =
```

```
98.1591    1.6798    0.1611
12.3228    85.6961    1.9811
         0         0 100.0000
```

When a totals structure array is passed to `transprobgrouptotals`, this function groups each structure in the array individually and preserves sparsity, if the fields in the input structures are sparse. One way to exploit this feature is to compute confidence intervals for the investment grade default rate and the speculative grade default rate (see also “Estimate Bootstrap Confidence Intervals” on page 8-29).

```
PDIGSG = transMatIGSG(1:2,3);
```

```
idTotalsIGSG = transprobgrouptotals(idTotals,[4 7 8]);
bootstat = bootstrp(100,@(totals)transprobybtotals(totals),idTotalsIGSG);
ci = prctile(bootstat,[2.5 97.5]); % 95% confidence
CIlower = reshape(ci(1,:),3,3);
CIupper = reshape(ci(2,:),3,3);
PDIGSG_LB = CIlower(1:2,3);
PDIGSG_UB = CIupper(1:2,3);
```

```
[PDIGSG_LB PDIGSG PDIGSG_UB]
```

```
ans =
```

```
0.0603    0.1611    0.2538
1.3470    1.9811    2.6195
```

Work with Nonsquare Matrices

Transition probabilities and the number of transitions between ratings are usually reported without the 'D' ('Default') row. For example, a credit report can contain the

following table, indicating the number of issuers starting in each rating (first column), and the number of transitions between ratings (remaining columns):

Initial	AAA	AA	A	BBB	BB	B	CCC	D
AAA	98	88	9	1	0	0	0	0
AA	389	0	368	19	2	0	0	0
A	1165	1	21	1087	56	0	0	0
BBB	1435	0	2	89	1289	45	8	0
BB	915	0	0	1	60	776	73	2
B	867	0	0	1	7	88	715	39
CCC	112	0	0	0	1	3	34	61

You can store the information in this table in a totals structure compatible with the cohort algorithm. For more information on the cohort algorithm and the totals structure, see “Algorithms” on page 18-1416. Note that the `totalsMat` field is a nonsquare array in this case.

```
% Define totals structure
totals.totalVec = [98 389 1165 1435 915 867 112];
totals.totalMat = [
    88    9    1    0    0    0    0    0;
    0   368   19    2    0    0    0    0;
    1    21 1087   56    0    0    0    0;
    0    2    89 1289   45    8    0    2;
    0    0    1    60  776   73    2    3;
    0    0    1    7   88  715   39   17;
    0    0    0    1    3   34   61   13];
totals.algorithm = 'cohort';
```

`transprobytotals` and `transprobrouptotals` accept totals inputs with nonsquare `totalsMat` fields. To get the transition matrix corresponding to the previous table, and to group ratings into investment and speculative grade with the corresponding matrix:

```
transMat = transprobytotals(totals)

% Group into IG/SG and get IG/SG transition matrix
totalsIGSG = transprobrouptotals(totals,[4 7]);
transMatIGSG = transprobytotals(totalsIGSG)

transMat =

    89.7959    9.1837    1.0204         0         0         0         0
         0    94.6015    4.8843    0.5141         0         0         0
    0.0858    1.8026   93.3047    4.8069         0         0         0
         0    0.1394    6.2021   89.8258    3.1359    0.5575         0    0.1394
         0         0    0.1093    6.5574   84.8087    7.9781    0.2186    0.3279
         0         0    0.1153    0.8074   10.1499   82.4683    4.4983    1.9608
         0         0         0    0.8929    2.6786   30.3571   54.4643   11.6071
```

```
transMatIGSG =
    98.2183    1.7169    0.0648
    3.6959    94.5618    1.7423
```

Remove Outliers

The `idTotals` output from `transprob` can also be exploited to update the transition probability estimates after removing some outlier information. For more information on `idTotals`, see “Algorithms” on page 18-1416. For example, if you know that the credit rating migration information for the 4th and 27th companies in the data have problems, you can remove those companies and efficiently update the transition probabilities as follows:

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
[transMat,~,idTotals] = transprob(data,'startDate', ...
startDate, 'endDate',endDate);
transMat

transMat =
    90.6236    7.9051    1.0314    0.4123    0.0210    0.0020    0.0003    0.0043
    4.4780   89.5558    4.5298    1.1225    0.2284    0.0094    0.0009    0.0754
    0.3983    6.1164   87.0641    5.4801    0.7637    0.0892    0.0050    0.0832
    0.1029    0.8572   10.7918   83.0204    3.9971    0.7001    0.1313    0.3992
    0.1043    0.3745    2.2962   14.0954   78.9840    3.0013    0.0463    1.0980
    0.0113    0.0544    0.7055    3.2925   15.4350   75.5988    1.8166    3.0860
    0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
         0         0         0         0         0         0         0   100.0000

nIDs = length(idTotals);
keepInd = setdiff(1:nIDs,[4 27]);
transMatNoOutlier = transprobbytotals(idTotals(keepInd))

transMatNoOutlier =
    90.6241    7.9067    1.0290    0.4124    0.0211    0.0020    0.0003    0.0043
    4.4917   89.5918    4.4779    1.1240    0.2288    0.0094    0.0009    0.0756
    0.3990    6.1220   87.0530    5.4841    0.7643    0.0893    0.0050    0.0833
    0.1030    0.8576   10.7909   83.0207    3.9971    0.7001    0.1313    0.3992
    0.1043    0.3746    2.2960   14.0955   78.9840    3.0013    0.0463    1.0980
    0.0113    0.0544    0.7054    3.2925   15.4350   75.5988    1.8166    3.0860
    0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
         0         0         0         0         0         0         0   100.0000
```

Deciding which companies to remove is a case-by-case situation. Reasons to remove a company can include a typo in one of the ratings history, or an unusual migration between ratings whose impact on the transition probability estimates must be measured. Note that `transprob` does not reorder the companies in any way. The ordering of companies in the input data is the same as the ordering in the `idTotals` array.

Estimate Probabilities for Different Segments

You can use `idTotals` efficiently to get estimates over different segments of the sample. For more information on `idTotals`, see “Algorithms” on page 18-1416. For example, assume that the companies in the example are grouped into three geographic regions and that the companies were grouped by geographic regions previously, so that the first 340 companies correspond to the first region, the next 572 companies to the second region, and the rest to the third region. You can efficiently get transition probabilities for each region as follows:

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
[~,~,idTotals] = transprob(data,'startDate', ...
    startDate, 'endDate',endDate);

n1 = 340;
n2 = 572;
transMatG1 = transprobytotals(idTotals(1:n1))
transMatG2 = transprobytotals(idTotals(n1+1:n1+n2))
transMatG3 = transprobytotals(idTotals(n1+n2+1:end))

transMatG1 =

90.8299    7.6501    0.3178    1.1700    0.0255    0.0044    0.0021    0.0002
4.3572    89.0262    5.7838    0.8039    0.0245    0.0029    0.0013    0.0001
0.7066    6.7567    86.6320    5.4950    0.3721    0.0252    0.0101    0.0023
0.0626    1.3688    10.3895    83.5022    3.6823    0.6466    0.3084    0.0396
0.0256    0.7884    2.6970    13.7857    78.8321    2.8310    0.0561    0.9842
0.0026    0.1095    0.4280    3.5204    21.1437    72.9230    1.6456    0.2273
0.0005    0.0216    0.0730    0.4574    4.9586    4.2821    80.3062    9.9006
0          0          0          0          0          0          0        100.0000

transMatG2 =

90.5798    8.4877    0.8202    0.0884    0.0132    0.0011    0.0000    0.0096
4.1999    90.0371    3.8657    1.4744    0.2144    0.0128    0.0001    0.1956
0.3022    5.9869    86.7128    5.5526    1.0411    0.1902    0.0015    0.2127
0.0204    0.5606    10.9342    82.9195    4.0123    0.7398    0.0059    0.8073
0.0089    0.3338    2.1185    16.6496    76.2395    3.1241    0.0261    1.4995
0.0013    0.0465    0.6710    2.4731    14.7281    76.7378    1.2993    4.0428
0.0002    0.0080    0.0681    0.4598    4.1324    8.4380    80.9092    5.9843
0          0          0          0          0          0          0        100.0000

transMatG3 =

90.5655    7.5408    1.5288    0.3369    0.0258    0.0015    0.0003    0.0004
4.8073    89.3842    4.4865    0.9582    0.3509    0.0095    0.0009    0.0025
0.3153    5.8771    87.6353    5.4101    0.7160    0.0322    0.0052    0.0088
0.1995    0.8625    10.8682    82.8717    4.1423    0.6903    0.1565    0.2090
0.2465    0.1091    2.1558    12.0289    81.5803    3.0057    0.0616    0.8122
0.0227    0.0400    0.9380    4.3175    12.3632    75.9429    2.5766    3.7991
0.0149    0.0180    0.3414    3.6918    8.1414    13.6010    70.7254    3.4661
0          0          0          0          0          0          0        100.0000
```

Work with Large Datasets

This example shows how to aggregate estimates from two (or more) datasets. It is possible that two datasets, coming from two different databases, must be considered for the estimation of the transition probabilities. Also, if a dataset is too large and cannot be loaded into memory, the dataset can be split into two (or more) datasets. In these cases, it is simple to apply `transprob` to each individual dataset, and then get the final estimates corresponding to the aggregated data with a call to `transprobbytotals` at the end.

For example, the dataset `data` is artificially split into two sections in this example. In practice the two datasets would come from different files or databases. When aggregating multiple datasets, the history of a company cannot be split across datasets. You can verify that this condition is satisfied for the arbitrarily chosen cut-off point.

```
load Data_TransProb
```

```
cutoff = 2099;  
data(cutoff-5:cutoff, :)  
data(cutoff+1:cutoff+6, :)
```

```
ans =
```

```
'00011166'    '24-Aug-1995'    'BBB'  
'00011166'    '25-Jan-1997'    'A'  
'00011166'    '01-Feb-1998'    'AA'  
'00014878'    '15-Mar-1983'    'B'  
'00014878'    '21-Sep-1986'    'BB'  
'00014878'    '17-Jan-1998'    'BBB'
```

```
ans =
```

```
'00012043'    '09-Feb-1985'    'BBB'  
'00012043'    '03-Jan-1988'    'A'  
'00012043'    '15-Jan-1994'    'AAA'  
'00011157'    '24-Jun-1984'    'A'  
'00011157'    '09-Dec-1999'    'BBB'  
'00011157'    '28-Mar-2001'    'A'
```

When working with multiple datasets, it is important to set the start and end dates explicitly. Otherwise, the estimation window will differ for each dataset because the

default start and end dates used by `transprob` are the earliest and latest dates found in the input data.

```
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
```

In practice, this is the point where you can read in the first dataset. Now, the dataset is already obtained. Call `transprob` with the first dataset and the explicit start and end dates. Keep only the `sampleTotals` output. For details on `sampleTotals`, see “Algorithms” on page 18-1416.

```
[~,sampleTotals(1)] = transprob(data(1:cutoff,:),...
    'startDate',startDate,'endDate',endDate);
```

Repeat for the remaining datasets. Note the different `sampleTotals` structures are stored in a structured array.

```
[~,sampleTotals(2)] = transprob(data(cutoff+1:end,:),...
    'startDate',startDate,'endDate',endDate);
```

To get the transition matrix corresponding to the aggregated dataset, use `transprobybtotals`. When the totals input is a structure array, `transprobybtotals` aggregates the information over all structures, and returns a single transition matrix.

```
transMatAggr = transprobybtotals(sampleTotals)
```

```
transMatAggr =
```

90.6236	7.9051	1.0314	0.4123	0.0210	0.0020	0.0003	0.0043
4.4780	89.5558	4.5298	1.1225	0.2284	0.0094	0.0009	0.0754
0.3983	6.1164	87.0641	5.4801	0.7637	0.0892	0.0050	0.0832
0.1029	0.8572	10.7918	83.0204	3.9971	0.7001	0.1313	0.3992
0.1043	0.3745	2.2962	14.0954	78.9840	3.0013	0.0463	1.0980
0.0113	0.0544	0.7055	3.2925	15.4350	75.5988	1.8166	3.0860
0.0044	0.0189	0.1903	1.9743	6.2320	10.2334	75.9983	5.3484
0	0	0	0	0	0	0	100.0000

As a sanity check, for this example you can verify that the aggregation procedure yields the same estimates (up to numerical differences) as estimating the probabilities directly over the entire sample:

```
transMatWhole = transprob(data,'startDate',startDate,'endDate',endDate)
aggError = max(max(abs(transMatAggr - transMatWhole)))
```

```
transMatWhole =
```

90.6236	7.9051	1.0314	0.4123	0.0210	0.0020	0.0003	0.0043
4.4780	89.5558	4.5298	1.1225	0.2284	0.0094	0.0009	0.0754
0.3983	6.1164	87.0641	5.4801	0.7637	0.0892	0.0050	0.0832
0.1029	0.8572	10.7918	83.0204	3.9971	0.7001	0.1313	0.3992
0.1043	0.3745	2.2962	14.0954	78.9840	3.0013	0.0463	1.0980

0.0113	0.0544	0.7055	3.2925	15.4350	75.5988	1.8166	3.0860
0.0044	0.0189	0.1903	1.9743	6.2320	10.2334	75.9983	5.3484
0	0	0	0	0	0	0	100.0000

aggError =

2.8422e-014

Forecasting Corporate Default Rates

This example shows how to build a forecasting model for corporate default rates.

Risk parameters are dynamic in nature, and understanding how these parameters change in time is a fundamental task for risk management.

In the first part, we work with historical credit migrations data to construct some time series of interest, and to visualize default rates dynamics. In the second part, we use some of the series constructed in the first part, and some additional data, to fit a forecasting model for corporate default rates, and to show some backtesting and stress testing concepts. A linear regression model for corporate default rates is studied there, but the tools and concepts described can be used in combination with other forecasting methodologies. The appendix at the end touches on the handling of models for full transition matrices.

People interested in forecasting, backtesting and stress testing may go directly to the second part. The first part is more relevant for people who work with credit migration data.

Part I: Working with Credit Migrations Data

We work with historical transition probabilities for corporate issuers (variable `TransMat`). This is yearly data for the period 1981-2005, from [10]. The data includes, for each year, the number of issuers per rating at the beginning of the year (variable `nIssuers`), and the number of new issuers per rating per year (variable `nNewIssuers`). There is also a corporate profits forecast, from [9], and a corporate spread, from [4] (variables `CPF` and `SPR`). A variable indicating recession years (`Recession`), consistent with recession dates from [7], is used mainly for visualizations.

`Example_LoadData`

Getting Default Rates for Different Ratings Categories

We start by performing some aggregations to get corporate default rates for Investment Grade (IG) and Speculative Grade (SG) issuers, and the overall corporate default rate.

Aggregation and segmentation are relative terms. IG is an aggregate with respect to credit ratings, but a segment from the perspective of the overall corporate portfolio. Other segments are of interest in practice, for example, economic sectors, industries, or geographic regions. The data we use, however, is aggregated by credit ratings, so further segmentation is not possible. Nonetheless, the tools and workflow discussed here may be useful to work with other segment-specific models.

We use existing functionality in Financial Toolbox™, specifically, functions `transprobgrouptotals` and `transprobbytotals`, to perform the aggregation. These functions take as inputs structures with credit migration information in a particular format. We set up the inputs here, and visualize them below to understand their information and format.

```
% Pre-allocate the struct array
totalsByRtg(nYears,1) = struct('totalsVec',[],'totalsMat',[],...
    'algorithm','cohort');
for t = 1:nYears
    % Number of issuers per rating at the beginning of the year
    totalsByRtg(t).totalsVec = nIssuers(t,:);
    % Number of transitions between ratings during the year
    totalsByRtg(t).totalsMat = round(diag(nIssuers(t,:))*...
        (0.01*TransMat(:,:,t)));
    % Algorithm
    totalsByRtg(t).algorithm = 'cohort';
end
```

It is useful to see both the original data and the data stored in these totals structures side to side. The original data contains number of issuers and transition probabilities for each year. For example, for 2005

```
fprintf('\nTransition matrix for 2005:\n\n')
Example_DisplayTransitions(squeeze(TransMat(:,:,end)),nIssuers(end,:),...
    {'AAA','AA','A','BBB','BB','B','CCC'},...
    {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

Transition matrix for 2005:

	Init	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	98	88.78	9.18	1.02	0	0	0	0	0	1.02
AA	407	0	90.66	4.91	0.49	0	0	0	0	3.93
A	1224	0.08	1.63	88.89	4.41	0	0	0	0	4.98
BBB	1535	0	0.2	5.93	84.04	3.06	0.46	0	0.07	6.25
BB	1015	0	0	0	5.71	76.75	6.9	0.2	0.2	10.25
B	1010	0	0	0.1	0.59	8.51	70.59	3.76	1.58	14.85
CCC	126	0	0	0	0.79	0.79	25.4	46.83	8.73	17.46

The totals structure stores the total number of issuers per rating at the beginning of the year in the `totalsVec` field, and the total *number of migrations* between ratings (instead of transition probabilities) in the `totalsMat` field. Here is the information for 2005


```
fprintf('\nTransition counts (totals struct) for 2005:\n\n')
Example_DisplayTransitions(totalsByRtg(end).totalsMat,...
    totalsByRtg(end).totalsVec,...
    {'AAA','AA','A','BBB','BB','B','CCC'},...
    {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

Transition counts (totals struct) for 2005:

	Init	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	98	87	9	1	0	0	0	0	0	1
AA	407	0	369	20	2	0	0	0	0	16
A	1224	1	20	1088	54	0	0	0	0	61
BBB	1535	0	3	91	1290	47	7	0	1	96
BB	1015	0	0	0	58	779	70	2	2	104
B	1010	0	0	1	6	86	713	38	16	150
CCC	126	0	0	0	1	1	32	59	11	22

The third field in the totals structure, `algorithm`, indicates that we are working with the 'cohort' method ('duration' is also supported, although the information in `totalsVec` and `totalsMat` would be different). These structures are usually obtained as optional outputs from `transprob`, but this example shows how a user can define these structures directly.

We now group ratings 'AAA' to 'BBB' (ratings 1 to 4) into the IG category and ratings 'BB' to 'CCC' (ratings 5 to 7) into the SG category. We use `transprobgroupptotals` for this. The 'edges' argument tells the function which ratings are to be grouped together (1 to 4, and 5 to 7). We also group all non-default ratings into one category. These are preliminary steps to get the IG, SG and overall default rates for each year.

```
edgesIGSG = [4 7];
totalsIGSG = transprobgroupptotals(totalsByRtg,edgesIGSG);
edgesAll = 7; % could also use edgesAll = 2 with totalsIGSG
totalsAll = transprobgroupptotals(totalsByRtg,edgesAll);
```

Here are the 2005 totals grouped at IG/SG level, and the corresponding transition matrix, recovered using `transprobbytotals`.

```
fprintf('\nTransition counts for 2005 at IG/SG level:\n\n')
Example_DisplayTransitions(totalsIGSG(end).totalsMat,...
    totalsIGSG(end).totalsVec,...
    {'IG','SG'},...
    {'IG','SG','D','NR'})
```

```
fprintf('\nTransition matrix for 2005 at IG/SG level:\n\n')
```

```
Example_DisplayTransitions(transprobytotals(totalsIGSG(end)),[],...
    {'IG','SG'},...
    {'IG','SG','D','NR'})
```

Transition counts for 2005 at IG/SG level:

	Init	IG	SG	D	NR
IG	3264	3035	54	1	174
SG	2151	66	1780	29	276

Transition matrix for 2005 at IG/SG level:

	IG	SG	D	NR
IG	92.98	1.65	0.03	5.33
SG	3.07	82.75	1.35	12.83

We now get transition matrices for every year both at IG/SG and non-default/default levels and store the default rates only (we do not use the rest of the transition probabilities).

```
DefRateIG = zeros(nYears,1);
DefRateSG = zeros(nYears,1);
DefRate = zeros(nYears,1);
for t=1:nYears
    % Get transition matrix at IG/SG level and extract IG default rate and
    % SG default rate for year t
    tmIGSG = transprobytotals(totalsIGSG(t));
    DefRateIG(t) = tmIGSG(1,3);
    DefRateSG(t) = tmIGSG(2,3);
    % Get transition matrix at most aggregate level and extract overall
    % corporate default rate for year t
    tmAll = transprobytotals(totalsAll(t));
    DefRate(t) = tmAll(1,2);
end
```

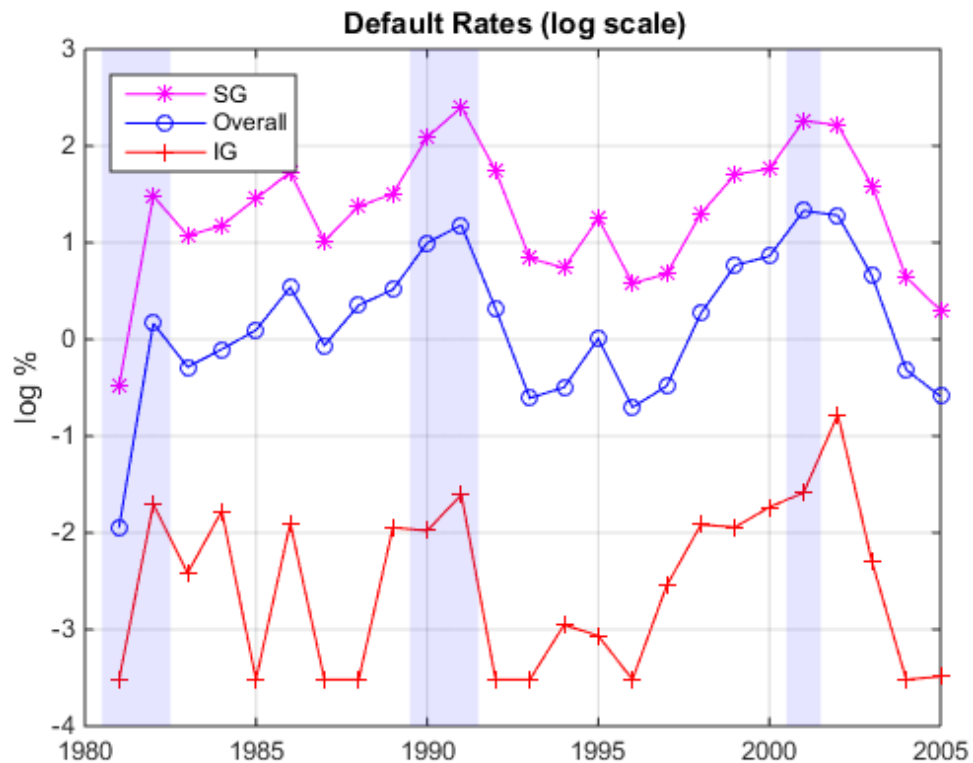
Here is a visualization of the dynamics of IG, SG and overall corporate default rates together. To emphasize their patterns, rather than their magnitudes, a log scale is used. The shaded bands indicate recession years. The patterns of SG and IG are slightly different. For example, the IG rate is higher in 1994 than in 1995, but the opposite is true for SG. More noticeably, the IG default rate peaked after the 2001 recession, in 2002, whereas the peak for SG is in 2001. This suggests that models for the dynamics of the IG and SG default rates could have important differences, a common situation when working with different segments. The overall corporate default rate is by construction

a combination of the other two, and its pattern is closer to SG, most likely due to the relative magnitude of SG versus IG.

```

minIG = min(DefRateIG(DefRateIG==0));
figure
plot(Years,log(DefRateSG),'m-*')
hold on
plot(Years,log(DefRate),'b-o')
plot(Years,log(max(DefRateIG,minIG-0.001)),'r-+')
Example_RecessionBands
hold off
grid on
title('\bf Default Rates (log scale)')
ylabel('log %')
legend({'SG','Overall','IG'},'location','NW')

```



Getting Default Rates for Different Time Periods

The default rates just obtained are examples of point-in-time (PIT) rates, only the most recent information is used to estimate them. On the other extreme, we can use all the migrations observed in the 25 years spanned by the dataset to estimate long-term, or through-the-cycle (TTC) default rates. Other rates of interest are the average default rates over recession or expansion years.

All of these are easy to estimate with the data we have and the same tools. For example, to estimate the average transition probabilities over recession years, pass to `transprobbytotals` the totals structures corresponding to the recession years only. We use logical indexing below, taking advantage of the `Recession` variable. `transprobbytotals` aggregates the information over time and returns the corresponding transition matrix.

```
tmAllRec = transprobbytotals(totalsAll(Recession));
DefRateRec = tmAllRec(1,2);

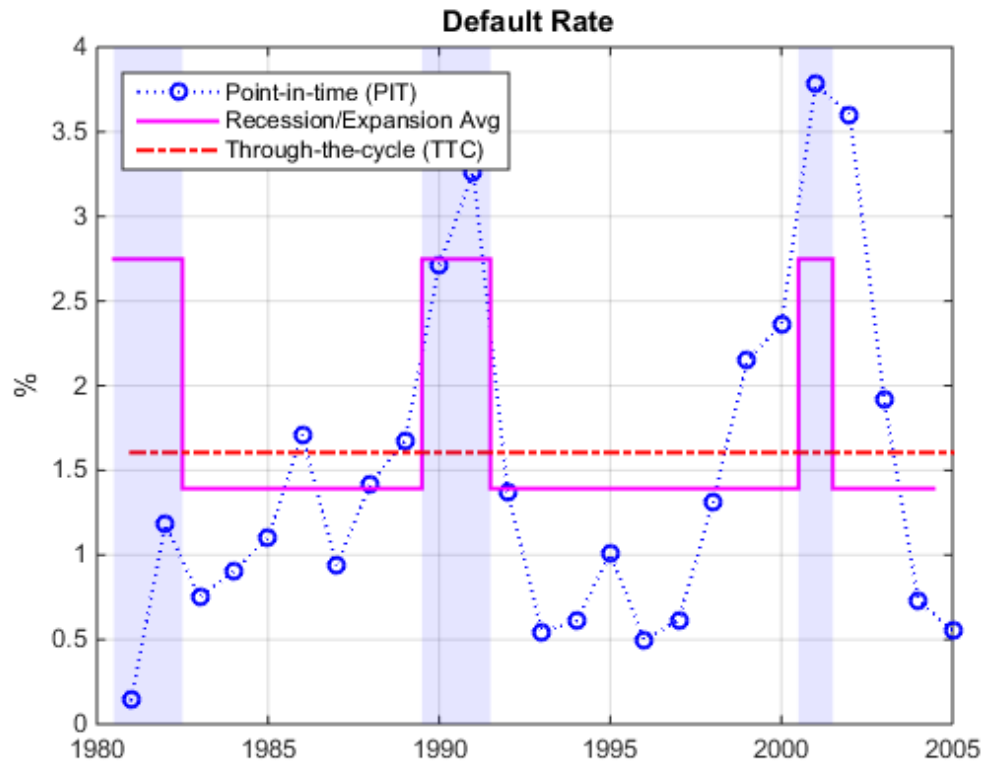
tmAllExp = transprobbytotals(totalsAll(~Recession));
DefRateExp = tmAllExp(1,2);

tmAllTTC = transprobbytotals(totalsAll);
DefRateTTC = tmAllTTC(1,2);
```

The following figure shows the estimated PIT rates, TTC rates and recession and expansion rates.

```
DefRateTwoValues = DefRateExp*ones(nYears,1);
DefRateTwoValues(Recession) = DefRateRec;

figure
plot(Years,DefRate,'bo:','LineWidth',1.2)
hold on
stairs(Years-0.5,DefRateTwoValues,'m-','LineWidth',1.5)
plot(Years,DefRateTTC*ones(nYears,1),'r-.','LineWidth',1.5)
Example_RecessionBands
hold off
grid on
title({'\bf Default Rate'})
ylabel('%')
legend({'Point-in-time (PIT)','Recession/Expansion Avg',...
       'Through-the-cycle (TTC)'},'location','NW')
```



Some analyses (see, for example, [11]) use simulations where the default rate is conditional on the general state of the economy, e.g., recession v. expansion. The recession and expansion estimates just obtained can be useful in such a framework. These are all historical averages, however, and may not work well if used as predictions for the actual default rates expected on any particular year. In the second part, we revisit the use these types of historical averages as forecasting tools in a backtesting exercise.

Building Predictors Using Credit Ratings Data

Using the credit data one can build new time series of interest. We start with an age proxy that is used as predictor in the forecasting model in the second part.

Age is known to be an important factor in predicting default rates; see, e.g., [1] and [5]. Age here means the number of years since a bond was issued. By extension, the age of a

portfolio is the average age of its bonds. Certain patterns have been observed historically. Many low quality borrowers default just a few years after issuing a bond. When troubled companies issue bonds, the amount borrowed helps them make payments for a year or two. Beyond that point, their only source of money is their cash flows, and if they are insufficient, default occurs.

We cannot calculate the exact age of the portfolio, because there is no information at issuer level in the dataset. We follow [6], however, and use the number of new issuers in year $t-3$ divided by the total number of issuers at the end of year t as an age proxy. Because of the lag, the age proxy starts in 1984. For the numerator, we have explicit information on the number of new issuers. For the denominator, the number of issuers at the end of a year equals the number of issuers at the beginning of next year. This is known for all years but the last one, which is set to the total transitions into a non-default rating plus the number of new issuers on that year.

```
% Total number of issuers at the end of the year
nEOY = zeros(nYears,1);
% nIssuers is number of issuers per ratings at the beginning of the year
% nEOY ( 1981 ) = sum nIssuers ( 1982 ), etc until 2004
nEOY(1:end-1) = sum(nIssuers(2:end,:),2);
% nEOY ( 2005 ) = issuers in non-default state at end of 2005 plus
% new issuers in 2005
nEOY(end) = totalsAll(end).totalsMat(1,1) + sum(nNewIssuers(end,:));
% Age proxy
AGE = 100*[nan(3,1); sum(nNewIssuers(1:end-3,:),2)./nEOY(4:end)];
```

Examples of other time series of interest are the proportion of SG issuers at the end of each year, or an age proxy for SG.

```
% nSGEOY: Number of SG issuers at the end of the year
% nSGEOY is similar to nEOY, but for SG only, from 5 ('BB') to 7 ('CCC')
indSG = 5:7;
nSGEOY = zeros(nYears,1);
nSGEOY(1:end-1) = sum(nIssuers(2:end,indSG),2);
nSGEOY(end) = sum(totalsIGSG(end).totalsMat(:,2)) + ...
    sum(nNewIssuers(end,indSG));
% Proportion of SG issuers
SG = 100*nSGEOY./nEOY;
% SG age proxy: new SG issuers in t-3 / total issuers at the end of year t
AGESG = 100*[nan(3,1); sum(nNewIssuers(1:end-3,indSG),2)./nEOY(4:end)];
```

Part II: A Forecasting Model for Default Rates

We work with the following linear regression model for corporate default rates

$$DefRate = \beta_0 + \beta_{age}AGE + \beta_{cpf}CPF + \beta_{spr}SPR$$

where

- AGE: Age proxy defined above
- CPF: Corporate profits forecast
- SPR: Corporate spread over treasuries

This is the same model as in [6], except the model in [6] is for IG only.

As discussed above, age is known to be an important factor regarding default rates. The corporate profits provide information on the economic environment. The corporate spread is a proxy for credit quality. Age, environment and quality are three dimensions frequently found in credit analysis models.

```
inSample = 4:nYears-1;
T = length(inSample);
varNames = {'AGE', 'CPF', 'SPR'};
X = [AGE CPF SPR];
X = X(inSample,:);
y = DefRate(inSample+1); % DefaultRate, year t+1
stats = regstats(y,X);

fprintf('\nConst  AGE  CPF  SPR  adjR^2\n')
fprintf('%1.2f  %1.2f  %1.2f  %1.2f  %1.4f\n',...
        [stats.beta;stats.adjrsquare])
```

```
Const  AGE  CPF  SPR  adjR^2
-1.19  0.15 -0.10  0.71  0.7424
```

The coefficients have the expected sign: default rates tend to increase with a higher proportion of 3-year issuers, decrease with good corporate profits, and increase when the corporate yields are higher. The adjusted R square shows a good fit.

The in-sample fit, or how close the model predictions are from the sample points used to fit the model, is shown in the following figure.

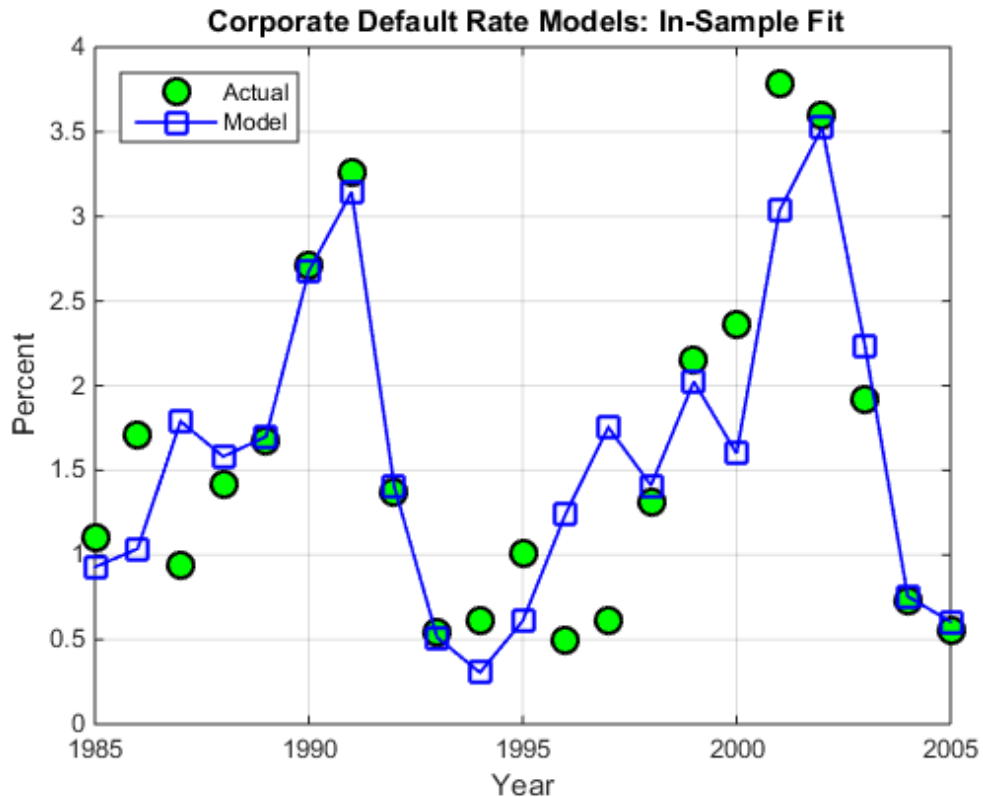
```
bHat = stats.beta;
yHat = [ones(T,1),X]*bHat;

figure
plot(Years(inSample+1),DefRate(inSample+1),'ko','LineWidth',1.5,...
     'MarkerSize',10,'MarkerFaceColor','g')
```

```

hold on
plot(Years(inSample+1),yHat,'b-s','LineWidth',1.2,'MarkerSize',10)
hold off
grid on
legend({'Actual','Model'},'location','NW')
title({'\bf Corporate Default Rate Models: In-Sample Fit'})
xlabel('Year')
ylabel('Percent')

```



It can be shown that there is no strong statistical evidence to conclude that the linear regression assumptions are violated. It is apparent that default rates are not normally distributed. The model, however, does not make that assumption. The only normality assumption in the model is that, given the predictors values, the error between the predicted and the observed default rates is normally distributed. By looking at the in-

sample fit, this does not seem unreasonable. The magnitude of the errors certainly seems independent of whether the default rates are high or low. Year 2001 has a high default rate and a high error, but years 1991 or 2002 also have high rates and yet very small errors. Likewise, low default rate years like 1996 and 1997 show considerable errors, but years 2004 or 2005 have similarly low rates and tiny errors.

A thorough statistical analysis of the model is out of scope here, but there are several detailed examples in Statistics Toolbox™ and Econometrics Toolbox™.

Backtesting

To evaluate how this model performs out-of-sample, we set up a backtesting exercise. Starting at the end of 1995, we fit the linear regression model with the information available up to that date, and compare the model prediction to the actual default rate observed the following year. We repeat the same for all subsequent years until the end of the sample.

For backtesting, relative performance of a model, when compared to alternatives, is easier to assess than the performance of a model in isolation. Here we include two alternatives to determine next year's default rate, both likely candidates in practice. One is the TTC default rate, estimated with data from the beginning of the sample to the current year, a very stable default rate estimate. The other is the PIT rate, estimated using data from the most recent year only, much more sensitive to recent events.

```
XBT = [AGE, CPF, SPR];
yBT = DefRate;

iYear0 = find(Years==1984); % index of first year in sample, 1984
T = find(Years==1995); % ind "current" year, start at 1995, updated in loop
YearsBT = 1996:2005; % years predicted in BT exercise
iYearsBT = find(Years==1996):find(Years==2005); % corresponding indices
nYearsBT = length(YearsBT); % number of years in BT exercise

MethodTags = {'Model', 'PIT', 'TTC'};
nMethods = length(MethodTags);
PredDefRate = zeros(nYearsBT, nMethods);
ErrorBT = zeros(nYearsBT, nMethods);

alpha = 0.05;
PredDefLoBnd = zeros(nYearsBT, 1);
PredDefUpBnd = zeros(nYearsBT, 1);

for k=1:nYearsBT
```

```
% In sample years for predictors, from 1984 to "last" year (T-1)
inSampleBT = iYear0:T-1;

% Method 1: Linear regression model
% Fit regression model with data up to "current" year (T)
s = regstats(yBT(inSampleBT+1),XBT(inSampleBT,:));
% Predict default rate for "next" year (T+1)
PredDefRate(k,1) = [1 XBT(T,:)]*s.beta;
% Compute prediction intervals
tCrit = tinv(1-alpha/2,s.tstat.dfe);
PredStd = sqrt([1 XBT(T,:)]*s.covb*[1 XBT(T,:)]'+s.mse);
PredDefLoBnd(k) = max(0,PredDefRate(k,1) - tCrit*PredStd);
PredDefUpBnd(k) = PredDefRate(k,1) + tCrit*PredStd;

% Method 2: Point-in-time (PIT) default rate
PredDefRate(k,2) = DefRate(T);

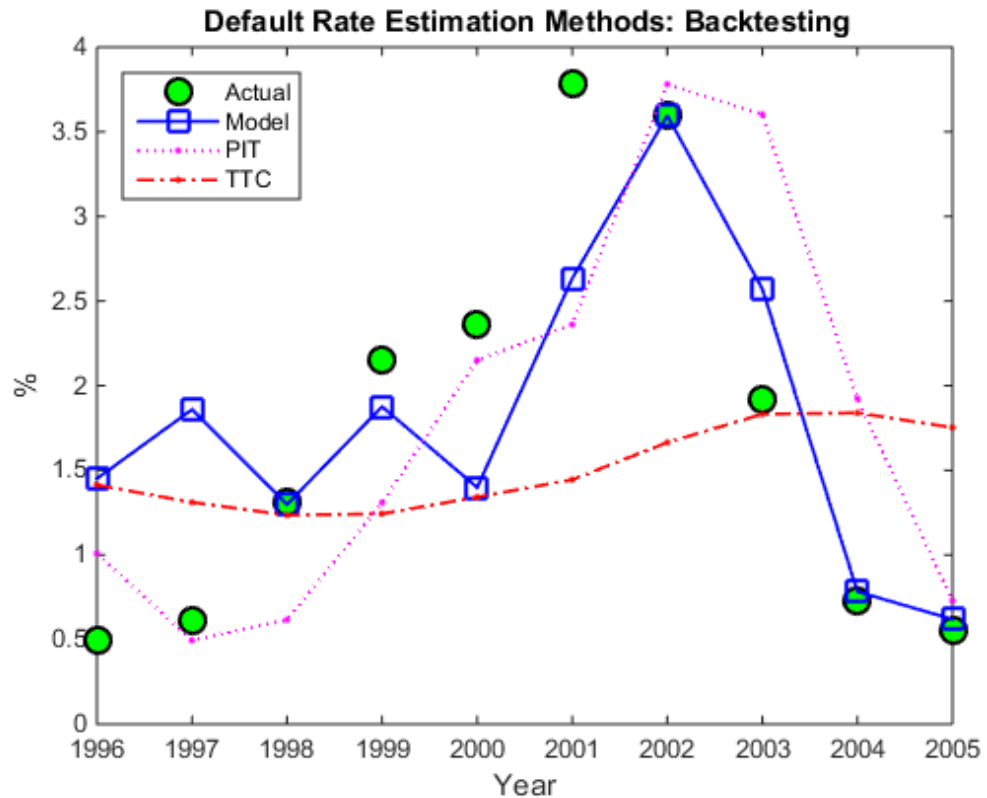
% Method 3: Through-the-cycle (TTC) default rate
tmAll = transprobytotals(totalsAll(iYear0:T));
PredDefRate(k,3) = tmAll(1,2);

% Update error
ErrorBT(k,:) = PredDefRate(k,:) - DefRate(T+1);

% Move to next year
T = T + 1;
end
```

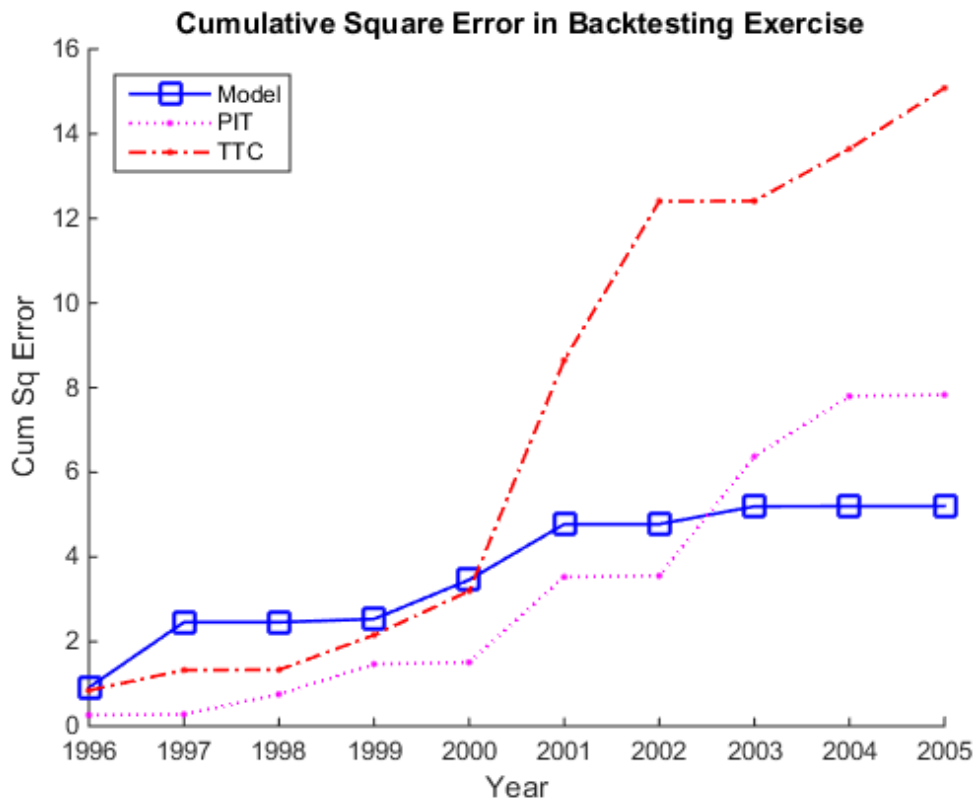
Here are the predictions of the three alternative approaches, compared to the actual default rates observed. Unsurprisingly, TTC shows a very poor predictive power. However, it is not obvious whether PIT or the linear regression model makes better predictions in this 10-year time span.

```
Example_BacktestPlot(YearsBT,DefRate(iYearsBT),PredDefRate,'Year','%',...
    '{\bf Default Rate Estimation Methods: Backtesting}',...
    ['Actual' MethodTags],'NW')
```



The following plot keeps track of cumulative square error, a measure often used for comparisons in backtesting exercises. This confirms TTC as a poor alternative. PIT shows lower cumulative error than the linear regression model in the late nineties, but after the 2001 recession the situation is reversed. Cumulative square error, however, is not an intuitive measure, it is hard to get a sense of what the difference between these alternatives means in practical terms.

```
CumSqError = cumsum(ErrorBT.^2);
Example_BacktestPlot(YearsBT,[],CumSqError,'Year','Cum Sq Error',...
    '\bf Cumulative Square Error in Backtesting Exercise',...
    MethodTags,'NW')
```



It makes sense to translate the prediction errors into a monetary measure. Here we measure the impact of the prediction error on a simplified framework for generating loss reserves in an institution.

We assume a homogeneous portfolio, where all credits have the same probability of default, the same loss given default (LGD), and the same exposure at default (EAD). Both LGD and EAD are assumed to be known. For simplicity, we keep these values constant for the 10 years of the exercise. We set LGD at 45%, and EAD per bond at 100 million. The portfolio is assumed to have a thousand bonds, so the total value of the portfolio, the total EAD, is 100 billion.

The predicted default rate for year t , determined at the end of year $t-1$, is used to calculate the expected loss for year t

$$EL_t = EAD_t \times LGD_t \times PredictedDefaultRate_t$$

This is the amount added to the loss reserves at the start of year t . At the end of the year, the actual losses are known

$$AL_t = EAD_t \times LGD_t \times ObservedDefaultRate_t$$

We assume that unused loss reserves remain in the reserves fund. The starting balance in reserves at the beginning of the exercise is set to zero. If the actual losses surpass the expected loss, unused reserves accumulated over the years are used first, and only if these run out, capital is used to cover a shortfall. All this translates into the following formula

$$Reserves_t = Reserves_{t-1} + (EL_t - AL_t)$$

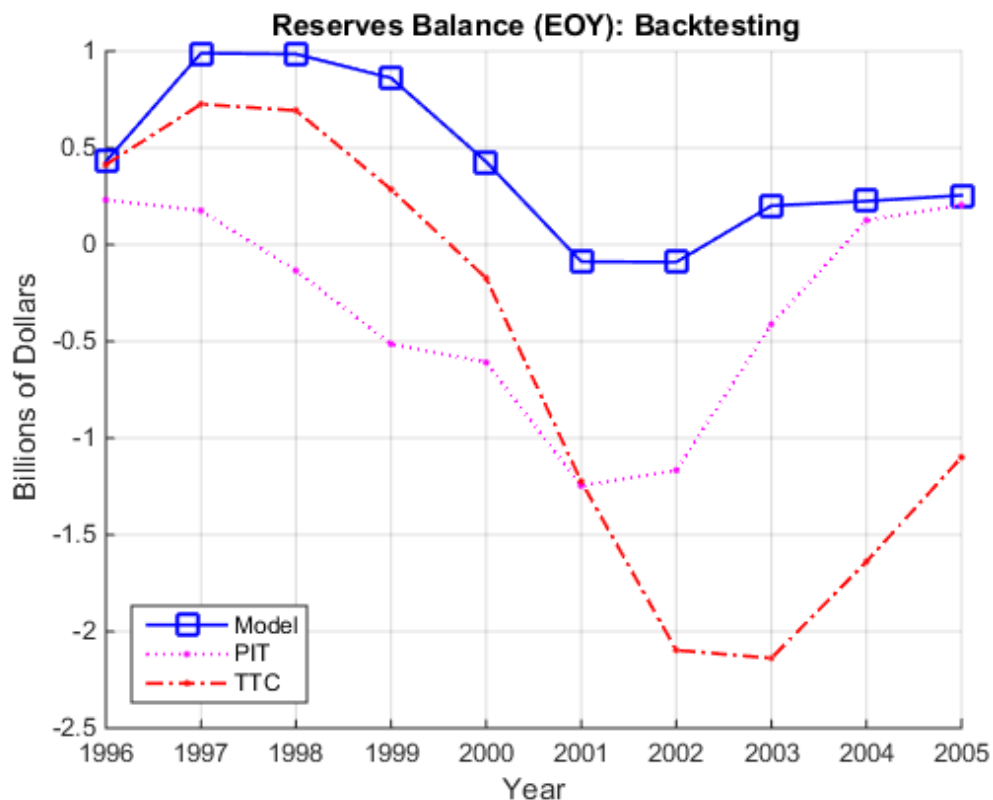
or equivalently

$$Reserves_t = \sum_{s=1}^t (EL_s - AL_s)$$

The following figure shows the loss reserves balance for each of the three alternatives in the backtesting exercise.

```
EAD = 100*ones(nYearsBT,1); % in billions
LGD = 0.45*ones(nYearsBT,1); % Loss given default, 45%
% Reserves excess or shortfall for each year, in billions
ReservesExcessShortfall = bsxfun(@times,EAD.*LGD>ErrorBT/100);
% Cumulative reserve balance for each year, in billions
ReservesBalanceEOY = cumsum(ReservesExcessShortfall);

Example_BacktestPlot(YearsBT,[],ReservesBalanceEOY,'Year',...
    'Billions of Dollars',...
    '\bf Reserves Balance (EOY): Backtesting',...
    MethodTags,'SW')
grid on
```



Using the linear regression model we only observe a deficit in reserves in two out of ten years, and the maximum deficit, in 2001, is 0.09 billion, only nine basis points of the portfolio value.

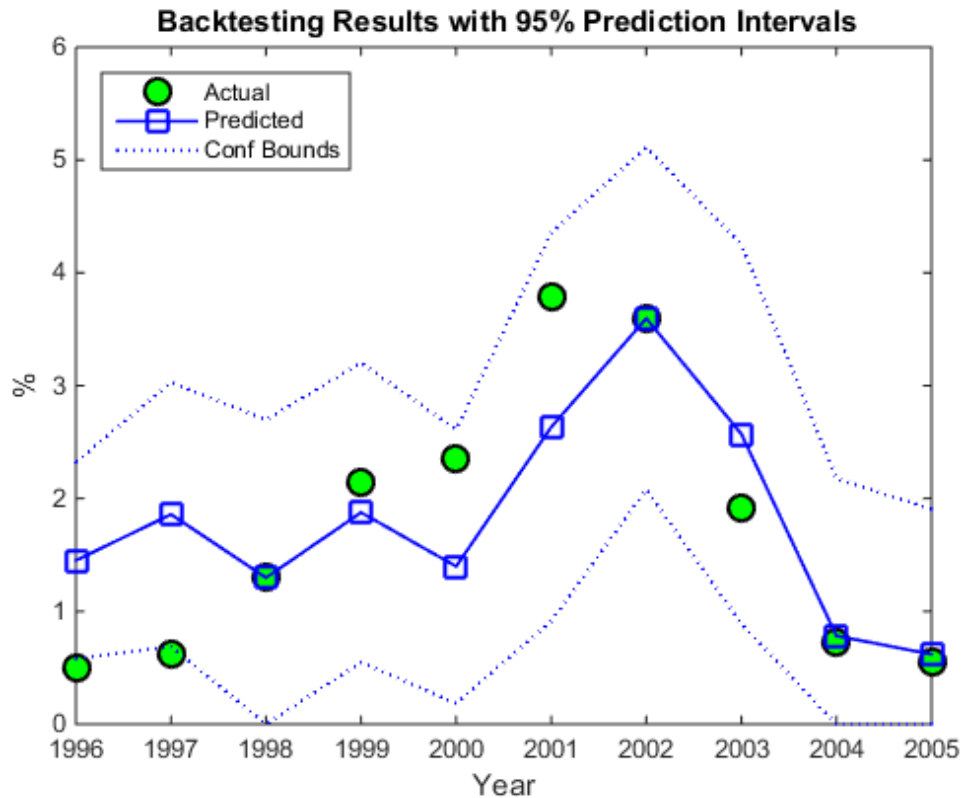
In contrast, both TTC and PIT reach a deficit of 1.2 billion by 2001. Things get worse for TTC in the next two years, reaching a deficit of 2.1 billion by 2003. PIT does make a correction quickly after 2001, and by 2004 the reserves have a surplus. Yet, both TTC and PIT lead to more deficit years than surplus years in this exercise.

The linear regression model shows more of a counter-cyclical effect than the alternatives in this exercise. The money set aside using the linear regression model reaches close to a billion in 1997 and 1998. High levels of unused reserves translate into a slower pace of lending (not reflected in the exercise, because we exogenously impose the portfolio

value). Moreover, capital is only slightly impacted during the 2001 recession thanks to the reserves accumulated over the previous expansion. This translates into more capital available to back up further lending, if desired, during the economic recovery.

The last backtesting tool we discuss is the use of prediction intervals. Linear regression models provide standard formulas to compute confidence intervals for the values of new observations. These intervals are shown in the next figure for the 10 years spanned in the backtesting exercise.

```
figure
plot(YearsBT,DefRate(iYearsBT),'ko','LineWidth',1.5,'MarkerSize',10,...
     'MarkerFaceColor','g')
hold on
plot(YearsBT,PredDefRate(:,1),'b-s','LineWidth',1.2,'MarkerSize',10)
plot(YearsBT,[PredDefLoBnd PredDefUpBnd],'b:','LineWidth',1.2)
hold off
strConf = num2str((1-alpha)*100);
title(['\bf Backtesting Results with ' strConf '% Prediction Intervals'])
xlabel('Year');
ylabel('%');
legend({'Actual','Predicted','Conf Bounds'},'location','NW');
```



The observed default rates fall outside the prediction intervals for two years, 1996 and 1997, where very low default rates are observed. For a 95% confidence level, two out of ten seems high. Yet, the observed values in these cases fall barely outside the prediction interval, which is a positive sign for the model. It is also positive that the prediction intervals contain the observed values around the 2001 recession.

Stress Testing

Stress testing is a broad area that reaches far beyond computational tools; see, for example, [3]. We show here some tools that can be incorporated into a comprehensive stress testing framework. We build on the linear regression model presented above, but the concepts and tools are compatible with other forecasting methodologies.

The first tool is the use of prediction intervals to define a worst-case scenario forecasts. This is to account for uncertainty in the model only, not in the value of the predictors.

We take a baseline scenario of predictors, in our case, the latest known values of our age proxy *AGE*, corporate profits forecast *CPF* and corporate spread *SPR*. We then use the linear regression model to compute a 95% confidence upper bound for the predicted default rate. The motivation for this is illustrated in the last plot of the backtesting section, where the 95% confidence upper limit acts as a conservative bound when the prediction underestimates the actual default rates.

```
tCrit = tinv(1-alpha/2,stats.tstat.dfe);
XLast = [AGE(end),CPF(end),SPR(end)];

yPred = [1 XLast]*stats.beta;
PredStd = sqrt([1 XLast]*stats.covb*[1 XLast]'+stats.mse);
yPredUB = yPred + tCrit*PredStd;

fprintf('\nPredicted default rate:\n');
fprintf('    Baseline: %4.2f%\n',yPred);
fprintf('    %g% Upper Bound: %4.2f%\n',(1-alpha)*100,yPredUB);
```

```
Predicted default rate:
    Baseline: 1.18%
    95% Upper Bound: 2.31%
```

The next step is to incorporate stressed scenarios of the predictors in the analysis. *CPF* and *SPR* can change in the short term, whereas *AGE* cannot. This is important. The corporate profits forecast and the corporate spread are influenced by world events, including, for example, natural disasters. These predictors can significantly change overnight. On the other hand, *AGE* depends on managerial decisions that can alter the proportion of old and new loans in time, but these decisions take months, if not years, to reflect in the *AGE* time series. Scenarios for *AGE* are compatible with longer term analyses. Here we look at one year ahead only, and keep *AGE* fixed for the remainder of this section.

It is convenient to define the predicted default rate and the confidence bounds as functions of *CPF* and *SPR* to simplify the scenario analysis.

```
yPredFn = @(cpf,spr) [1 AGE(end) cpf spr]*stats.beta;
PredStdFn = @(cpf,spr) sqrt([1 AGE(end) cpf spr]*stats.covb*...
    [1 AGE(end) cpf spr]'+stats.mse);
yPredUBFn = @(cpf,spr) (yPredFn(cpf,spr) + tCrit*PredStdFn(cpf,spr));
yPredLBFn = @(cpf,spr) (yPredFn(cpf,spr) - tCrit*PredStdFn(cpf,spr));
```

Two extreme scenarios of interest can be a drop in the corporate profits forecast of 4% relative to the baseline, and an increase in the corporate spread of 100 basis points over the baseline.

Moving one predictor at a time is not unreasonable in this case, because the correlation between CPF and SPR is very low. Moderate correlation levels may require perturbing predictors together to get more reliable results. Highly correlated predictors usually do not coexist in the same model, since they offer redundant information.

```
fprintf('\n\n          What-if Analysis\n');
fprintf('Scenario          LB    Pred    UB\n');
cpf = CPF(end)-4;
spr = SPR(end);
yPredRange = [yPredLBFn(cpf, spr), yPredFn(cpf, spr), yPredUBFn(cpf, spr)];
fprintf('CPF drops 4%      %4.2f%% %4.2f%% %4.2f%%\n', yPredRange);
cpf = CPF(end);
spr = SPR(end)+1;
yPredRange = [yPredLBFn(cpf, spr), yPredFn(cpf, spr), yPredUBFn(cpf, spr)];
fprintf('SPR rises 1%      %4.2f%% %4.2f%% %4.2f%%\n', yPredRange);
cpf = CPF(end);
spr = SPR(end);
yPredRange = [yPredLBFn(cpf, spr), yPredFn(cpf, spr), yPredUBFn(cpf, spr)];
fprintf('    Baseline      %4.2f%% %4.2f%% %4.2f%%\n', yPredRange);

fprintf('\nCorrelation between CPF and SPR: %4.3f\n', corr(CPF, SPR));
```

What-if Analysis			
Scenario	LB	Pred	UB
CPF drops 4%	0.42%	1.57%	2.71%
SPR rises 1%	0.71%	1.88%	3.05%
Baseline	0.04%	1.18%	2.31%

Correlation between CPF and SPR: 0.012

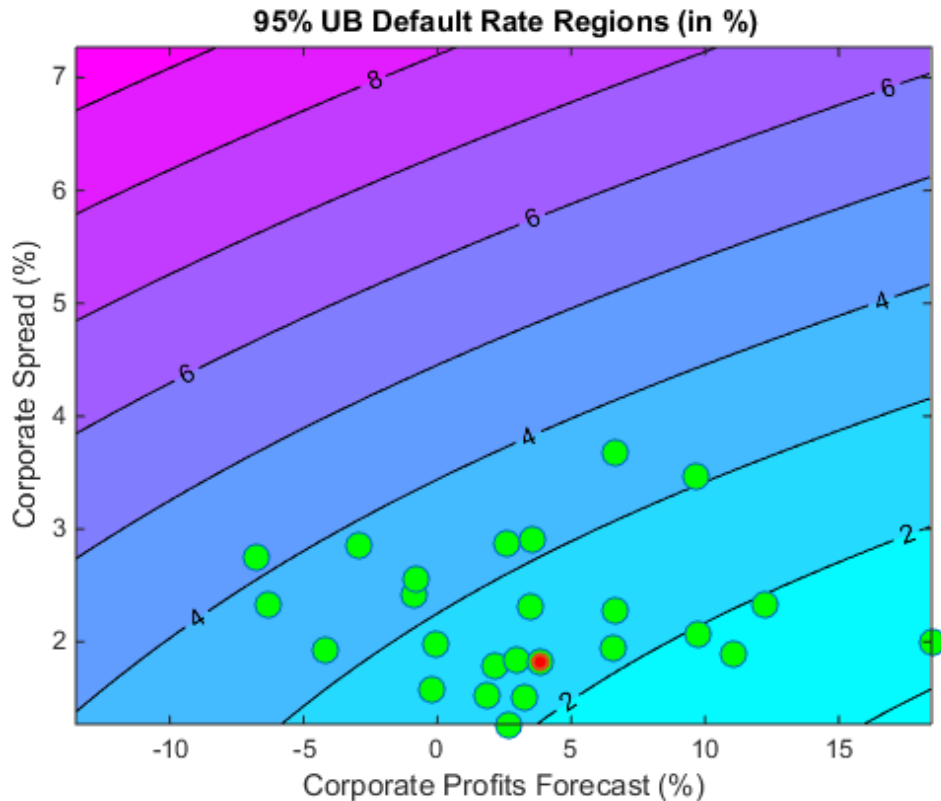
We now take a more global view of the scenario analysis. Instead of analyzing one scenario at a time, we visualize the default rate forecasts as a function of CPF and SPR. More precisely, we plot default rate contours over a whole grid of CPF and SPR values. We use the conservative 95% upper bound.

If we assumed a particular bivariate distribution for the values of CPF and SPR, we could plot the contours of their distribution in the same figure. That would give visual

information on the probability of falling on each region. Lacking such a distribution, we simply add to the plot the CPF - SPR pairs observed in our sample, as a historical, empirical distribution. The last observation in the sample, the baseline scenario, is marked in red.

```
gridCPF = 2*min(CPF):0.1:max(CPF);
gridSPR = min(SPR):0.1:2*max(SPR);
nGridCPF = length(gridCPF);
nGridSPR = length(gridSPR);

DefRateUB = zeros(nGridCPF,nGridSPR);
for i=1:nGridCPF
    for j=1:nGridSPR
        DefRateUB(i,j) = yPredUBFn(gridCPF(i),gridSPR(j));
    end
end
Example_StressTestPlot(gridCPF,gridSPR,DefRateUB,CPF,SPR,...
    'Corporate Profits Forecast (%)','Corporate Spread (%)',...
    [{'\bf ' strConf '% UB Default Rate Regions (in %)}'])
```



Very different predictor values result in similar default rate levels. For example, consider a profits forecast around 10% with a spread of 3.5%, and a profits forecast of -2.5% with a spread of 2%, they both result in a default rate slightly above 3%. Also, only one point in the available history yields a default rate higher than 4%.

Monetary terms, once again, may be more meaningful. We use Basel II's capital requirements formula (see [2]) to translate the default rates into a monetary measure. Basel II's formula is convenient because it is analytic (there is no need to simulate to estimate the capital requirements), but also because it depends only on the probabilities of default. We define Basel II's capital requirements as a function K .

```
% Correlation as a function of PD
w = @(pd) (1-exp(-50*pd))/(1-exp(-50)); % weight
R = @(pd) (0.12*w(pd)+0.24*(1-w(pd))); % correlation
```

```

% Vasicek formula
V = @(pd) normcdf(norminv(pd)+R(pd).*norminv(0.999)./sqrt(1-R(pd)));
% Parameter b for maturity adjustment
b = @(pd) (0.11852-0.05478*log(pd)).^2;
% Basel II capital requirement with LGD=45% and maturity M=2.5 (numerator
% in maturity adjustment term becomes 1)
K = @(pd) 0.45*(V(pd)-pd).*(1./(1-1.5*b(pd)));

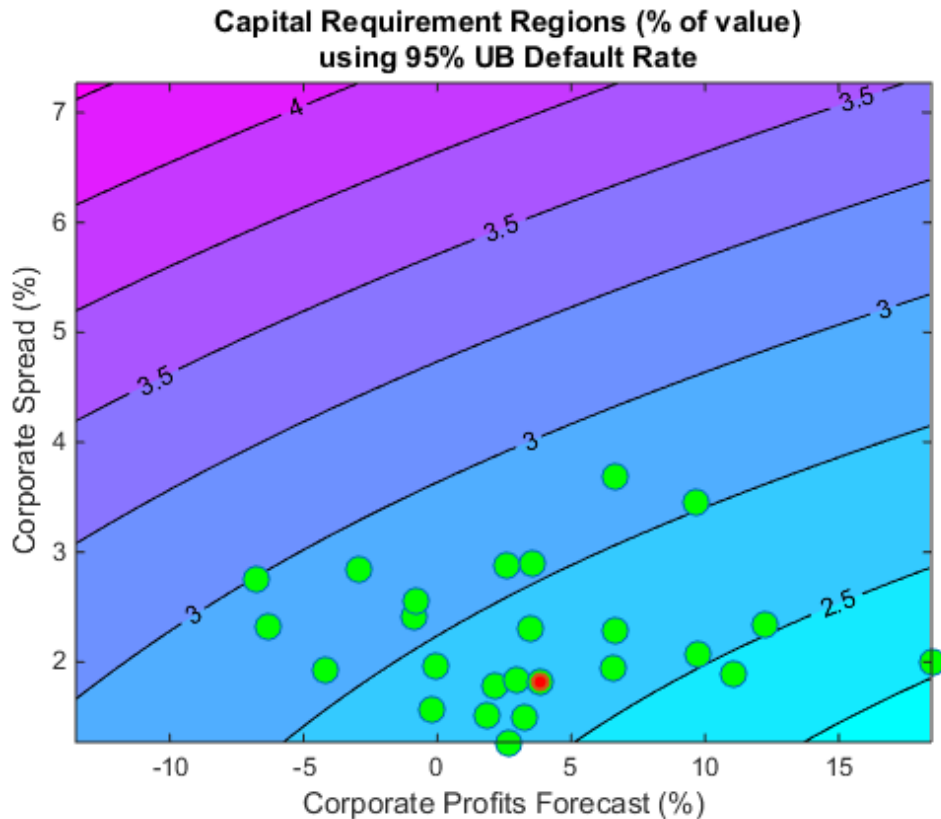
```

Worst-case default rates for a whole grid of CPF - SPR pairs are stored in DefRateUB. By applying the function K to DefRateUB we can visualize the capital requirements over the same grid.

```

CapReq = 100*K(DefRateUB/100);
Example_StressTestPlot(gridCPF,gridSPR,CapReq,CPF,SPR,...
    'Corporate Profits Forecast (%)','Corporate Spread (%)',...
    {'{\bf Capital Requirement Regions (% of value)}';...
    [{'\bf using ' strConf '% UB Default Rate}']})

```



The contour levels now indicate capital requirements as a percentage of portfolio value. The two scenarios above, profits of 10% with spread of 3.5%, and profits of -2.5% and spread of 2%, result in capital requirements near 2.75%. The worst-case point from the historical data yields a capital requirement of about 3%.

This visualization can also be used, for example, as part of a reverse stress test analysis. Critical levels of capital can be determined first, and the figure can be used to determine regions of risk factor values (in this case CPF and SPR) that lead to those critical levels.

Instead of historical observations of CPF and SPR, an empirical distribution for the risk factors can be simulated using, for example, a vector autoregressive (VAR) model from Econometrics Toolbox™. The capital requirements corresponding to each default probability level can be found by simulation if a closed form formula is not available,

and the same plots can be generated. For large simulations, a distributed computing implementation using Parallel Computing Toolbox™ or MATLAB Distributed Computing Server™ can make the process more efficient.

Appendix: Modeling Full Transition Matrices

Transition matrices change in time, and a full description of their dynamics requires working with multi-dimensional time series. There are, however, techniques that exploit the particular structure of transition matrices to reduce the dimensionality of the problem. In [8], for example, a single parameter related to the proportion of downgrades is used, and both [6] and [8] describe a method to shift transition probabilities using a single parameter. The latter approach is shown in this appendix.

The method takes the TTC transition matrix as a baseline.

```
tmTTC = transprobytotals(totalsByRtg);
Example_DisplayTransitions(tmTTC,[],...
    {'AAA','AA','A','BBB','BB','B','CCC'},...
    {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	88.2	7.67	0.49	0.09	0.06	0	0	0	3.49
AA	0.58	87.16	7.63	0.58	0.06	0.11	0.02	0.01	3.85
A	0.05	1.9	87.24	5.59	0.42	0.15	0.03	0.04	4.58
BBB	0.02	0.16	3.85	84.13	4.27	0.76	0.17	0.27	6.37
BB	0.03	0.04	0.25	5.26	75.74	7.36	0.9	1.12	9.29
B	0	0.05	0.19	0.31	5.52	72.67	4.21	5.38	11.67
CCC	0	0	0.28	0.41	1.24	10.92	47.06	27.02	13.06

An equivalent way to represent this matrix is by transforming it into credit quality thresholds, that is, critical values of a standard normal distribution that yield the same transition probabilities (row by row).

```
thresholdMat = transprobt thresholds(tmTTC);
Example_DisplayTransitions(thresholdMat,[],...
    {'AAA','AA','A','BBB','BB','B','CCC'},...
    {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

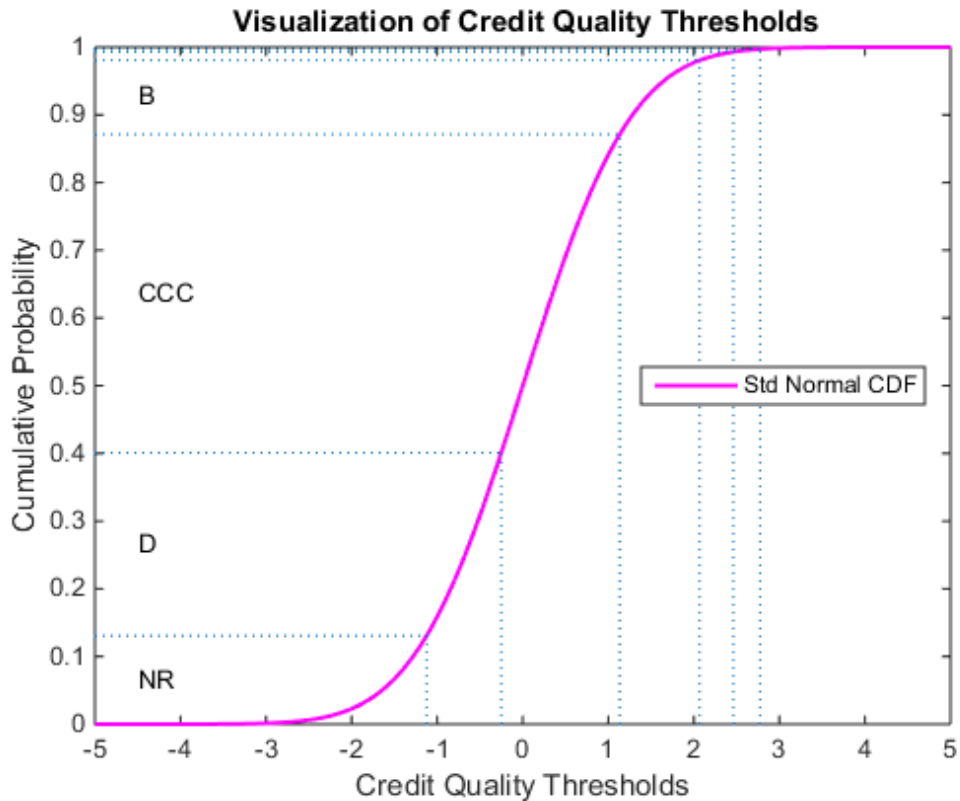
	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	Inf	-1.19	-1.74	-1.8	-1.81	-1.81	-1.81	-1.81	-1.81
AA	Inf	2.52	-1.16	-1.68	-1.75	-1.75	-1.76	-1.77	-1.77
A	Inf	3.31	2.07	-1.24	-1.62	-1.66	-1.68	-1.68	-1.69
BBB	Inf	3.57	2.91	1.75	-1.18	-1.43	-1.49	-1.5	-1.52

BB	Inf	3.39	3.16	2.72	1.59	-0.89	-1.21	-1.26	-1.32
B	Inf	Inf	3.28	2.82	2.54	1.55	-0.8	-0.95	-1.19
CCC	Inf	Inf	Inf	2.77	2.46	2.07	1.13	-0.25	-1.12

Credit quality thresholds are illustrated in the following figure. The segments in the vertical axis represent transition probabilities, and the boundaries between them determine the critical values in the horizontal axis, via the standard normal distribution. Each row in the transition matrix determines a set of thresholds. The figure shows the thresholds for the 'CCC' rating.

```
xliml = -5;
xlimr = 5;
step = 0.1;
x=xliml:step:xlimr;
thresCCC = thresholdMat(7,:);
centersY = (normcdf([thresCCC(2:end) xliml])+...
    normcdf([xlimr thresCCC(2:end)]))/2;
labels = {'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D', 'NR'};

figure
plot(x,normcdf(x),'m','LineWidth',1.5)
for i=2:length(labels)
    val = thresCCC(i);
    line([val val],[0 normcdf(val)],'LineStyle',':');
    line([x(1) val],[normcdf(val) normcdf(val)],'LineStyle',':');
    if (centersY(i-1)-centersY(i))>0.05
        text(-4.5,centersY(i),labels{i});
    end
end
xlabel('Credit Quality Thresholds')
ylabel('Cumulative Probability')
title('\bf Visualization of Credit Quality Thresholds')
legend('Std Normal CDF', 'Location', 'E')
```

Shifting the critical values to the right or left changes the transition probabilities. For example, here is the transition matrix obtained by shifting the TTC thresholds by 0.5 to the right. Note that default probabilities increase.

```
shiftedThresholds = thresholdMat+0.5;
Example_DisplayTransitions(transprobfromthresholds(shiftedThresholds),...
    [], {'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC'},...
    {'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D', 'NR'})
```

	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	75.34	13.84	1.05	0.19	0.13	0	0	0	9.45
AA	0.13	74.49	13.53	1.21	0.12	0.22	0.04	0.02	10.24
A	0.01	0.51	76.4	10.02	0.83	0.31	0.06	0.08	11.77
BBB	0	0.03	1.2	74.03	7.22	1.39	0.32	0.51	15.29
BB	0	0.01	0.05	1.77	63.35	10.94	1.47	1.88	20.52

B	0	0.01	0.04	0.07	1.91	59.67	5.74	8.1	24.46
CCC	0	0	0.05	0.1	0.36	4.61	35.06	33.18	26.65

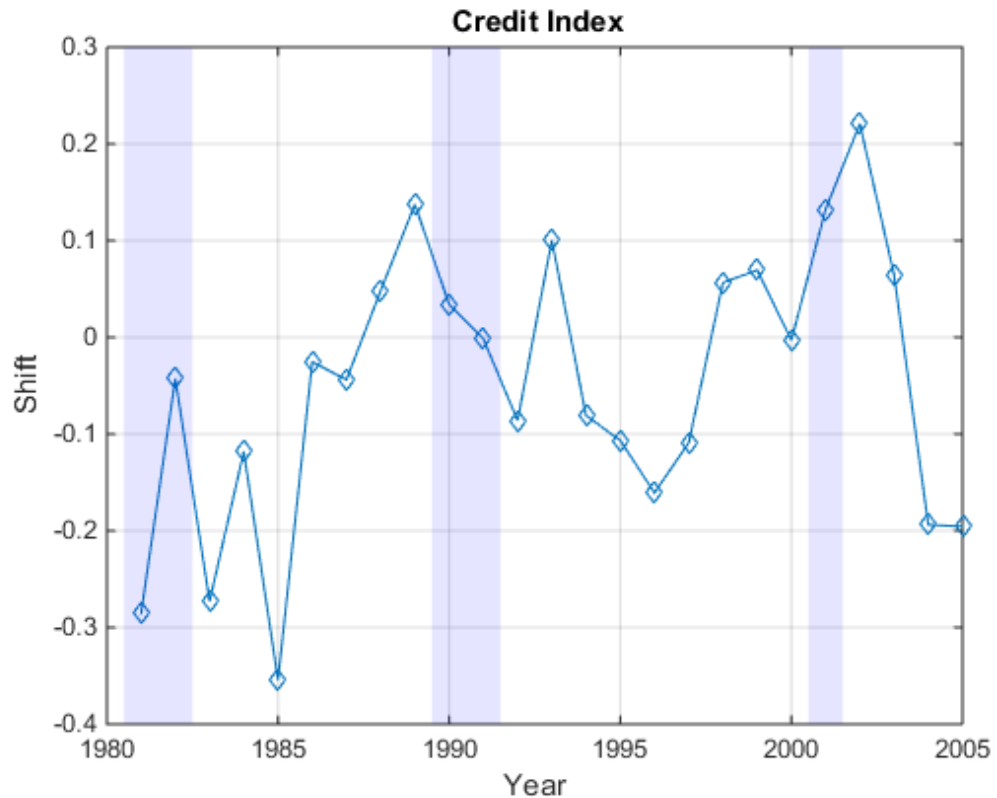
Given a particular PIT matrix, the idea in [6] and [8] is to vary the shifting parameter applied to the TTC thresholds so that the resulting transition matrix is as close as possible to the PIT matrix. Closeness is measured as the sum of squares of differences between corresponding transition probabilities. The optimal shifting value is called credit index. A credit index is determined for every PIT transition matrix in the sample.

Here we use `fminunc` from Optimization Toolbox™ to find the credit indices.

```
CreditIndex = zeros(nYears,1);
ExitFlag = zeros(nYears,1);
options = optimset('LargeScale','Off','Display','Off');
for i=1:nYears
    errorfun = @(z)norm(squeeze(TransMat(:,:,i))-...
        transprobfromthresholds(...
            transprobtothresholds(tmTTC)+z),'fro');
    [CreditIndex(i),~,ExitFlag(i)] = fminunc(errorfun,0,options);
end
```

In general, one expects that higher credit indices correspond to riskier years. The series of credit indices found does not entirely match this pattern. There may be different reasons for this. First, transition probabilities may deviate from their long-term averages in different ways that may lead to confounding effects in the single parameter trying to capture these difference, the credit index. Having separate credit indices for IG and SG, for example, may help separate confounding effects. Second, a difference of five basis points may be very significant for the 'BBB' default rate, but not as important for the 'CCC' default rate, yet the norm used weights them equally. Other norms can be considered. Also, it is always a good idea to check the exit flags of optimization solvers, in case the algorithm could not find a solution. Here we get valid solutions for each year (all exit flags are 1).

```
figure
plot(Years,CreditIndex,'-d')
hold on
Example_RecessionBands
hold off
grid on
xlabel('Year')
ylabel('Shift')
title('{\bf Credit Index}')
```



The workflow above can be adapted to work with the series of credit indices instead of the series of corporate default rates. A model can be fit to predict a credit index for the following year, and a predicted transition matrix can be inferred and used for risk analyses.

References

[1] Altman, E., and E. Hotchkiss, *Corporate Financial Distress and Bankruptcy*, third edition, New Jersey: Wiley Finance, 2006.

[2] Basel Committee on Banking Supervision, "International Convergence of Capital Measurement and Capital Standards: A Revised Framework," Bank for International Settlements (BIS), comprehensive version, June 2006. Available at: <http://www.bis.org/publ/bcbsca.htm>.

- [3] Basel Committee on Banking Supervision, "Principles for Sound Stress Testing Practices and Supervision - Final Paper," Bank for International Settlements (BIS), May 2009. Available at: <http://www.bis.org/publ/bcb155.htm>.
- [4] FRED, St. Louis Federal Reserve, Federal Reserve Economic Database, <http://research.stlouisfed.org/fred2/>.
- [5] Helwege, J., and P. Kleiman, "Understanding Aggregate Default Rates of High Yield Bonds," Federal Reserve Bank of New York, Current Issues in Economics and Finance, Volume 2, Number 6, May, 1996.
- [6] Loeffler, G., and P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.
- [7] NBER, National Bureau of Economic Research, Business Cycle Expansions and Contractions, <http://www.nber.org/cycles/>.
- [8] Otani, A., S. Shiratsuka, R. Tsurui, and T. Yamada, "Macro Stress-Testing on the Loan Portfolio of Japanese Banks," Bank of Japan Working Paper Series No.09-E-1, March, 2009.
- [9] Survey of Professional Forecasters, Federal Reserve Bank of Philadelphia, <http://www.philadelphiafed.org/>.
- [10] Vazza, D., D. Aurora, and R. Schneck, "Annual 2005 Global Corporate Default Study And Rating Transitions," Standard & Poor's, Global Fixed Income Research, New York, January, 2006.
- [11] Wilson, T. C., "Portfolio Credit Risk," FRBNY Economic Policy Review, October 1998.

Credit Quality Thresholds

In this section...

“Introduction” on page 8-69

“Compute Credit Quality Thresholds” on page 8-69

“Visualize Credit Quality Thresholds” on page 8-70

Introduction

An equivalent way to represent transition probabilities is by transforming them into credit quality thresholds. These are critical values of a standard normal distribution that yield the same transition probabilities.

An M-by-N matrix of transition probabilities **TRANS** and the corresponding M-by-N matrix of credit quality thresholds **THRESH** are related as follows. The thresholds **THRESH(i,j)** are critical values of a standard normal distribution z , such that

$$\text{TRANS}(i,N) = P[z < \text{THRESH}(i,N)],$$

$$\text{TRANS}(i,j) = P[z < \text{THRESH}(i,j)] - P[z < \text{THRESH}(i,j+1)], \text{ for } 1 \leq j < N$$

Financial Toolbox supports the transformation between transition probabilities and credit quality thresholds with the functions `transprobtothresholds` and `transprobfromthresholds`.

Compute Credit Quality Thresholds

To compute credit quality thresholds, transition probabilities are required as input. Here is a transition matrix estimated from credit ratings data:

```
load Data_TransProb
trans = transprob(data)
```

```
trans =
```

```

93.1170   5.8428   0.8232   0.1763   0.0376   0.0012   0.0001   0.0017
 1.6166  93.1518   4.3632   0.6602   0.1626   0.0055   0.0004   0.0396
 0.1237   2.9003  92.2197   4.0756   0.5365   0.0661   0.0028   0.0753
 0.0236   0.2312   5.0059  90.1846   3.7979   0.4733   0.0642   0.2193
 0.0216   0.1134   0.6357   5.7960  88.9866   3.4497   0.2919   0.7050
 0.0010   0.0062   0.1081   0.8697   7.3366  86.7215   2.5169   2.4399
 0.0002   0.0011   0.0120   0.2582   1.4294   4.2898  81.2927  12.7167
      0          0          0          0          0          0          0  100.0000
```

Convert the transition matrix to credit quality thresholds using `transprobtothresholds`:

```

thresh = transprobt thresholds(trans)
thresh =
    Inf    -1.4846    -2.3115    -2.8523    -3.3480    -4.0083    -4.1276    -4.1413
    Inf    2.1403    -1.6228    -2.3788    -2.8655    -3.3166    -3.3523    -3.3554
    Inf    3.0264    1.8773    -1.6690    -2.4673    -2.9800    -3.1631    -3.1736
    Inf    3.4963    2.8009    1.6201    -1.6897    -2.4291    -2.7663    -2.8490
    Inf    3.5195    2.9999    2.4225    1.5089    -1.7010    -2.3275    -2.4547
    Inf    4.2696    3.8015    3.0477    2.3320    1.3838    -1.6491    -1.9703
    Inf    4.6241    4.2097    3.6472    2.7803    2.1199    1.5556    -1.1399
    Inf         Inf         Inf         Inf         Inf         Inf         Inf         Inf

```

Conversely, given a matrix of thresholds, you can compute transition probabilities using `transprobfromthresholds`. For example, take the thresholds computed previously as input to recover the original transition probabilities:

```

trans1 = transprobfromthresholds(thresh)
trans1 =
    93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001    0.0017
    1.6166    93.1518    4.3632    0.6602    0.1626    0.0055    0.0004    0.0396
    0.1237    2.9003    92.2197    4.0756    0.5365    0.0661    0.0028    0.0753
    0.0236    0.2312    5.0059    90.1846    3.7979    0.4733    0.0642    0.2193
    0.0216    0.1134    0.6357    5.7960    88.9866    3.4497    0.2919    0.7050
    0.0010    0.0062    0.1081    0.8697    7.3366    86.7215    2.5169    2.4399
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898    81.2927    12.7167
    0         0         0         0         0         0         0         100.0000

```

Visualize Credit Quality Thresholds

You can graphically represent the relationship between credit quality thresholds and transition probabilities. Here, this example shows the relationship for the 'CCC' credit rating. In the plot, the thresholds are marked by the vertical lines and the transition probabilities are the area below the standard normal density curve:

```

load Data_TransProb
trans = transprob(data);
thresh = transprobt thresholds(trans);

xliml = -5;
xlimr = 5;
step = 0.1;
x=xliml:step:xlimr;
thresCCC = thresh(7,:);
labels = {'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D'};

centersX = ([5 thresCCC(2:end)]+[thresCCC(2:end) -5])*0.5;
omag = round(log10(trans(7,:)));
omag(omag>0)=omag(omag>0).^2;

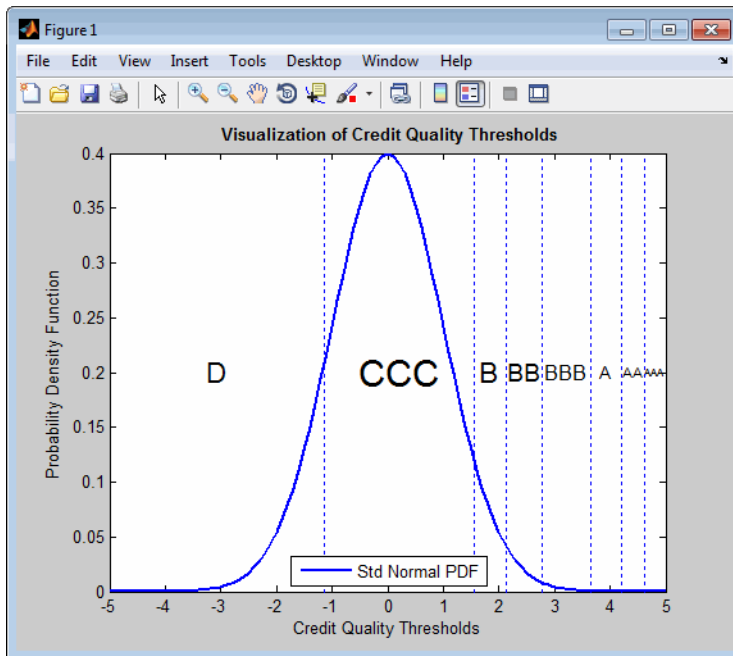
```

```

fs = 14+2*omag;

figure
plot(x,normpdf(x),'LineWidth',1.5)
text(centersX(1),0.2,labels{1},'FontSize',fs(1),...
     'HorizontalAlignment','center')
for i=2:length(labels)
    val = thresCCC(i);
    line([val val],[0 0.4],'LineStyle',':')
    text(centersX(i),0.2,labels{i},'FontSize',fs(i),...
         'HorizontalAlignment','center')
end
xlabel('Credit Quality Thresholds')
ylabel('Probability Density Function')
title('\bf Visualization of Credit Quality Thresholds')
legend('Std Normal PDF','Location','S')

```

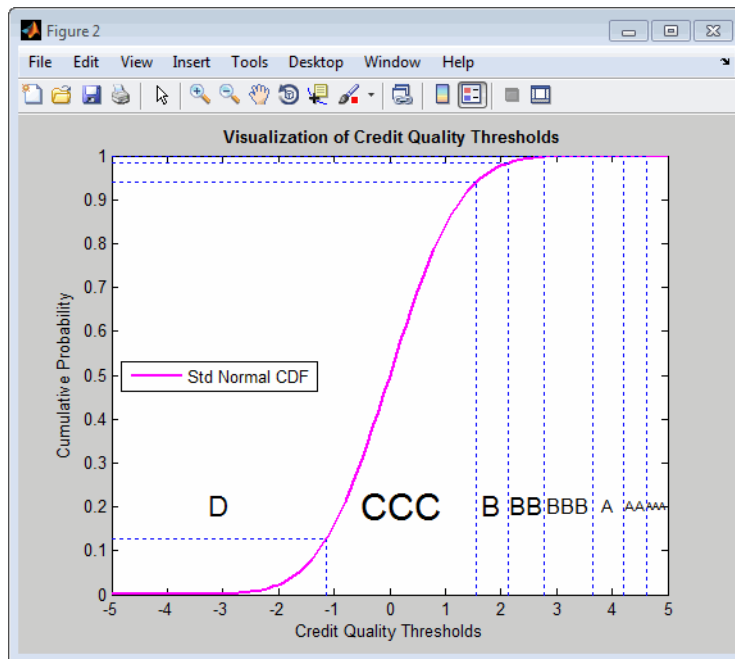


The second plot uses the cumulative density function instead. The thresholds are represented by vertical lines. The transition probabilities are given by the distance between horizontal lines.

```

figure
plot(x,normcdf(x),'m','LineWidth',1.5)
text(centersX(1),0.2,labels{1},'FontSize',fs(1),...
    'HorizontalAlignment','center')
for i=2:length(labels)
    val = thresCCC(i);
    line([val val],[0 normcdf(val)],'LineStyle',':');
    line([x(1) val],[normcdf(val) normcdf(val)],'LineStyle',':');
    text(centersX(i),0.2,labels{i},'FontSize',fs(i),...
        'HorizontalAlignment','center')
end
xlabel('Credit Quality Thresholds')
ylabel('Cumulative Probability')
title('\bf Visualization of Credit Quality Thresholds')
legend('Std Normal CDF','Location','W')

```



About Credit Scorecards

In this section...
“What Is a Credit Scorecard?” on page 8-73
“Credit Scorecard Development Process” on page 8-76

What Is a Credit Scorecard?

Credit scoring is one of the most widely used credit risk analysis tools. The goal of credit scoring is ranking borrowers by their credit worthiness. In the context of retail credit (credit cards, mortgages, car loans, etc.), credit scoring is usually performed using a credit scorecard. Credit scorecards represent different characteristics of a customer (age, residential status, time at current address, time at current job, and so on) translated into points and the total number of points becomes the credit score. The credit worthiness of customers is summarized by their credit score; high scores usually correspond to low-risk customers, and vice versa. Scores are also used for corporate credit analysis of small and medium enterprises, as well as, large corporations.

A credit scorecard is a lookup table that maps specific characteristics of a borrower into points. The total number of points becomes the credit score. Credit scorecards are a widely used type of credit scoring model. As such, the goal of a credit scorecard is to distinguish between customers who will repay their loans (“good” customers), and customers who won’t (“bad” customers). Like other credit scoring models, credit scorecards quantify the risk that a borrower will not repay a loan in the form of a score and a probability of default.

For example, a credit scorecard can give individual borrowers points for their age and income according to the following table. Other characteristics such as residential status, employment status, and so on, might also be included, although, for brevity, they are not shown in this table.

Age	Points
Up to 25	10
26 to 40	25
41 to 65	38
66 and up	43
Income	
Up to 40k	16
40k to 70k	28
...	
Total score	(Sum of Points)

Using the credit scorecard in this example, a particular customer who is 31 and has an income of \$52,000 a year, is placed into the second age group (26 to 40) and receives 25 points for his age, and similarly, receives 28 points for his income. Other characteristics (not shown here) might contribute additional points to his score. The total score is the sum of all points, which in this example is assumed to give the customer a total of 238 points (this is a fictitious example on an arbitrary scoring scale).

Age	Points
Up to 25	10
26 to 40	25
41 to 65	38
66 and up	43
Income	
Up to 40k	16
40k to 70k	28
...	
Total score	238

John:

- 31 years old
- 52k a year
- Single
- ...

Score 238

Technically, to determine the credit scorecard points, you must start out by selecting a set of potential predictors (column 1 in the next figure). Then you need to bin data into groups (for example, ages 'Up to 25', '25 to 40', and so on (column 2 in the figure). This grouping helps to distinguish between "good" and "bad" customers. The Weight of Evidence (WOE) is a way to measure how well the distribution of "good" and "bad" are separated across bins or groups for each individual predictor (column 3 in the figure). By fitting a logistic regression model, you can identify which predictors, when put together, do a better job distinguishing between "good" and "bad" customers. The model is summarized by its coefficients (column 4 in the figure). Finally, the combination of WOE's and model coefficients (commonly scaled, shifted, and rounded) make up the scorecard points (column 5 in the figure).

Predictor	Bin	WOE	Model	Points*
Age			β_{age}	
	'Up to 25'	$\text{WOE}_{\text{age}}(\text{'Up to 25'})$		$\beta_{\text{age}} * \text{WOE}_{\text{age}}(\text{'Up to 25'})$
	'26 to 40'	$\text{WOE}_{\text{age}}(\text{'26 to 40'})$		$\beta_{\text{age}} * \text{WOE}_{\text{age}}(\text{'26 to 40'})$
	'41 to 65'	$\text{WOE}_{\text{age}}(\text{'41 to 65'})$		$\beta_{\text{age}} * \text{WOE}_{\text{age}}(\text{'41 to 65'})$
	'66 and up'	$\text{WOE}_{\text{age}}(\text{'66 and up'})$		$\beta_{\text{age}} * \text{WOE}_{\text{age}}(\text{'66 and up'})$
Income			β_{income}	
	'Up to 40k'	$\text{WOE}_{\text{income}}(\text{'Up to 40k'})$		$\beta_{\text{income}} * \text{WOE}_{\text{income}}(\text{'Up to 40k'})$
	'40k to 70k'	$\text{WOE}_{\text{income}}(\text{'40k to 70k'})$		$\beta_{\text{income}} * \text{WOE}_{\text{income}}(\text{'40k to 70k'})$
	...			

* Points may include a constant and may be scaled and rounded.

Credit Scorecard Development Process

1 Data gathering and preparation phase

This includes data gathering and integration, such as querying, merging, aligning, and so on. It also includes treatment of missing information and outliers. There is a prescreening step based on reports of association measures between the predictors and the response variable. Finally, there is a sampling step, to produce a training set, sometimes called the modeling view, and usually a validation set, too. The training set, in the form of a table, is the required data input to the `creditscorecard` object, and this training set table must be prepared before creating a `creditscorecard` object in the Modeling phase.

2 Modeling phase

Use the `creditscorecard` object and associated object functions to develop a credit scorecard model. You can bin the data, apply the Weight of Evidence (WOE) transformation, and compute other statistics, such as the Information Value. You can fit a logistic regression model and also review the resulting scorecard points and format their scaling and rounding. For details on using the `creditscorecard` object, see [Using creditscorecard Objects](#).

3 Deployment phase

Deployment entails integrating a credit scorecard model into an IT production environment and keeping tracking logs, performance reports, and so on.

The `creditscorecard` object is mainly designed for the Modeling phase of the credit scorecard workflow. Support for all three phases requires other MathWorks® products.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

Related Examples

- “Troubleshooting Credit Scorecard Results”
- “Case Study for a Credit Scorecard Analysis” on page 8-87

More About

- “Credit Scorecard Modeling Workflow” on page 8-78

Credit Scorecard Modeling Workflow

Create, model, and analyze credit scorecards as follows.

1 Create a `creditscorecard` object.

Create a `creditscorecard` object for credit scorecard analysis by specifying “training” data in table format. The training data, sometimes called the modeling view, is the result of multiple data preparation tasks (see “About Credit Scorecards” on page 8-73) that must be performed before creating a `creditscorecard` object.

You can use optional input arguments for `creditscorecard` to specify scorecard properties such as the response variable and the `GoodLabel`. Perform some initial data exploration as soon as the `creditscorecard` object is created, although data analysis is usually done in combination with data binning (see step 2). For more information and examples, see `creditscorecard` and step 1 in “Case Study for a Credit Scorecard Analysis” on page 8-87.

2 Bin the data.

Perform manual or automatic binning of the data loaded into the `creditscorecard` object.

A common starting point is to apply automatic binning to all or selected variables using `autobinning`, report using `bininfo`, and visualize bin information with respect to bin counts and statistics or association measures such as Weight of Evidence (WOE) using `plotbins`. The bins can be modified or fine-tuned either manually using `modifybins` or with a different automatic binning algorithm using `autobinning`. Bins that show a close-to-linear trend in the WOE are frequently desired in the credit scorecard context. For more information and examples, see `autobinning`, `modifybins`, `bininfo`, and `plotbins` and step 2 in “Case Study for a Credit Scorecard Analysis” on page 8-87.

3 Fit a logistic regression model.

Fit a logistic regression model to the WOE data from the `creditscorecard` object. The `fitmodel` function internally bins the training data, transforms it into WOE values, maps the response variable so that 'Good' is 1, and fits a linear logistic regression model.

By default, `fitmodel` uses a stepwise procedure to determine which predictors should be in the model, but optional input arguments can also be used, for example,

to fit a full model. For more information and examples, see `fitmodel` and step 3 in “Case Study for a Credit Scorecard Analysis” on page 8-87.

4 Review and format credit scorecard points.

After fitting the logistic model, use `displaypoints` to summarize the scorecard points. By default, the points are unscaled and come directly from the combination of Weight of Evidence (WOE) values and model coefficients.

The `formatpoints` function lets you control scaling and rounding of scorecard points. For more information and examples, see `displaypoints` and `formatpoints` and step 4 in “Case Study for a Credit Scorecard Analysis” on page 8-87.

5 Score the data.

The `score` function computes the scores for the training data.

An optional data input can also be passed to `score`, for example, validation data. The points per predictor for each customer are also provided as an optional output. For more information and examples, see `score` and step 5 in “Case Study for a Credit Scorecard Analysis” on page 8-87.

For an example of this workflow, see “Case Study for a Credit Scorecard Analysis”.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

Related Examples

- “Troubleshooting Credit Scorecard Results”
- “Case Study for a Credit Scorecard Analysis” on page 8-87

More About

- “About Credit Scorecards” on page 8-73

Troubleshooting Credit Scorecard Results

In this section...

“Predictor Name Is Unspecified and the Parser Returns an Error” on page 8-80

“Using bininfo or plotbins Before Binning” on page 8-80

“If Categorical Data Is Given as Numeric or Vice Versa” on page 8-83

“NaNs Returned When Scoring a “Test” Dataset ” on page 8-84

This topic shows some of the results when using credit scorecards that need troubleshooting. These examples cover the full range of the credit score card workflow. For details on the overall process of creating and developing credit scorecards, see “Credit Scorecard Modeling Workflow”.

Predictor Name Is Unspecified and the Parser Returns an Error

If you attempt to bin using `autobinning` or `modifybins` and omit the predictor's name, the parser returns an error.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID', 'GoodLabel', 0);
modifybins(sc, 'CutPoints', [20 30 50 65])
```

Error using `creditscorecard/modifybins` (line 72)
Expected a string for the parameter name, instead the input type was 'double'.

Solution: Use this syntax to specify the `PredictorName` when using `modifybins`.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID', 'GoodLabel', 0);
modifybins(sc, 'CustIncome', 'CutPoints', [20 30 50 65]);
```

Using bininfo or plotbins Before Binning

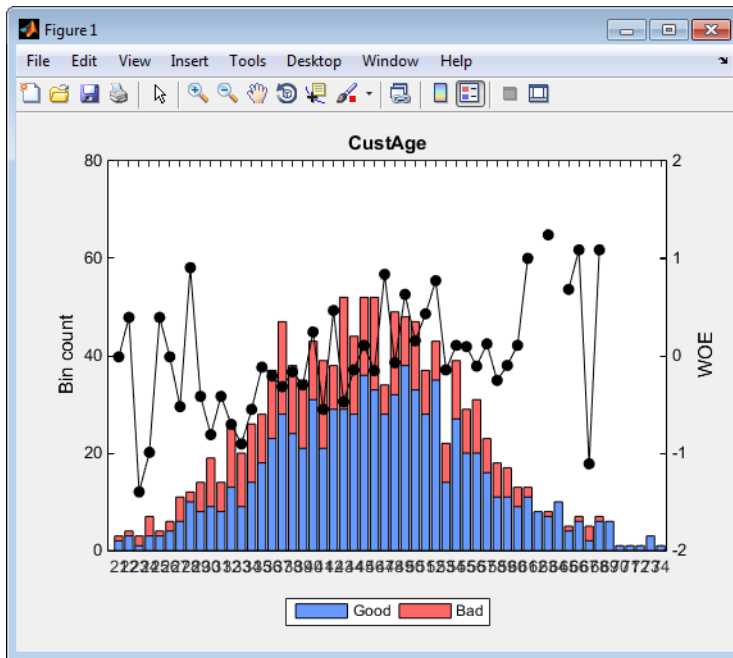
If you use `bininfo` or `plotbins` before binning, the results might be unusable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID', 'GoodLabel', 0);
bininfo(sc, 'CustAge')
```


plotbins(sc, 'CustAge')

ans =

Bin	Good	Bad	Odds	WOE	InfoValue
'21'	2	1	2	-0.011271	3.1821e-07
'22'	3	1	3	0.39419	0.00047977
'23'	1	2	0.5	-1.3976	0.0053002
'24'	3	4	0.75	-0.9921	0.0062895
'25'	3	1	3	0.39419	0.00047977
'26'	4	2	2	-0.011271	6.3641e-07
'27'	6	5	1.2	-0.5221	0.0026744
'28'	10	2	5	0.90502	0.0067112
'29'	8	6	1.3333	-0.41674	0.0021465
'30'	9	10	0.9	-0.80978	0.011321
'31'	8	6	1.3333	-0.41674	0.0021465
'32'	13	13	1	-0.70442	0.011663
'33'	9	11	0.81818	-0.90509	0.014934
'34'	14	12	1.1667	-0.55027	0.0070391
'35'	18	10	1.8	-0.11663	0.00032342
'36'	23	14	1.6429	-0.20798	0.0013772
'37'	28	19	1.4737	-0.31665	0.0041132
'38'	24	14	1.7143	-0.16542	0.0008894
'39'	21	14	1.5	-0.29895	0.0027242
'40'	31	12	2.5833	0.24466	0.0020499
'41'	21	18	1.1667	-0.55027	0.010559
'42'	29	9	3.2222	0.46565	0.0062605
'43'	29	23	1.2609	-0.47262	0.010312
'44'	28	16	1.75	-0.1448	0.00078672
'45'	36	16	2.25	0.10651	0.00048246
'46'	33	19	1.7368	-0.15235	0.0010303
'47'	28	6	4.6667	0.83603	0.016516
'48'	32	17	1.8824	-0.071896	0.00021357
'49'	38	10	3.8	0.63058	0.013957
'50'	33	14	2.3571	0.15303	0.00089239
'51'	28	9	3.1111	0.43056	0.0052525
'52'	35	8	4.375	0.77149	0.01808
'53'	14	8	1.75	-0.1448	0.00039336
'54'	27	12	2.25	0.10651	0.00036184
'55'	20	9	2.2222	0.094089	0.00021044
'56'	20	11	1.8182	-0.10658	0.00029856
'57'	16	7	2.2857	0.12226	0.00028035
'58'	11	7	1.5714	-0.25243	0.00099297
'59'	11	6	1.8333	-0.098283	0.00013904
'60'	9	4	2.25	0.10651	0.00012061
'61'	11	2	5.5	1.0003	0.0086637
'62'	8	0	Inf	Inf	Inf
'63'	7	1	7	1.2415	0.0076953
'64'	10	0	Inf	Inf	Inf
'65'	4	1	4	0.68188	0.0016791
'66'	6	1	6	1.0873	0.0053857
'67'	2	3	0.66667	-1.1099	0.0056227
'68'	6	1	6	1.0873	0.0053857
'69'	6	0	Inf	Inf	Inf
'70'	1	0	Inf	Inf	Inf
'71'	1	0	Inf	Inf	Inf
'72'	1	0	Inf	Inf	Inf
'73'	3	0	Inf	Inf	Inf
'74'	1	0	Inf	Inf	Inf
'Totals'	803	397	2.0227	NaN	Inf



The plot for `CustAge` is not readable because it has too many bins. Additionally, `bininfo` returns data that have `Inf` values for the WOE due to zero observations for either **Good** or **Bad**.

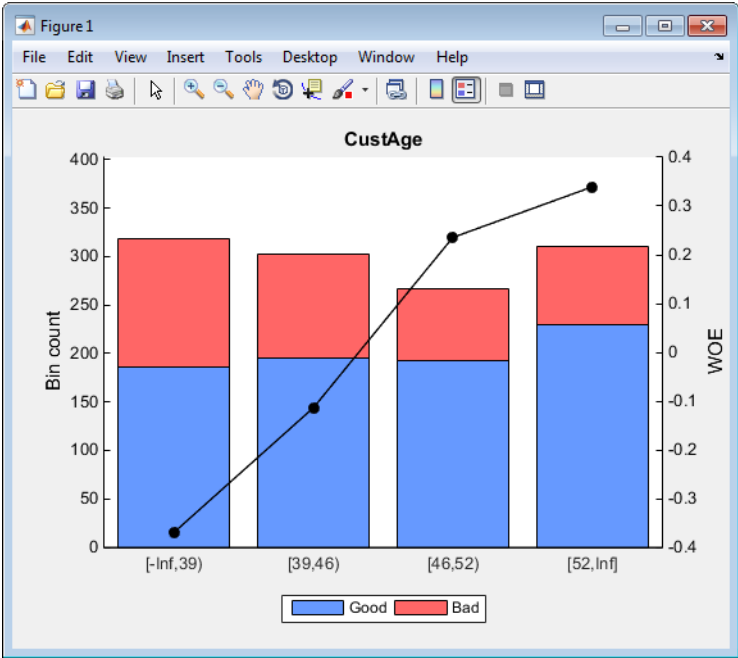
Solution: Bin the data using `autobinning` or `modifybins` before plotting or inquiring about the bin statistics, to avoid having too many bins or having NaNs and Infs. For example, you can use the name-value pair argument for `AlgoOptions` with the `autobinning` function to define the number of bins.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
AlgoOptions = {'NumBins',4};
sc = autobinning(sc,'CustAge','Algorithm','EqualFrequency',...
'AlgorithmOptions',AlgoOptions);
bininfo(sc,'CustAge','Totals','off')
plotbins(sc,'CustAge')

ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,39)'	186	133	1.3985	-0.36902	0.03815
'[39,46)'	195	108	1.8056	-0.11355	0.0033158
'[46,52)'	192	75	2.56	0.23559	0.011823

' [52,Inf] ' 230 81 2.8395 0.33921 0.02795



If Categorical Data Is Given as Numeric or Vice Versa

To show the case where numeric data is given as categorical data, the data for the variable `CustAge` is converted to categorical values. Then, if you use `modifybins` for the predictor variable `CustAge` with the name-value argument for `CutPoints`, an error is returned.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
data2 = data;
data2.CustAge = categorical(data2.CustAge);
sc2 = creditscorecard(data2, 'IDVar', 'CustID');
sc2 = modifybins(sc2, 'CustAge', 'CutPoints', [27 38 49 65]);
```

Error using creditscorecard/ValidateDataType (line 578)
The data type in the predictor and the cutpoints or category groupings are not compatible. Use cutpoints for numeric predictors and a table of category groupings for categorical predictors.

Error in creditscorecard/modifybins (line 95)
obj.ValidateDataType(obj.BinContainers{Ind},parser.Results);

Solution: The predictor `CustAge` is categorical data and not numeric data. When using `modifybins`, the name-value pair argument for categorical data is `CatGrouping` and not `CutPoints`.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
data2 = data;
data2.CustAge = categorical(data2.CustAge);
sc2 = creditscorecard(data2, 'IDVar', 'CustID');
[~,cg] = bininfo(sc2, 'CustAge');
sc2 = modifybins(sc2, 'CustAge', 'CatGrouping', cg);
```

Categorical does not mean categories, as in numbered categories, but rather MATLAB categorical-type data. For more information on categorical data, see `categorical`.

NaNs Returned When Scoring a “Test” Dataset

When applying a `creditscorecard` model to a “test” dataset using the `score` function, if an observation in the “test” dataset has a NaN or `<undefined>` value, a NaN total score is returned for each of these observations. For example, a `creditscorecard` object is created using “training” data.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
sc = autobinning(sc);
sc = fitmodel(sc);
```

Suppose that a missing observation (NaN) is added to the data and then `newdata` is scored using the `score` function. By default, the points and score assigned to the missing value is NaN.

```
newdata = data(1:10, :);
newdata.CustAge(1) = NaN;
[Scores, Points] = score(sc, newdata)
```

```
Scores =
     NaN
     1.3630
     0.9314
     1.5300
     1.2919
     1.9908
    -0.2524
     1.1070
     1.0299
     0.9386
```

```
Points =
```

AMBalance	CustAge	CustIncome	EmpStatus	OtherCC	ResStatus	TmWBank
-0.0053831	NaN	0.47018	-0.077573	0.16128	-0.031688	0.63608
-0.0053831	0.2851	0.47018	0.3163	0.16128	0.1272	0.0083609
-0.0053831	0.071887	0.14951	0.3163	-0.20527	-0.031688	0.63608
0.22621	0.27308	0.47018	0.3163	0.16128	0.1272	-0.044231
-0.023939	0.2851	0.47018	0.3163	0.16128	0.1272	-0.044231
-0.0053831	0.2851	0.47018	0.3163	0.16128	0.1272	0.63608
-0.023939	-0.13732	-0.31036	-0.077573	0.16128	0.1272	0.0083609
-0.1334	0.27308	0.47018	0.3163	-0.20527	0.37773	0.0083609
0.22621	0.27308	0.47018	-0.077573	0.16128	-0.031688	0.0083609
-0.023939	0.27308	0.47018	-0.077573	0.16128	0.1272	0.0083609

Additionally, notice that because the `CustAge` predictor for the first observation is `NaN`, the corresponding `Scores` output is `NaN` also.

Solution: Decide to assign points or not to cases where there is a `NaN` or `<undefined>` value for a specific record. To resolve this issue, use the `formatpoints` function with the name-value pair argument `Missing`. When using `Missing`, you can replace a predictor's `NaN` value with `Zero (0)`, `MinPoints`, or `MaxPoints`. By replacing `NaNs` with `MinPoints` or `MaxPoints`, either the minimum or maximum points for the predictor with the missing value or values are used.

For example, use `Missing` to replace the missing value with the minimum points for the predictor `CustAge` that has the missing value.

```
sc = formatpoints(sc, 'Missing', 'MinPoints');
[ Scores, Points ] = score(sc, newdata)
```

```
Scores =
```

1.0156
1.3630
0.9314
1.5300
1.2919
1.9908
-0.2524
1.1070
1.0299
0.9386

```
Points =
```

AMBalance	CustAge	CustIncome	EmpStatus	OtherCC	ResStatus	TmWBank
-0.0053831	-0.13732	0.47018	-0.077573	0.16128	-0.031688	0.63608
-0.0053831	0.2851	0.47018	0.3163	0.16128	0.1272	0.0083609

-0.0053831	0.071887	0.14951	0.3163	-0.20527	-0.031688	0.63608
0.22621	0.27308	0.47018	0.3163	0.16128	0.1272	-0.044231
-0.023939	0.2851	0.47018	0.3163	0.16128	0.1272	-0.044231
-0.0053831	0.2851	0.47018	0.3163	0.16128	0.1272	0.63608
-0.023939	-0.13732	-0.31036	-0.077573	0.16128	0.1272	0.0083609
-0.1334	0.27308	0.47018	0.3163	-0.20527	0.37773	0.0083609
0.22621	0.27308	0.47018	-0.077573	0.16128	-0.031688	0.0083609
-0.023939	0.27308	0.47018	-0.077573	0.16128	0.1272	0.0083609

Notice that the `Scores` output has a value for the first customer record because `CustAge` now has a value and the score can be calculated for the first customer record.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

Related Examples

- “Case Study for a Credit Scorecard Analysis” on page 8-87

More About

- “About Credit Scorecards” on page 8-73
- “Credit Scorecard Modeling Workflow” on page 8-78

Case Study for a Credit Scorecard Analysis

This example shows how to create a `creditscorecard` object, bin data, display and plot binned data information, fit a logistic regression model, and obtain a score for the scorecard model. This example also shows how to compute probabilities of default and compute the Kolmogorov-Smirnov statistic.

Step 1. Create a `creditscorecard` object.

Use the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). By default, `'ResponseVar'` is set to the last column in the data (`'status'` in this example) and the `'GoodLabel'` to the response value with the highest count (0 in this example). The syntax for `creditscorecard` indicates that `'CustID'` is the `'IDVar'` to remove from the list of predictors.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

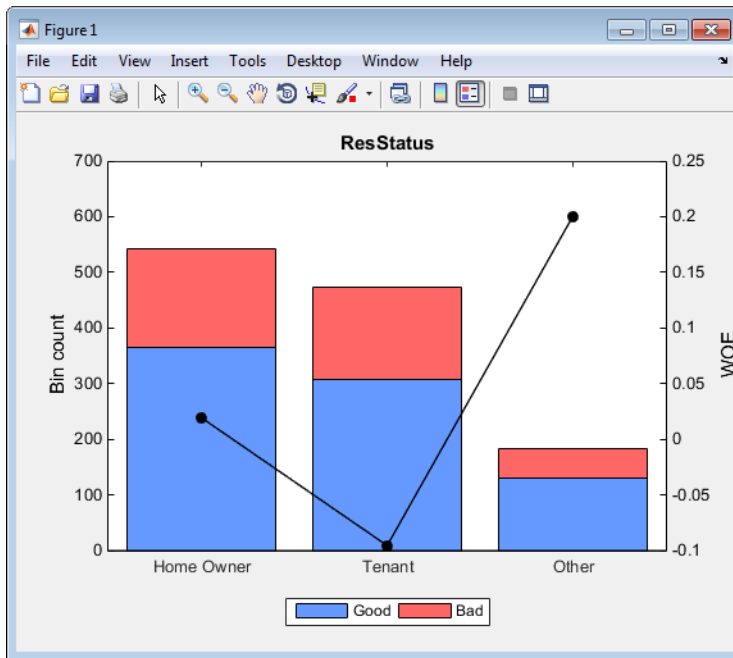
```
    IDVar: 'CustID'
  GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}
```

Perform some initial data exploration. Inquire about predictor statistics for the categorical variable `'ResStatus'` and plot the bin information for `'ResStatus'`.

```
bininfo(sc, 'ResStatus')
plotbins(sc, 'ResStatus')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682
'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738



This bin information contains the frequencies of “Good” and “Bad,” and bin statistics. It is recommended to avoid having bins with frequencies of zero, because they lead to infinite or undefined (NaN) statistics. Use the `modifybins` or `autobinning` functions to bin the data accordingly.

For numeric data, a common first step is "fine classing." This means binning the data into several bins, usually defined with a regular grid. To illustrate this point, use the predictor `'CustIncome'`.

```
cp = 20000:5000:60000;
```

```
sc = modifybins(sc, 'CustIncome', 'CutPoints', cp);
```

```
bininfo(sc, 'CustIncome')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
' [-Inf,20000) '	3	5	0.6	-1.2152	0.010765
' [20000,25000) '	23	16	1.4375	-0.34151	0.0039819

'[25000,30000]'	38	47	0.80851	-0.91698	0.065166
'[30000,35000]'	131	75	1.7467	-0.14671	0.003782
'[35000,40000]'	193	98	1.9694	-0.026696	0.00017359
'[40000,45000]'	173	76	2.2763	0.11814	0.0028361
'[45000,50000]'	131	47	2.7872	0.32063	0.014348
'[50000,55000]'	82	24	3.4167	0.52425	0.021842
'[55000,60000]'	21	8	2.625	0.26066	0.0015642
'[60000,Inf]'	8	1	8	1.375	0.010235
'Totals'	803	397	2.0227	NaN	0.13469

Step 2. Bin the data.

Use the `autobinning` function to perform automatic binning for every predictor variable, using default algorithm options.

```
sc = autobinning(sc);
```

After the automatic binning step, every predictor bin needs to be reviewed using the `bininfo` and `plotbins` functions and fine tuned. A monotonic, ideally linear trend in the Weight of Evidence (WOE) can be visualized using `plotbins`, which is desirable for credit scorecards because this translates into linear points for a given predictor. You can manually adjust the bins using the `modifybins` function. To demonstrate this, the bin for the predictor 'CustIncome' is adjusted. To begin, you can display and plot the bin information for 'CustIncome'.

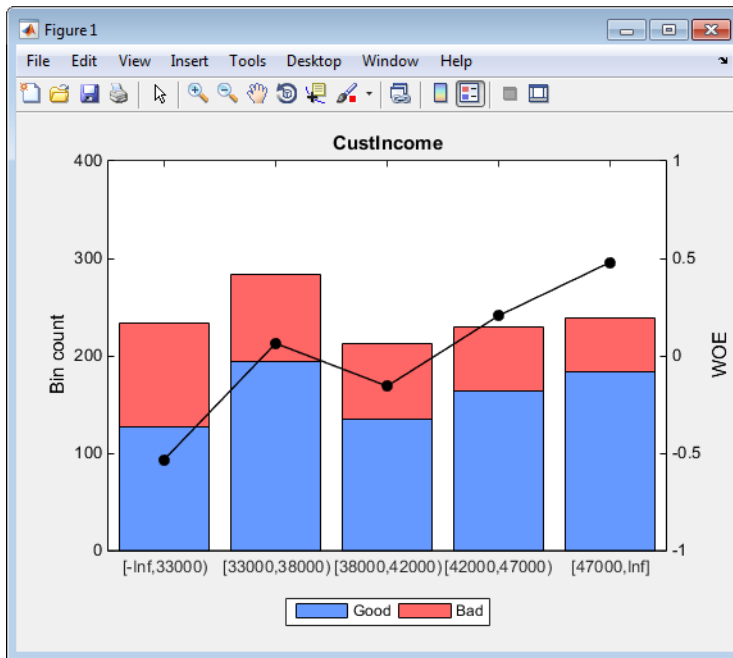
```
[bi,cp] = bininfo(sc, 'CustIncome')
plotbins(sc, 'CustIncome')
```

bi =

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,33000]'	127	107	1.1869	-0.53306	0.059364
'[33000,38000]'	194	90	2.1556	0.06363	0.00094769
'[38000,42000]'	135	78	1.7308	-0.15585	0.004419
'[42000,47000]'	164	66	2.4848	0.20579	0.0078175
'[47000,Inf]'	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.11421

cp =

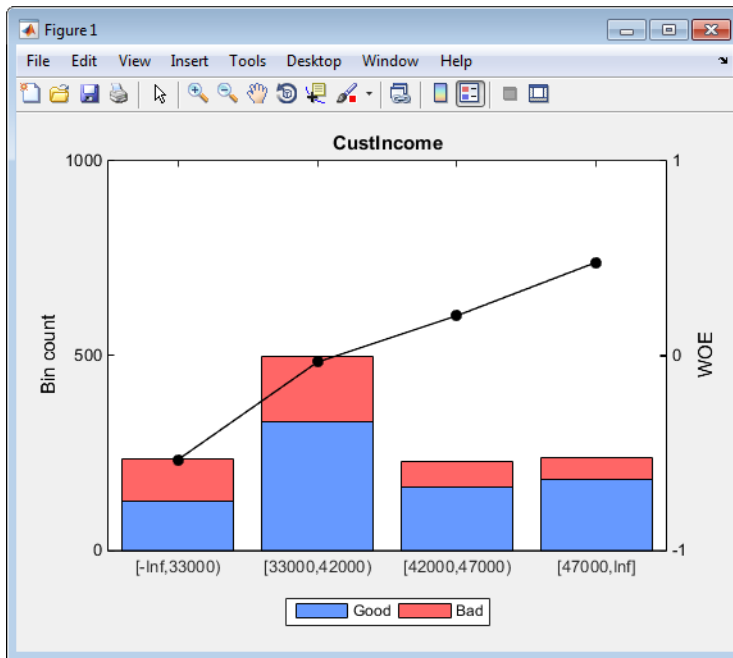
```
33000
38000
42000
47000
```



The WOE does not have a linear trend in this case.

To correct for the nonlinear trend for 'CustIncome', you can manually merge the second and third bins and then plot the bins again.

```
cp(2) = [];  
sc = modifybins(sc, 'CustIncome', 'CutPoints', cp);  
plotbins(sc, 'CustIncome')
```



This new binning shows a trend that looks much more linear than before. With similar adjustments, the bins for the other predictors can be modified to yield more linear WOE trends.

For brevity, this case study does not show the whole process for the remaining predictors. Cut points are set that yield close to linear WOE trends. Using `modifybins`, the cut points are set for the numeric predictors.

```
sc = modifybins(sc, 'CustAge', 'CutPoints', [33 48 58]);
sc = modifybins(sc, 'TmAtAddress', 'CutPoints', [30 54]);
sc = modifybins(sc, 'TmWBank', 'CutPoints', [24 48 64]);
sc = modifybins(sc, 'AMBalance', 'CutPoints', [760 1519 2287]);
sc = modifybins(sc, 'UtilRate', 'CutPoints', [0.09 0.14 0.32]);
```

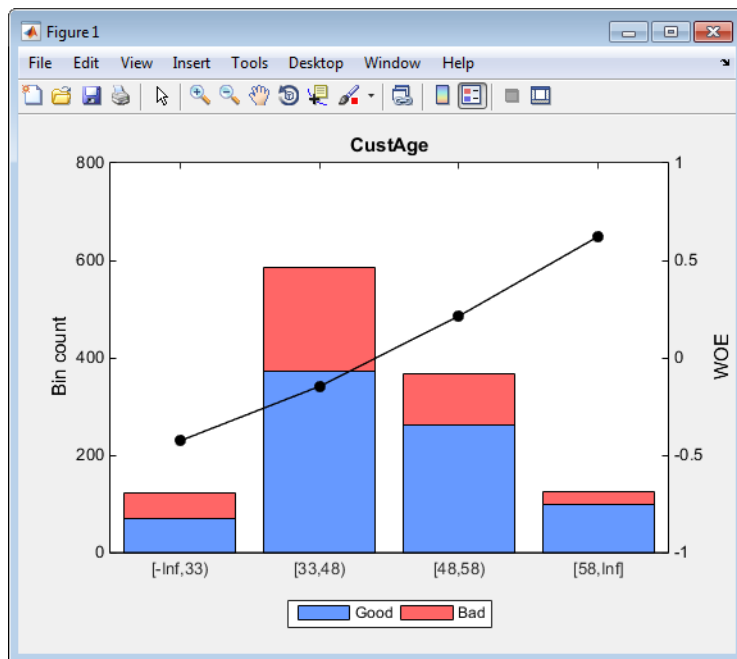
Similarly, categorical predictors can also be modified to group the categories. The categorical predictors in this example's data only have two or three categories. The automatic binning algorithm tries to bin categorical data. That is, automatic binning tries to perform cardinality reduction on it, or group the original categories. In this example, the categorical data remain unchanged because the binning algorithm by

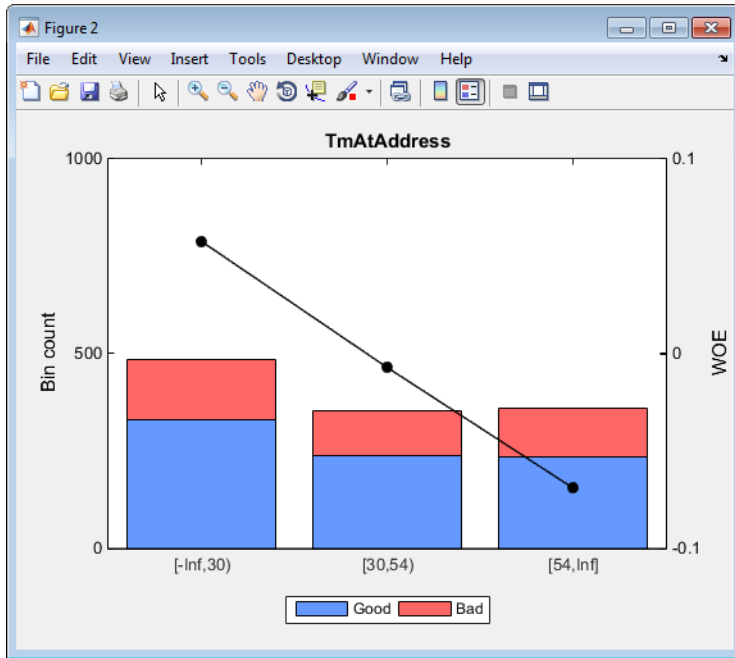
default tries to make five bins, yet the number of categories in the data is less than five. To manually group the categorical data, modify the 'BinNumber' field in the "category grouping" table that `bininfo` returns as an optional output. Here is an example to do this for 'ResStatus'.

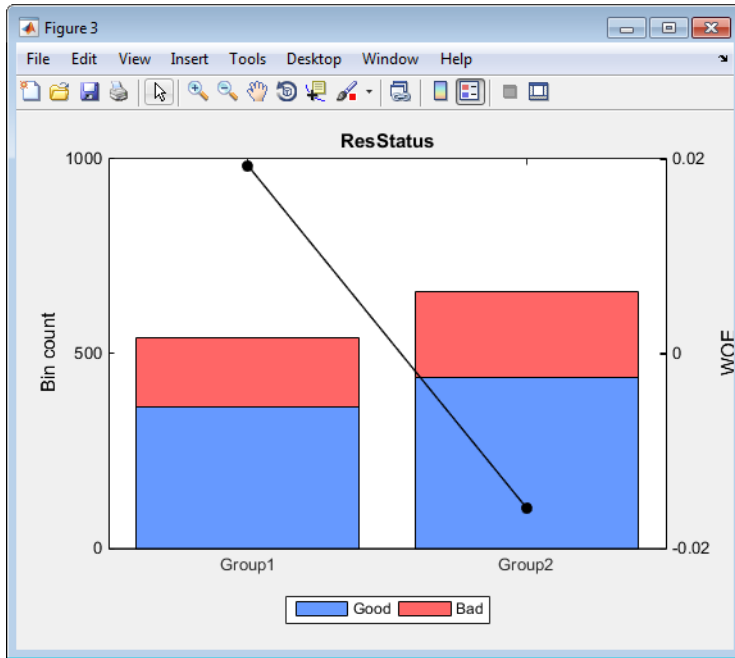
```
[~,cg] = bininfo(sc, 'ResStatus');  
cg.BinNumber(3) = 2;  
sc = modifybins(sc, 'ResStatus', 'CatGrouping', cg);
```

Plot bins for all predictors to verify close-to-linear WOE trends.

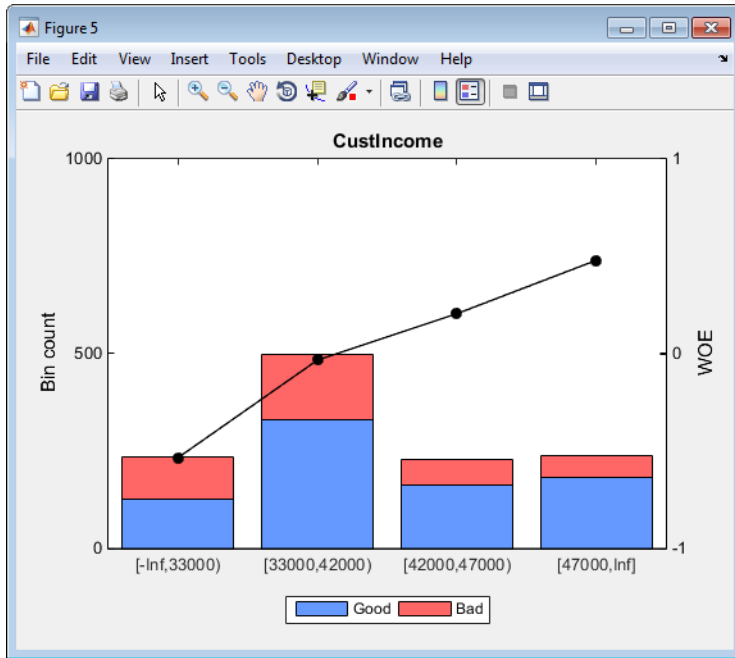
```
plotbins(sc, sc.PredictorVars)
```

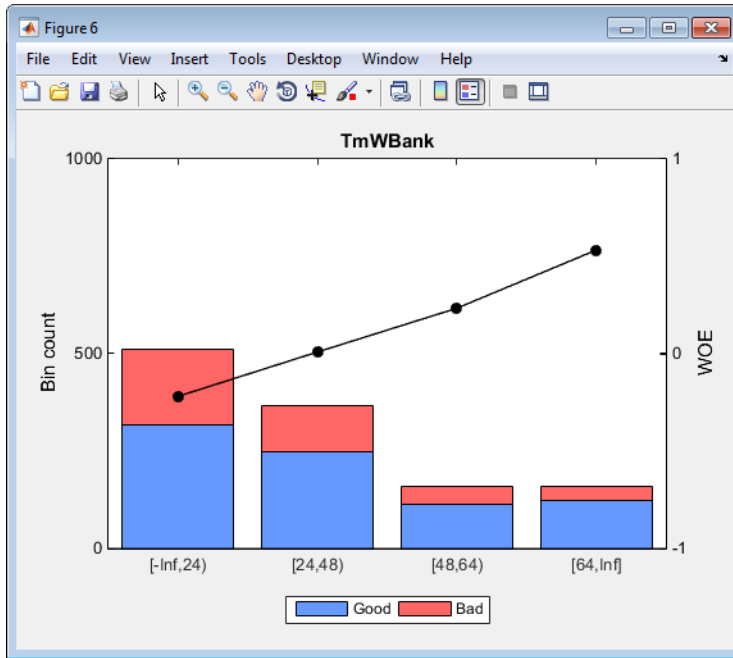


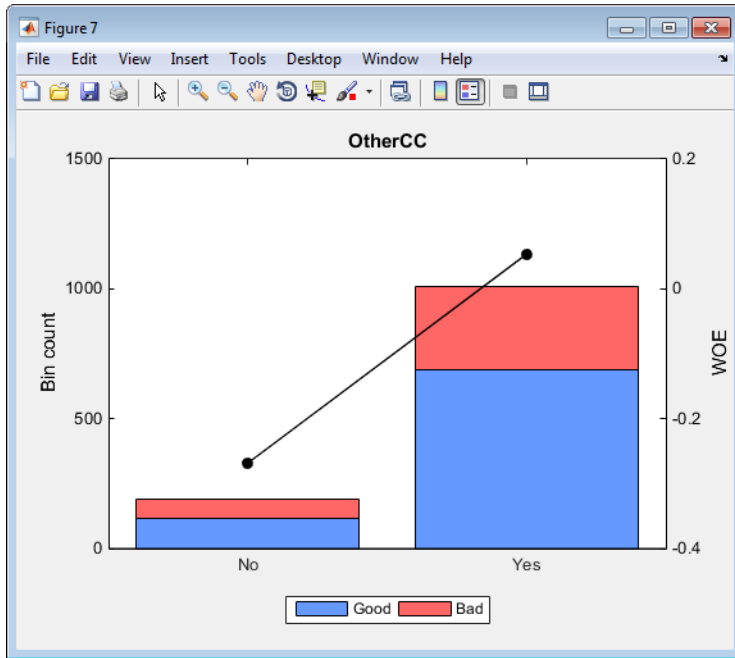


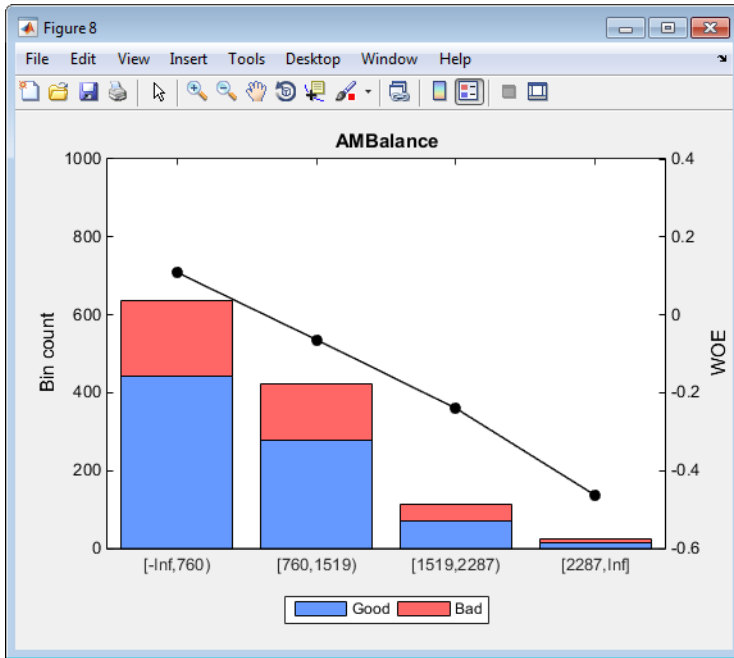


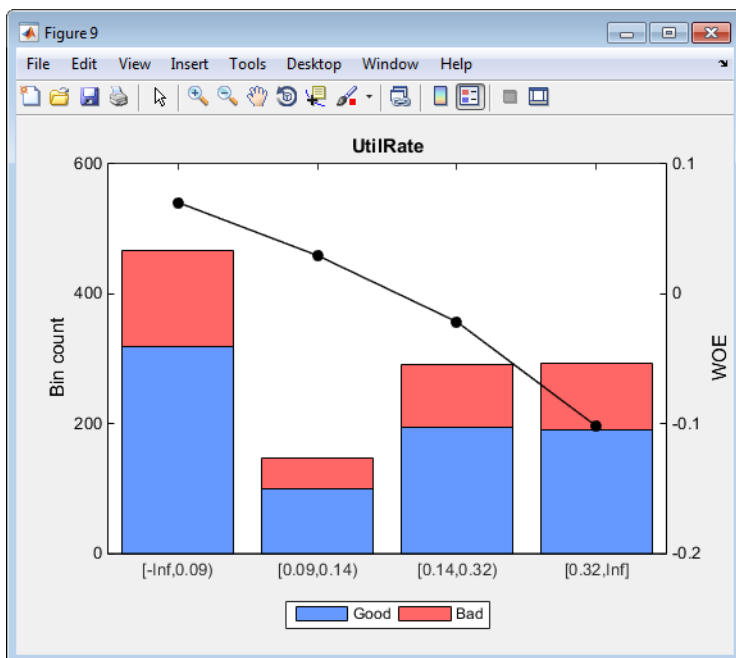












Step 3. Fit a logistic regression model.

The `fitmodel` function fits a logistic regression model to the WOE data. `fitmodel` internally bins the training data, transforms it into WOE values, maps the response variable so that 'Good' is 1, and fits a linear logistic regression model. By default, `fitmodel` uses a stepwise procedure to determine which predictors should be in the model.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1494.5261, Chi2Stat = 28.915224, PValue = 7.5616265e-08
2. Adding TmWBank, Deviance = 1476.8645, Chi2Stat = 17.661618, PValue = 2.6389838e-05
3. Adding EmpStatus, Deviance = 1466.4486, Chi2Stat = 10.415936, PValue = 0.0012493251
4. Adding CustAge, Deviance = 1461.8578, Chi2Stat = 4.5907441, PValue = 0.032145056
5. Adding OtherCC, Deviance = 1457.4045, Chi2Stat = 4.4533256, PValue = 0.03483335

Generalized Linear regression model:

```
logit(status) ~ 1 + CustAge + EmpStatus + CustIncome + TmWBank + OtherCC
Distribution = Binomial
```

Estimated Coefficients:

Estimate	SE	tStat	pValue
_____	_____	_____	_____

(Intercept)	0.70318	0.063284	11.112	1.1024e-28
CustAge	0.60958	0.27086	2.2505	0.024416
EmpStatus	0.96968	0.28965	3.3478	0.00081467
CustIncome	0.73252	0.22779	3.2157	0.0013013
TmWBank	1.0371	0.26022	3.9853	6.7389e-05
OtherCC	1.1139	0.52369	2.1271	0.033416

1200 observations, 1194 error degrees of freedom
 Dispersion: 1
 Chi²-statistic vs. constant model: 66, p-value = 6.83e-13

Step 4. Review and format scorecard points.

After fitting the logistic model, by default the points are unscaled and come directly from the combination of WOE values and model coefficients. The `displaypoints` function summarizes the scorecard points.

```
p1 = displaypoints(sc);
disp(p1)
```

Predictors	Bin	Points
'CustAge'	'[-Inf,33)'	-0.11918
'CustAge'	'[33,48)'	0.05114
'CustAge'	'[48,58)'	0.27095
'CustAge'	'[58,Inf]'	0.52007
'EmpStatus'	'Employed'	0.3751
'EmpStatus'	'Unknown'	-0.052783
'CustIncome'	'[-Inf,33000)'	-0.24984
'CustIncome'	'[33000,42000)'	0.11696
'CustIncome'	'[42000,47000)'	0.29138
'CustIncome'	'[47000,Inf]'	0.49204
'TmWBank'	'[-Inf,24)'	-0.085978
'TmWBank'	'[24,48)'	0.15016
'TmWBank'	'[48,64)'	0.38315
'TmWBank'	'[64,Inf]'	0.69271
'OtherCC'	'No'	-0.15824
'OtherCC'	'Yes'	0.20008

This is a good time to modify the bin labels, if this is something of interest for cosmetic reasons. To do so, use `modifybins` to change the bin labels.

```
sc = modifybins(sc,'CustAge','BinLabels',...
{'Up to 32' '33 to 47' '48 to 57' '58 and up'});
sc = modifybins(sc,'CustIncome','BinLabels',...
```

```
{'Up to 32999' '33000 to 41999' '42000 to 46999' '47000 and up'});
sc = modifybins(sc,'TmWBank','BinLabels',...
{'Up to 23' '24 to 47' '48 to 63' '64 and up'});
p1 = displaypoints(sc);
disp(p1)
```

Predictors	Bin	Points
'CustAge'	'Up to 32'	-0.11918
'CustAge'	'33 to 47'	0.05114
'CustAge'	'48 to 57'	0.27095
'CustAge'	'58 and up'	0.52007
'EmpStatus'	'Employed'	0.3751
'EmpStatus'	'Unknown'	-0.052783
'CustIncome'	'Up to 32999'	-0.24984
'CustIncome'	'33000 to 41999'	0.11696
'CustIncome'	'42000 to 46999'	0.29138
'CustIncome'	'47000 and up'	0.49204
'TmWBank'	'Up to 23'	-0.085978
'TmWBank'	'24 to 47'	0.15016
'TmWBank'	'48 to 63'	0.38315
'TmWBank'	'64 and up'	0.69271
'OtherCC'	'No'	-0.15824
'OtherCC'	'Yes'	0.20008

Points are usually scaled and also often rounded. To do this, use the `formatpoints` function. For example, you can set a target level of points corresponding to a target odds level and also set the required points-to-double-the-odds (PDO).

```
TargetPoints = 500;
TargetOdds = 2;
PDO = 50; % Points to double the odds

sc = formatpoints(sc,'PointsOddsAndPDO',[TargetPoints TargetOdds PDO],'Round','AllPoints');
p2 = displaypoints(sc);
disp(p2)
```

Predictors	Bin	Points
'CustAge'	'Up to 32'	81
'CustAge'	'33 to 47'	94
'CustAge'	'48 to 57'	110
'CustAge'	'58 and up'	128
'EmpStatus'	'Employed'	117
'EmpStatus'	'Unknown'	86

'CustIncome'	'Up to 32999'	72
'CustIncome'	'33000 to 41999'	98
'CustIncome'	'42000 to 46999'	111
'CustIncome'	'47000 and up'	125
'TmWBank'	'Up to 23'	84
'TmWBank'	'24 to 47'	101
'TmWBank'	'48 to 63'	118
'TmWBank'	'64 and up'	140
'OtherCC'	'No'	79
'OtherCC'	'Yes'	104

Step 5. Score the data.

The `score` function computes the scores for the training data. An optional `data` input can also be passed to `score`, for example, validation data. The points per predictor for each customer are provided as an optional output.

```
[Scores,Points] = score(sc);
disp(Scores(1:10))
disp(Points(1:10,:))
```

```
543
575
506
540
558
592
457
532
526
509
```

CustAge	CustIncome	EmpStatus	OtherCC	TmWBank
110	125	86	104	118
128	125	117	104	101
94	98	117	79	118
110	125	117	104	84
128	125	117	104	84
128	125	117	104	118
94	72	86	104	101
110	125	117	79	101
110	125	86	104	101
110	125	86	104	84

Appendix A. Calculate the probability of default.

To calculate the probability of default, you need to use unscaled points. If you scaled points previously, you can unscale them as shown here. Because 'Good' is mapped to '1' to fit the logistic model, the application of the logistic function yields probabilities

of being good. The probability of default is thus 1 minus that probability. Remember to restore previous formatting of scorecard points at the end of this process.

```
% Revert to unscaled scores
sc = formatpoints(sc, 'ShiftAndSlope', [0 1], 'Round', 'None');

% Get unscaled scores
UnscaledScores = score(sc);

% Apply logistic function to get probabilities of good (and bad)
ProbGood = 1./(1+exp(-UnscaledScores));
ProbDefault = 1 - ProbGood;

% Reset previous formatting of scorecard points
sc = formatpoints(sc, 'PointsOddsAndPDO', [TargetPoints TargetOdds PDO], 'Round', 'AllPoints');
```

Plot the formatted scores versus the predicted odds. Visually verify that the target points and target odds match and that the points-to-double-the-odds (PDO) relationship holds.

```
PredictedOdds = ProbGood./ProbDefault;

figure
scatter(Scores, PredictedOdds)
title('Score vs. Predicted Odds')
xlabel('Score')
ylabel('Predicted Odds')

hold on

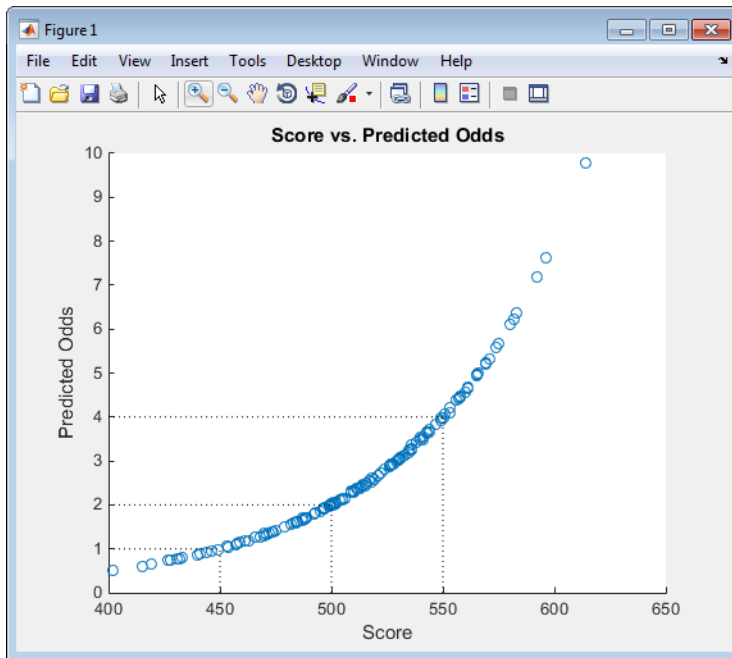
xLimits = xlim;
yLimits = ylim;

% Target points and odds
plot([TargetPoints TargetPoints], [yLimits(1) TargetOdds], 'k:')
plot([xLimits(1) TargetPoints], [TargetOdds TargetOdds], 'k:')

% Target points plus PDO
plot([TargetPoints+PDO TargetPoints+PDO], [yLimits(1) 2*TargetOdds], 'k:')
plot([xLimits(1) TargetPoints+PDO], [2*TargetOdds 2*TargetOdds], 'k:')

% Target points minus PDO
plot([TargetPoints-PDO TargetPoints-PDO], [yLimits(1) TargetOdds/2], 'k:')
plot([xLimits(1) TargetPoints-PDO], [TargetOdds/2 TargetOdds/2], 'k:')

hold off
```

Appendix B. Validate the credit scorecard model using the Kolmogorov-Smirnov statistic.

The Kolmogorov-Smirnov (K-S) plot, also known as the fish-eye graph, is a common statistic used to measure the predictive power of scorecards. The K-S plot illustrates the effectiveness of scorecards and measures the difference between the cumulative distributions of 'Goods' and 'Bads' in a sample. The statistic of interest is called the K-S statistic and is the largest difference between the cumulative distributions of 'Bads' and 'Goods'. The K-S statistic is commonly in the range 20-70%, outside of which the model is either questionable or "too good to be true." To get the cumulative distribution of 'Bads' and 'Goods':

- Append the response variable 'status' to the scores and sort by scores.
- Get cumulative counts for 'Goods' and 'Bads'. 'Bads' are 1s in the original data (sc.GoodLabel is 0).
- Convert from cumulative counts to frequencies (normalize by total 'Bads' and total 'Goods').

The K-S statistic is the maximum difference between the cumulative distributions. The score at which this maximum is reached is usually reported, too. The K-S computation is not affected if multiple individuals in the sample have the same score.

```
SortedScoresAndResp = sortrows([Scores data.status]);
SortedScores = SortedScoresAndResp(:,1);

NumObs = size(SortedScoresAndResp,1);

CumBad = cumsum(SortedScoresAndResp(:,2));
CumGood = (1:NumObs)' - CumBad;

CumBad = CumBad/CumBad(end);
CumGood = CumGood/CumGood(end);

[KSValue,KSInd] = max(CumBad-CumGood);
KSScore = SortedScores(KSInd);
fprintf('K-S value: %3.1f%%, at %g points\n',KSValue*100,KSScore)
```

K-S value: 22.2%, at 509 points

Plot the cumulative distributions and add lines to indicate the location of the score that yields the K-S statistic.

```
figure
plot(SortedScores,CumBad,SortedScores,CumGood);
legend('Cum Bad','Cum Good','location','SE')
title('K-S Plot')
xlabel('Score')
ylabel('Cum Prob')

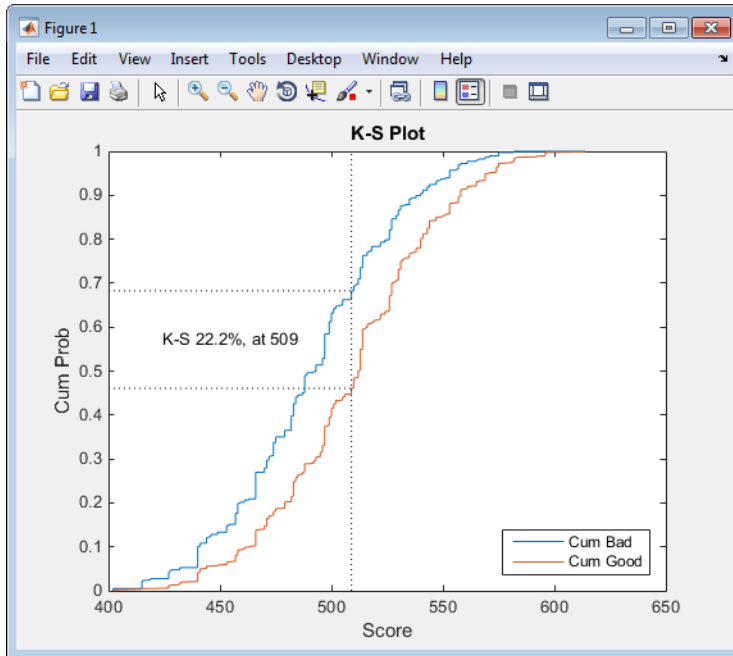
hold on

yLimits = ylim;
xLimits = xlim;

plot([KSScore KSScore],yLimits,'k:')
plot([xLimits(1) KSScore],[CumBad(KSInd) CumBad(KSInd)],'k:')
plot([xLimits(1) KSScore],[CumGood(KSInd) CumGood(KSInd)],'k:')

TextInPlot = sprintf('K-S %3.1f%%, at %g',KSValue*100,KSScore);
text((xLimits(1)+KSScore)/2,(CumGood(KSInd)+CumBad(KSInd))/2,TextInPlot,...
    'HorizontalAlignment','Center')

hold off
```



See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

Related Examples

- “Troubleshooting Credit Scorecard Results”

More About

- “About Credit Scorecards” on page 8-73
- “Credit Scorecard Modeling Workflow” on page 8-78

Regression with Missing Data

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-8
- “Multivariate Normal Regression Types” on page 9-15
- “Valuation with Missing Data” on page 9-30

Multivariate Normal Regression

In this section...

“Introduction” on page 9-2

“Multivariate Normal Linear Regression” on page 9-2

“Maximum Likelihood Estimation” on page 9-3

“Special Case of Multiple Linear Regression Model” on page 9-4

“Least-Squares Regression” on page 9-5

“Mean and Covariance Estimation” on page 9-5

“Convergence” on page 9-5

“Fisher Information” on page 9-6

“Statistical Tests” on page 9-7

Introduction

This section focuses on using likelihood-based methods for multivariate normal regression. The parameters of the regression model are estimated via maximum likelihood estimation. For multiple series, this requires iteration until convergence. The complication due to the possibility of missing data is incorporated into the analysis with a variant of the EM algorithm known as the ECM algorithm.

The underlying theory of maximum likelihood estimation and the definition and significance of the Fisher information matrix can be found in Caines [1] and Cramér [2]. The underlying theory of the ECM algorithm can be found in Meng and Rubin [8] and Sexton and Swensen [9].

In addition, these two examples of maximum likelihood estimation are presented:

- “Portfolios with Missing Data” on page 9-24
- “Estimation of Some Technology Stock Betas” on page 9-32

Multivariate Normal Linear Regression

Suppose you have a multivariate normal linear regression model in the form

$$\begin{bmatrix} \mathbf{Z}_1 \\ \vdots \\ \mathbf{Z}_m \end{bmatrix} \sim N \left(\begin{bmatrix} \mathbf{H}_1 \mathbf{b} \\ \vdots \\ \mathbf{H}_m \mathbf{b} \end{bmatrix}, \begin{bmatrix} \mathbf{C} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \mathbf{C} \end{bmatrix} \right),$$

where the model has m observations of n -dimensional random variables $\mathbf{Z}_1, \dots, \mathbf{Z}_m$ with a linear regression model that has a p -dimensional model parameter vector \mathbf{b} . In addition, the model has a sequence of m design matrices $\mathbf{H}_1, \dots, \mathbf{H}_m$, where each design matrix is a known n -by- p matrix.

Given a parameter vector \mathbf{b} and a collection of design matrices, the collection of m independent variables \mathbf{Z}_k is assumed to have independent identically distributed multivariate normal residual errors $\mathbf{Z}_k - \mathbf{H}_k \mathbf{b}$ with n -vector mean $\mathbf{0}$ and n -by- n covariance matrix \mathbf{C} for each $k = 1, \dots, m$.

A concise way to write this model is

$$\mathbf{Z}_k \sim N(\mathbf{H}_k \mathbf{b}, \mathbf{C})$$

for $k = 1, \dots, m$.

The goal of multivariate normal regression is to obtain maximum likelihood estimates for \mathbf{b} and \mathbf{C} given a collection of m observations $\mathbf{z}_1, \dots, \mathbf{z}_m$ of the random variables $\mathbf{Z}_1, \dots, \mathbf{Z}_m$. The estimated parameters are the p distinct elements of \mathbf{b} and the $n(n+1)/2$ distinct elements of \mathbf{C} (the lower-triangular elements of \mathbf{C}).

Note: Quasi-maximum likelihood estimation works with the same models but with a relaxation of the assumption of normally distributed residuals. In this case, however, the parameter estimates are asymptotically optimal.

Maximum Likelihood Estimation

To estimate the parameters of the multivariate normal linear regression model using maximum likelihood estimation, it is necessary to maximize the log-likelihood function over the estimation parameters given observations $\mathbf{z}_1, \dots, \mathbf{z}_m$.

Given the multivariate normal model to characterize residual errors in the regression model, the log-likelihood function is

$$L(z_1, \dots, z_m; \mathbf{b}, \mathbf{C}) = \frac{1}{2} mn \log(2\pi) + \frac{1}{2} m \log(\det(\mathbf{C})) \\ + \frac{1}{2} \sum_{k=1}^m (z_k - H_k \mathbf{b})^T \mathbf{C}^{-1} (z_k - H_k \mathbf{b}).$$

Although the cross-sectional residuals must be independent, you can use this log-likelihood function for quasi-maximum likelihood estimation. In this case, the estimates for the parameters \mathbf{b} and \mathbf{C} provide estimates to characterize the first and second moments of the residuals. See Caines [1] for details.

Except for a special case (see “Special Case of Multiple Linear Regression Model” on page 9-4), if both the model parameters in \mathbf{b} and the covariance parameters in \mathbf{C} are to be estimated, the estimation problem is intractably nonlinear and a solution must use iterative methods. Denote estimates for the parameters \mathbf{b} and \mathbf{C} for iteration $t = 0, 1, \dots$ with the superscript notation $\mathbf{b}^{(t)}$ and $\mathbf{C}^{(t)}$.

Given initial estimates $\mathbf{b}^{(0)}$ and $\mathbf{C}^{(0)}$ for the parameters, the maximum likelihood estimates for \mathbf{b} and \mathbf{C} are obtained using a two-stage iterative process with

$$\mathbf{b}^{(t+1)} = \left(\sum_{k=1}^m H_k^T (\mathbf{C}^{(t)})^{-1} H_k \right)^{-1} \left(\sum_{k=1}^m H_k^T (\mathbf{C}^{(t)})^{-1} z_k \right)$$

and

$$\mathbf{C}^{(t+1)} = \frac{1}{m} \sum_{k=1}^m (z_k - H_k \mathbf{b}^{(t+1)}) (z_k - H_k \mathbf{b}^{(t+1)})^T$$

for $t = 0, 1, \dots$.

Special Case of Multiple Linear Regression Model

The special case mentioned in “Maximum Likelihood Estimation” on page 9-3 occurs if $n = 1$ so that the sequence of observations is a sequence of scalar observations. This model is known as a multiple linear regression model. In this case, the covariance matrix \mathbf{C} is a 1-by-1 matrix that drops out of the maximum likelihood iterates so that a single-step estimate for \mathbf{b} and \mathbf{C} can be obtained with converged estimates $\mathbf{b}^{(1)}$ and $\mathbf{C}^{(1)}$.

Least-Squares Regression

Another simplification of the general model is called least-squares regression. If $\mathbf{b}^{(0)} = \mathbf{0}$ and $\mathbf{C}^{(0)} = \mathbf{I}$, then $\mathbf{b}^{(1)}$ and $\mathbf{C}^{(1)}$ from the two-stage iterative process are least-squares estimates for \mathbf{b} and \mathbf{C} , where

$$b^{LS} = \left(\sum_{k=1}^m H_k^T H_k \right)^{-1} \left(\sum_{k=1}^m H_k^T z_k \right)$$

and

$$\mathbf{C}^{LS} = \frac{1}{m} \sum_{k=1}^m (z_k - H_k b^{LS})(z_k - H_k b^{LS})^T.$$

Mean and Covariance Estimation

A final simplification of the general model is to estimate the mean and covariance of a sequence of n -dimensional observations $\mathbf{z}_1, \dots, \mathbf{z}_m$. In this case, the number of series is equal to the number of model parameters with $n = p$ and the design matrices are identity matrices with $\mathbf{H}_k = \mathbf{I}$ for $i = 1, \dots, m$ so that \mathbf{b} is an estimate for the mean and \mathbf{C} is an estimate of the covariance of the collection of observations $\mathbf{z}_1, \dots, \mathbf{z}_m$.

Convergence

If the iterative process continues until the log-likelihood function increases by no more than a specified amount, the resultant estimates are said to be maximum likelihood estimates \mathbf{b}^{ML} and \mathbf{C}^{ML} .

Note that if $n = 1$ (which implies a single data series), convergence occurs after only one iterative step, which, in turn, implies that the least-squares and maximum likelihood estimates are identical. If, however, $n > 1$, the least-squares and maximum likelihood estimates are usually distinct.

In Financial Toolbox software, both the changes in the log-likelihood function and the norm of the change in parameter estimates are monitored. Whenever both changes fall

below specified tolerances (which should be something between machine precision and its square root), the toolbox functions terminate under an assumption that convergence has been achieved.

Fisher Information

Since maximum likelihood estimates are formed from samples of random variables, their estimators are random variables; an estimate derived from such samples has an uncertainty associated with it. To characterize these uncertainties, which are called standard errors, two quantities are derived from the total log-likelihood function.

The Hessian of the total log-likelihood function is

$$\nabla^2 L(z_1, \dots, z_m; \theta)$$

and the Fisher information matrix is

$$I(\theta) = -E\left[\nabla^2 L(z_1, \dots, z_m; \theta)\right],$$

where the partial derivatives of the ∇^2 operator are taken with respect to the combined parameter vector Θ that contains the distinct components of \mathbf{b} and \mathbf{C} with a total of $q = p + n(n + 1)/2$ parameters.

Since maximum likelihood estimation is concerned with large-sample estimates, the central limit theorem applies to the estimates and the Fisher information matrix plays a key role in the sampling distribution of the parameter estimates. Specifically, maximum likelihood parameter estimates are asymptotically normally distributed such that

$$(\theta^{(t)} - \theta) \sim N(0, I^{-1}, (\theta^{(t)})) \text{ as } t \rightarrow \infty,$$

where Θ is the combined parameter vector and $\Theta^{(t)}$ is the estimate for the combined parameter vector at iteration $t = 0, 1, \dots$.

The Fisher information matrix provides a lower bound, called a Cramér-Rao lower bound, for the standard errors of estimates of the model parameters.

Statistical Tests

Given an estimate for the combined parameter vector Θ , the squared standard errors are the diagonal elements of the inverse of the Fisher information matrix

$$s^2(\hat{\theta}_i) = (I^{-1}(\hat{\theta}_i))_{ii}$$

for $i = 1, \dots, q$.

Since the standard errors are estimates for the standard deviations of the parameter estimates, you can construct confidence intervals so that, for example, a 95% interval for each parameter estimate is approximately

$$\hat{\theta}_i \pm 1.96s(\hat{\theta}_i)$$

for $i = 1, \dots, q$.

Error ellipses at a level-of-significance $\alpha \in [0, 1]$ for the parameter estimates satisfy the inequality

$$(\theta - \hat{\theta})^T I(\hat{\theta})(\theta - \hat{\theta}) \leq \chi_{1-\alpha, q}^2$$

and follow a χ^2 distribution with q degrees-of-freedom. Note that similar inequalities can be formed for any subcollection of the parameters.

In general, given parameter estimates, the computed Fisher information matrix, and the log-likelihood function, you can perform numerous statistical tests on the parameters, the model, and the regression.

Maximum Likelihood Estimation with Missing Data

In this section...

“Introduction” on page 9-8

“ECM Algorithm” on page 9-9

“Standard Errors” on page 9-9

“Data Augmentation” on page 9-10

“Multivariate Normal Regression Functions” on page 9-11

“Multivariate Normal Regression Without Missing Data” on page 9-12

“Multivariate Normal Regression With Missing Data” on page 9-13

“Least-Squares Regression with Missing Data” on page 9-13

“Multivariate Normal Parameter Estimation with Missing Data” on page 9-14

“Support Functions” on page 9-14

Introduction

Suppose that a portion of the sample data is missing, where missing values are represented as NaNs. If the missing values are missing-at-random and ignorable, where Little and Rubin [7] have precise definitions for these terms, it is possible to use a version of the Expectation Maximization, or EM, algorithm of Dempster, Laird, and Rubin [3] to estimate the parameters of the multivariate normal regression model. The algorithm used in Financial Toolbox software is the ECM (Expectation Conditional Maximization) algorithm of Meng and Rubin [8] with enhancements by Sexton and Swensen [9].

Each sample \mathbf{z}_k for $k = 1, \dots, m$, is either complete with no missing values, empty with no observed values, or incomplete with both observed and missing values. Empty samples are ignored since they contribute no information.

To understand the missing-at-random and ignorability conditions, consider an example of stock price data before an IPO. For a counterexample, censored data, in which all values greater than some cutoff are replaced with NaNs, does not satisfy these conditions.

In sample k , let \mathbf{x}_k represent the missing values in \mathbf{z}_k , and \mathbf{y}_k represent the observed values. Define a permutation matrix \mathbf{P}_k so that

$$z_k = P_k \begin{bmatrix} x_k \\ y_k \end{bmatrix}$$

for $k = 1, \dots, m$.

ECM Algorithm

The ECM algorithm has two steps – an E, or expectation step, and a CM, or conditional maximization, step. As with maximum likelihood estimation, the parameter estimates evolve according to an iterative process, where estimates for the parameters after t iterations are denoted as $\mathbf{b}^{(t)}$ and $\mathbf{C}^{(t)}$.

The E step forms conditional expectations for the elements of missing data with

$$E \left[X_k | Y_k = y_k; \mathbf{b}^{(t)}, \mathbf{C}^{(t)} \right]$$

$$\text{cov} \left[X_k | Y_k = y_k; \mathbf{b}^{(t)}, \mathbf{C}^{(t)} \right]$$

for each sample $k \in \{1, \dots, m\}$ that has missing data.

The CM step proceeds in the same manner as the maximum likelihood procedure without missing data. The main difference is that missing data moments are imputed from the conditional expectations obtained in the E step.

The E and CM steps are repeated until the log-likelihood function ceases to increase. One of the important properties of the ECM algorithm is that it is always guaranteed to find a maximum of the log-likelihood function and, under suitable conditions, this maximum can be a global maximum.

Standard Errors

The negative of the expected Hessian of the log-likelihood function and the Fisher information matrix are identical if no data is missing. However, if data is missing, the Hessian, which is computed over available samples, accounts for the loss of information due to missing data. Consequently, the Fisher information matrix provides standard errors that are a Cramér-Rao lower bound whereas the Hessian matrix provides standard errors that may be greater if there is missing data.

Data Augmentation

The ECM functions do not “fill in” missing values as they estimate model parameters. In some cases, you may want to fill in the missing values. Although you can fill in the missing values in your data with conditional expectations, you would get optimistic and unrealistic estimates because conditional estimates are not random realizations.

Several approaches are possible, including resampling methods and multiple imputation (see Little and Rubin [7] and Shafer [10] for details). A somewhat informal sampling method for data augmentation is to form random samples for missing values based on the conditional distribution for the missing values. Given parameter estimates for $X \subset R^n$ and \hat{C} , each observation has moments

$$E[Z_k] = H_k \hat{b}$$

and

$$cov(Z_k) = H_k \hat{C} H_k^T$$

for $k = 1, \dots, m$, where you have dropped the parameter dependence on the left sides for notational convenience.

For observations with missing values partitioned into missing values \mathbf{X}_k and observed values $\mathbf{Y}_k = \mathbf{y}_k$, you can form conditional estimates for any subcollection of random variables within a given observation. Thus, given estimates $E[Z_k]$ and $cov(Z_k)$ based on the parameter estimates, you can create conditional estimates

$$E[X_k | y_k]$$

and

$$cov(X_k | y_k)$$

using standard multivariate normal distribution theory. Given these conditional estimates, you can simulate random samples for the missing values from the conditional distribution

$$X_k \sim N\left(E[X_k | y_k], \text{cov}(X_k | y_k)\right).$$

The samples from this distribution reflect the pattern of missing and nonmissing values for observations $k = 1, \dots, m$. You must sample from conditional distributions for each observation to preserve the correlation structure with the nonmissing values at each observation.

If you follow this procedure, the resultant filled-in values are random and generate mean and covariance estimates that are asymptotically equivalent to the ECM-derived mean and covariance estimates. Note, however, that the filled-in values are random and reflect likely samples from the distribution estimated over all the data and may not reflect “true” values for a particular observation.

Multivariate Normal Regression Functions

Financial Toolbox software has a number of functions for multivariate normal regression with or without missing data. The toolbox functions solve four classes of regression problems with functions to estimate parameters, standard errors, log-likelihood functions, and Fisher information matrices. The four classes of regression problems are:

- “Multivariate Normal Regression Without Missing Data” on page 9-12
- “Multivariate Normal Regression With Missing Data” on page 9-13
- “Least-Squares Regression with Missing Data” on page 9-13
- “Multivariate Normal Parameter Estimation with Missing Data” on page 9-14

Additional support functions are also provided, see “Support Functions” on page 9-14.

In all functions, the MATLAB representation for the number of observations (or samples) is `NumSamples = m`, the number of data series is `NumSeries = n`, and the number of model parameters is `NumParams = p`. Note that the moment estimation functions have `NumSeries = NumParams`.

The collection of observations (or samples) is stored in a MATLAB matrix `Data` such that

$$\text{Data}(k, :) = z_k^T$$

for $k = 1, \dots, \text{NumSamples}$, where `Data` is a `NumSamples`-by-`NumSeries` matrix.

For the multivariate normal regression or least-squares functions, an additional required input is the collection of design matrices that is stored as either a MATLAB matrix or a vector of cell arrays denoted as `Design`.

If `Numseries = 1`, `Design` can be a `NumSamples`-by-`NumParams` matrix. This is the “standard” form for regression on a single data series.

If `Numseries = 1`, `Design` can be either a cell array with a single cell or a cell array with `NumSamples` cells. Each cell in the cell array contains a `NumSeries`-by-`NumParams` matrix such that

$$\text{Design}\{k\} = H_k$$

for $k = 1, \dots, \text{NumSamples}$. If `Design` has a single cell, it is assumed to be the same `Design` matrix for each sample such that

$$\text{Design}\{1\} = H_1 = \dots = H_m.$$

Otherwise, `Design` must contain individual design matrices for each and every sample.

The main distinction among the four classes of regression problems depends upon how missing values are handled and where missing values are represented as the MATLAB value `NaN`. If a sample is to be ignored given any missing values in the sample, the problem is said to be a problem “without missing data.” If a sample is to be ignored if and only if every element of the sample is missing, the problem is said to be a problem “with missing data” since the estimation must account for possible `NaN` values in the data.

In general, `Data` may or may not have missing values and `Design` should have no missing values. In some cases, however, if an observation in `Data` is to be ignored, the corresponding elements in `Design` are also ignored. Consult the function reference pages for details.

Multivariate Normal Regression Without Missing Data

You can use the following functions for multivariate normal regression without missing data.

<code>mvnrmls</code>	Estimate model parameters, residuals, and the residual covariance.
----------------------	--

<code>mvnrstd</code>	Estimate standard errors of model and covariance parameters.
<code>mvnrfish</code>	Estimate the Fisher information matrix.
<code>mvnrobj</code>	Calculate the log-likelihood function.

The first two functions are the main estimation functions. The second two are supporting functions that can be used for more detailed analyses.

Multivariate Normal Regression With Missing Data

You can use the following functions for multivariate normal regression with missing data.

<code>ecmmvnrml</code>	Estimate model parameters, residuals, and the residual covariance.
<code>ecmmvnrstd</code>	Estimate standard errors of model and covariance parameters.
<code>ecmmvnrfish</code>	Estimate the Fisher information matrix.
<code>ecmmvnrobj</code>	Calculate the log-likelihood function.

The first two functions are the main estimation functions. The second two are supporting functions used for more detailed analyses.

Least-Squares Regression with Missing Data

You can use the following functions for least-squares regression with missing data or for covariance-weighted least-squares regression with a fixed covariance matrix.

<code>ecmlsrml</code>	Estimate model parameters, residuals, and the residual covariance.
<code>ecmlsrobj</code>	Calculate the least-squares objective function (pseudo log-likelihood).

To compute standard errors and estimates for the Fisher information matrix, the multivariate normal regression functions with missing data are used.

<code>ecmmvnrstd</code>	Estimate standard errors of model and covariance parameters.
-------------------------	--

<code>ecmmvnrfish</code>	Estimate the Fisher information matrix.
--------------------------	---

Multivariate Normal Parameter Estimation with Missing Data

You can use the following functions to estimate the mean and covariance of multivariate normal data.

<code>ecmmle</code>	Estimate the mean and covariance of the data.
<code>ecmstd</code>	Estimate standard errors of the mean and covariance of the data.
<code>ecmfish</code>	Estimate the Fisher information matrix.
<code>ecmhess</code>	Estimate the Fisher information matrix using the Hessian.
<code>ecmobj</code>	Calculate the log-likelihood function.

These functions behave slightly differently from the more general regression functions since they solve a specialized problem. Consult the function reference pages for details.

Support Functions

Two support functions are included.

<code>convert2sur</code>	Convert a multivariate normal regression model into an SUR model.
<code>ecmncinit</code>	Obtain initial estimates for the mean and covariance of a Data matrix.

The `convert2sur` function converts a multivariate normal regression model into a seemingly unrelated regression, or SUR, model. The second function `ecmncinit` is a specialized function to obtain initial ad hoc estimates for the mean and covariance of a Data matrix with missing data. (If there are no missing values, the estimates are the maximum likelihood estimates for the mean and covariance.)

Multivariate Normal Regression Types

In this section...

- “Regressions” on page 9-15
- “Multivariate Normal Regression” on page 9-16
- “Multivariate Normal Regression Without Missing Data” on page 9-16
- “Multivariate Normal Regression with Missing Data” on page 9-16
- “Least-Squares Regression” on page 9-16
- “Least-Squares Regression Without Missing Data” on page 9-17
- “Least-Squares Regression with Missing Data” on page 9-17
- “Covariance-Weighted Least Squares” on page 9-17
- “Covariance-Weighted Least Squares Without Missing Data” on page 9-18
- “Covariance-Weighted Least Squares with Missing Data” on page 9-18
- “Feasible Generalized Least Squares” on page 9-18
- “Feasible Generalized Least Squares Without Missing Data” on page 9-18
- “Feasible Generalized Least Squares with Missing Data” on page 9-19
- “Seemingly Unrelated Regression” on page 9-19
- “Seemingly Unrelated Regression Without Missing Data” on page 9-20
- “Seemingly Unrelated Regression with Missing Data” on page 9-20
- “Mean and Covariance Parameter Estimation” on page 9-21
- “Troubleshooting Multivariate Normal Regression” on page 9-21
- “Slow Convergence” on page 9-22
- “Nonrandom Residuals” on page 9-22
- “Nonconvergence” on page 9-23
- “Portfolios with Missing Data” on page 9-24

Regressions

Each regression function has a specific operation. This section shows how to use these functions to perform specific types of regressions. To illustrate use of the functions for various regressions, “typical” usage is shown with optional arguments kept to a minimum. For a typical regression, you estimate model parameters and residual

covariance matrices with the `mle` functions and estimate the standard errors of model parameters with the `std` functions. The regressions “without missing data” essentially ignore samples with any missing values, and the regressions “with missing data” ignore samples with every value missing.

Multivariate Normal Regression

Multivariate normal regression, or MVNR, is the “standard” implementation of the regression functions in Financial Toolbox software.

Multivariate Normal Regression Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrml(Data, Design);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Multivariate Normal Regression with Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmmvnrml(Data, Design);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Least-Squares Regression

Least-squares regression, or LSR, sometimes called ordinary least-squares or multiple linear regression, is the simplest linear regression model. It also enjoys the property that, independent of the underlying distribution, it is a best linear unbiased estimator (BLUE).

Given $m = \text{NumSamples}$ observations, the typical least-squares regression model seeks to minimize the objective function

$$\sum_{k=1}^m (Z_k - H_k b)^T (Z_k - H_k b),$$

which, within the maximum likelihood framework of the multivariate normal regression routine `mvnrml`, is equivalent to a single-iteration estimation of just the parameters to obtain `Parameters` with the initial covariance matrix `Covariance` held fixed as the identity matrix. In the case of missing data, however, the internal algorithm to handle missing data requires a separate routine `ecmlsrml` to do least-squares instead of multivariate normal regression.

Least-Squares Regression Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrml(Data, Design, 1);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Least-Squares Regression with Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmlsrml(Data, Design);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Covariance-Weighted Least Squares

Given $m = \text{NUMSAMPLES}$ observations, the typical covariance-weighted least squares, or CWLS, regression model seeks to minimize the objective function

$$\sum_{k=1}^m (Z_k - H_k b)^T C_0 (Z_k - H_k b)$$

with fixed covariance C_0 .

In most cases, C_0 is a diagonal matrix. The inverse matrix $W = C_0^{-1}$ has diagonal elements that can be considered relative “weights” for each series. Thus, CWLS is a form of weighted least squares with the weights applied across series.

Covariance-Weighted Least Squares Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmlc(Data, Design, 1, [], [], [],  
                                Covar0);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Covariance-Weighted Least Squares with Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmlsrmlc(Data, Design, [], [], [], [],  
                                Covar0);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Feasible Generalized Least Squares

An ad hoc form of least squares that has surprisingly good properties for misspecified or nonnormal models is known as feasible generalized least squares, or FGLS. The basic procedure is to do least-squares regression and then to do covariance-weighted least-squares regression with the resultant residual covariance from the first regression.

Feasible Generalized Least Squares Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmlc(Data, Design, 2, 0, 0);
```

or (to illustrate the FGLS process explicitly)

```
[Parameters, Covar0] = mvnrmlc(Data, Design, 1);  
[Parameters, Covariance] = mvnrmlc(Data, Design, 1, [], [], [],  
                                Covar0);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Feasible Generalized Least Squares with Missing Data

Estimate Parameters

```
[Parameters, Covar0] = ecmlsrml(Data, Design);
[Parameters, Covariance] = ecmlsrml(Data, Design, [], [], [], [],
                                     Covar0);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Seemingly Unrelated Regression

Given a multivariate normal regression model in standard form with a **Data** matrix and a **Design** array, it is possible to convert the problem into a seemingly unrelated regression (SUR) problem by a simple transformation of the **Design** array. The main idea of SUR is that instead of having a common parameter vector over all data series, you have a separate parameter vector associated with each separate series or with distinct groups of series that, nevertheless, share a common residual covariance. It is this ability to aggregate and disaggregate series and to perform comparative tests on each design that is the power of SUR.

To make the transformation, use the function `convert2sur`, which converts a standard-form design array into an equivalent design array to do SUR with a specified mapping of the series into **NUMGROUPS** groups. The regression functions are used in the usual manner, but with the SUR design array instead of the original design array. Instead of having **NUMPARAMS** elements, the SUR output parameter vector has **NUMGROUPS** of stacked parameter estimates, where the first **NUMPARAMS** elements of **Parameters** contain parameter estimates associated with the first group of series, the next **NUMPARAMS** elements of **Parameters** contain parameter estimates associated with the second group of series, and so on. If the model has only one series, for example, **NUMSERIES** = 1, then the SUR design array is the same as the original design array since SUR requires two or more series to generate distinct parameter estimates.

Given **NUMPARAMS** parameters and **NUMGROUPS** groups with a parameter vector **Parameters** with **NUMGROUPS** * **NUMPARAMS** elements from any of the regression routines, the following MATLAB code fragment shows how to print a table of SUR parameter estimates with rows that correspond to each parameter and columns that correspond to each group or series:

```
fprintf(1, 'Seemingly Unrelated Regression Parameter  
Estimates\n');  
fprintf(1, ' %7s ', ' ');  
fprintf(1, ' Group(%3d) ', 1:NumGroups);  
fprintf(1, '\n');  
for i = 1:NumParams  
    fprintf(1, ' %7d ', i);  
    ii = i;  
    for j = 1:NumGroups  
        fprintf(1, '%12g ', Param(ii));  
        ii = ii + NumParams;  
    end  
    fprintf(1, '\n');  
end  
fprintf(1, '\n');
```

Seemingly Unrelated Regression Without Missing Data

Form an SUR Design

```
DesignSUR = convert2sur(Design, Group);
```

Estimate Parameters

```
[Parameters, Covariance] = mvnrml(Data, DesignSUR);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, DesignSUR, Covariance);
```

Seemingly Unrelated Regression with Missing Data

Form an SUR Design

```
DesignSUR = convert2sur(Design, Group);
```

Estimate Parameters

```
[Parameters, Covariance] = ecmmvnrml(Data, DesignSUR);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, DesignSUR, Covariance);
```


Mean and Covariance Parameter Estimation

Without missing data, you can estimate the mean of your `Data` with the function `mean` and the covariance with the function `cov`. Nevertheless, the function `ecmmle` does this for you if it detects an absence of missing values. Otherwise, it uses the ECM algorithm to handle missing values.

Estimate Parameters

```
[Mean, Covariance] = ecmmle(Data);
```

Estimate Standard Errors

```
StdMean = ecmnstd(Data, Mean, Covariance);
```

Troubleshooting Multivariate Normal Regression

This section provides a few pointers to handle various technical and operational difficulties that might occur.

Biased Estimates

If samples are ignored, the number of samples used in the estimation is less than `NumSamples`. Clearly the actual number of samples used must be sufficient to obtain estimates. In addition, although the model parameters `Parameters` (or mean estimates `Mean`) are unbiased maximum likelihood estimates, the residual covariance estimate `Covariance` is biased. To convert to an unbiased covariance estimate, multiply `Covariance` by

$$\text{Count}/(\text{Count} - 1),$$

where `Count` is the actual number of samples used in the estimation with `Count ≤ NumSamples`. Note that none of the regression functions perform this adjustment.

Requirements

The regression functions, particularly the estimation functions, have several requirements. First, they must have consistent values for `NumSamples`, `NumSeries`, and `NumParams`. As a general rule, the multivariate normal regression functions require

$$\text{Count} \times \text{NumSeries} \leq \max\{\text{NumParams}, \text{NumSeries} \times (\text{NumSeries} + 1) / 2\}$$

and the least-squares regression functions require

$$\text{Count} \times \text{NumSeries} \leq \text{NumParams},$$

where **Count** is the actual number of samples used in the estimation with

$$\text{Count} \leq \text{NumSamples}.$$

Second, they must have enough nonmissing values to converge. Third, they must have a nondegenerate covariance matrix.

Although some necessary and sufficient conditions can be found in the references, general conditions for existence and uniqueness of solutions in the missing-data case do not exist. Nonconvergence is usually due to an ill-conditioned covariance matrix estimate, which is discussed in greater detail in “Nonconvergence” on page 9-23.

Slow Convergence

Since worst-case convergence of the ECM algorithm is linear, it is possible to execute hundreds and even thousands of iterations before termination of the algorithm. If you are estimating with the ECM algorithm on a regular basis with regular updates, you can use prior estimates as initial guesses for the next period's estimation. This approach often speeds things up since the default initialization in the regression functions sets the initial parameters **b** to zero and the initial covariance **C** to be the identity matrix.

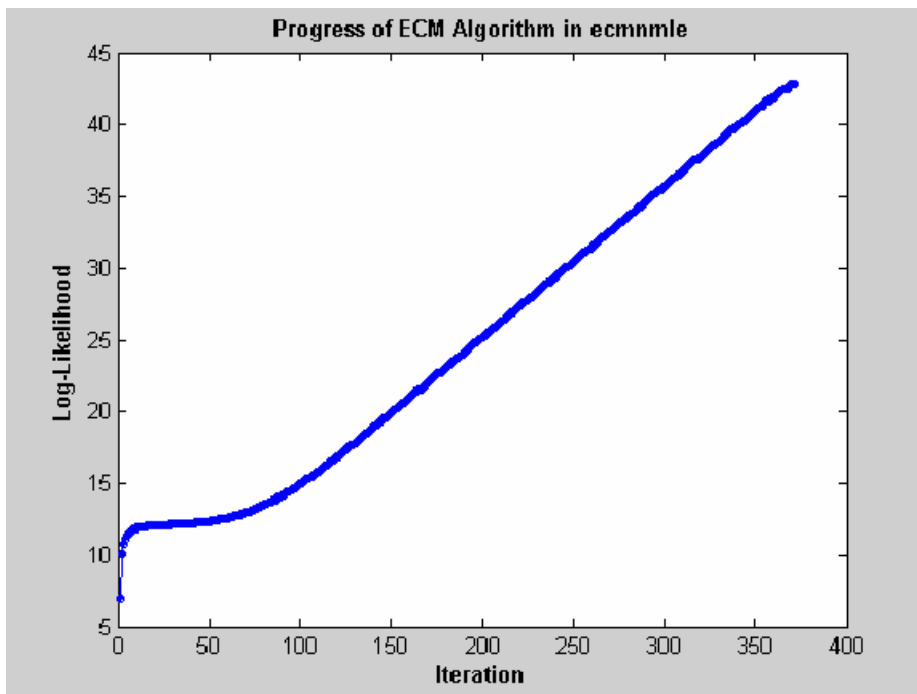
Other ad hoc approaches are possible although most approaches are problem-dependent. In particular, for mean and covariance estimation, the estimation function `ecmmle` uses a function `ecmncinit` to obtain an initial estimate.

Nonrandom Residuals

Simultaneous estimates for parameters **b** and covariances **C** require **C** to be positive-definite. Consequently, the general multivariate normal regression routines require nondegenerate residual errors. If you are faced with a model that has exact results, the least-squares routine `ecmlsrml` still works, although it provides a least-squares estimate with a singular residual covariance matrix. The other regression functions will fail.

Nonconvergence

Although the regression functions are robust and work for most “typical” cases, they can fail to converge. The main failure mode is an ill-conditioned covariance matrix, where failures are either soft or hard. A soft failure wanders endlessly toward a nearly singular covariance matrix and can be spotted if the algorithm fails to converge after about 100 iterations. If `MaxIterations` is increased to 500 and display mode is initiated (with no output arguments), a typical soft failure looks like this.



This case, which is based on 20 observations of 5 assets with 30% of data missing, shows that the log-likelihood goes linearly to infinity as the likelihood function goes to 0. In this case, the function converges but the covariance matrix is effectively singular with a smallest eigenvalue on the order of machine precision (`eps`).

For the function `ecmnml`, a hard error looks like this:

```
> In ecmninit at 60
```

```
In ecmmle at 140
???: Error using ==> ecmmle
Full covariance not positive-definite in iteration 218.
```

From a practical standpoint, if in doubt, test your residual covariance matrix from the regression routines to ensure that it is positive-definite. This is important because a soft error has a matrix that appears to be positive-definite but actually has a near-zero-valued eigenvalue to within machine precision. To do this with a covariance estimate `Covariance`, use `cond(Covariance)`, where any value greater than $1/\epsilon$ should be considered suspect.

If either type of failure occurs, however, note that the regression routine is indicating that something is probably wrong with the data. (Even with no missing data, two time series that are proportional to one another produce a singular covariance matrix.)

Portfolios with Missing Data

This example illustrates how to use the missing data algorithms for portfolio optimization and for valuation. This example works with 5 years of daily total return data for 12 computer technology stocks, with 6 hardware and 6 software companies. The example estimates the mean and covariance matrix for these stocks, forms efficient frontiers with both a naïve approach and the ECM approach, and compares results.

You can run the example directly with `ecmtechdemo.m`.

1 Load the following data file:

```
load ecmtechdemo
```

This file contains these three quantities:

- `Assets` is a cell array of the tickers for the twelve stocks in the example.
- `Data` is a 1254-by-12 matrix of 1254 daily total returns for each of the 12 stocks.
- `Dates` is a 1254-by-1 column vector of the dates associated with the data.

The time period for the data extends from April 19, 2000 to April 18, 2005.

The sixth stock in `Assets` is Google (GOOG), which started trading on August 19, 2004. Consequently, all returns before August 20, 2004 are missing and represented as `NaNs`. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past 5 years.

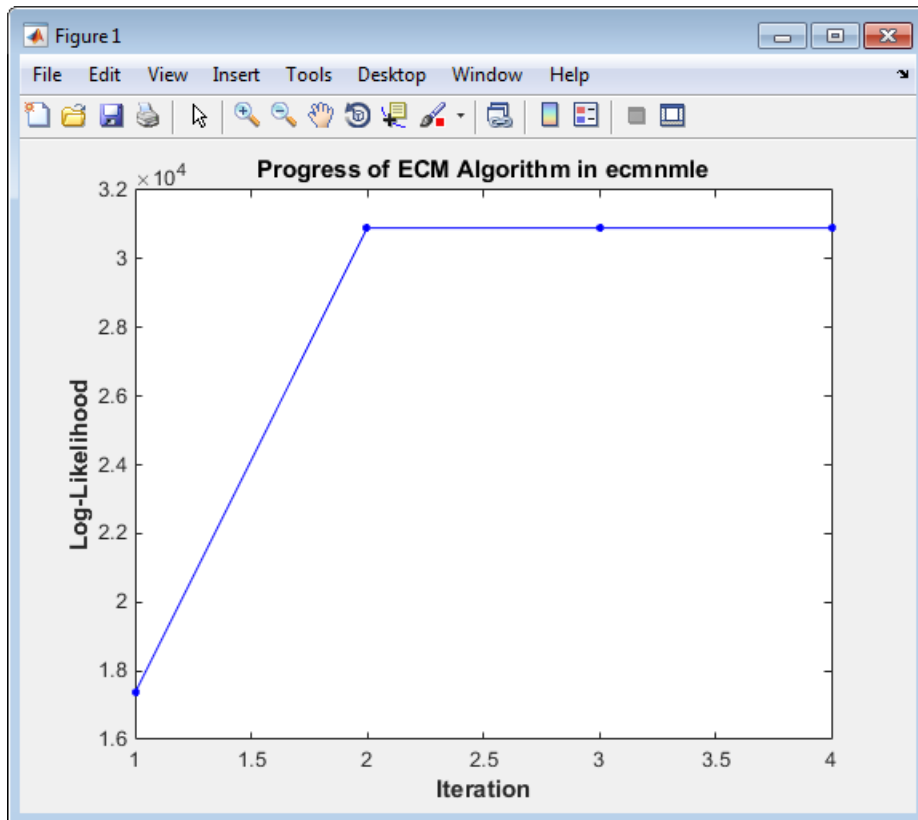
- 2 A naïve approach to the estimation of the mean and covariance for these 12 assets is to eliminate all days that have missing values for any of the 12 assets. Use the function `ecmninit` with the `nanskip` option to do this.

```
[NaNMean, NaNCovar] = ecmninit(Data, 'nanskip');
```

- 3 Contrast the result of this approach with using all available data and the function `ecmmle` to compute the mean and covariance. First, call `ecmmle` with no output arguments to establish that enough data is available to obtain meaningful estimates.

```
ecmmle(Data);
```

The following figure shows that, even with almost 87% of the Google data being NaN values, the algorithm converges after only four iterations.



- 4 Estimate the mean and covariance as computed by `ecmmle`.

```
[ECMMean, ECMCovar] = ecmmle(Data)

ECMMean =
    0.0008
    0.0008
   -0.0005
    0.0002
    0.0011
    0.0038
   -0.0003
   -0.0000
   -0.0003
   -0.0000
   -0.0000
   -0.0003
    0.0004

ECMCovar =
    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001
    0.0005    0.0006    0.0008    0.0005    0.0007    0.0003
    0.0006    0.0012    0.0008    0.0007    0.0011    0.0016

ECMCovar (continued)
    0.0005    0.0003    0.0006    0.0003    0.0005    0.0006
    0.0005    0.0003    0.0006    0.0004    0.0006    0.0012
    0.0006    0.0004    0.0008    0.0005    0.0008    0.0008
    0.0005    0.0003    0.0007    0.0004    0.0005    0.0007
    0.0005    0.0003    0.0006    0.0004    0.0007    0.0011
    0.0001    0.0002    0.0002    0.0001    0.0003    0.0016
    0.0009    0.0003    0.0005    0.0004    0.0005    0.0006
    0.0003    0.0005    0.0004    0.0003    0.0004    0.0004
    0.0005    0.0004    0.0011    0.0005    0.0007    0.0007
    0.0004    0.0003    0.0005    0.0006    0.0004    0.0005
    0.0005    0.0004    0.0007    0.0004    0.0013    0.0007
    0.0006    0.0004    0.0007    0.0005    0.0007    0.0020
```

- 5 Given estimates for the mean and covariance of asset returns derived from the naïve and ECM approaches, estimate portfolios, and associated expected returns and risks on the efficient frontier for both approaches.

```
[ECMRisk, ECMReturn, ECMWts] = portopt(ECMMean',ECMCovar,10);
[NaNRisk, NaNReturn, NaNWts] = portopt(NaNMean',NaNCovar,10);
```

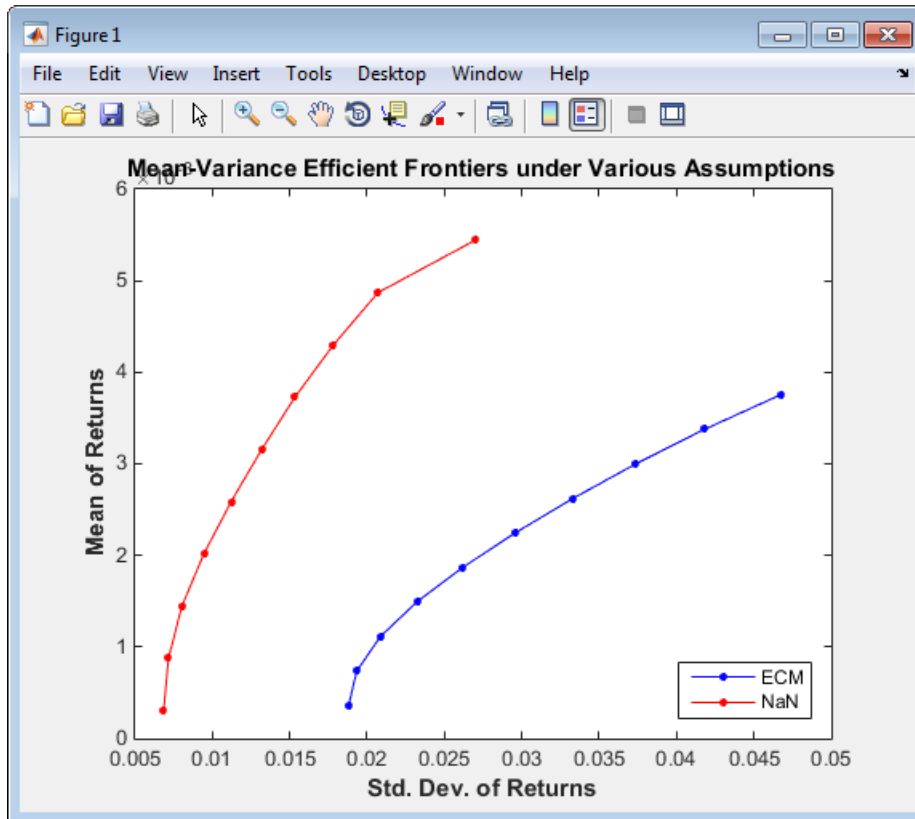
- 6 Plot the results on the same graph to illustrate the differences.

```
figure(gcf)
plot(ECMRisk,ECMReturn,'-bo','MarkerFaceColor','b','MarkerSize',3);
hold on
```

```

plot(NaNRisk,NaNReturn,'-ro','MarkerFaceColor','r','MarkerSize',3);
title('\bfMean-Variance Efficient Frontiers under Various Assumptions');
legend('ECM','NaN','Location','SouthEast');
xlabel('\bfStd. Dev. of Returns');
ylabel('\bfMean of Returns');
hold off

```



- 7 Clearly, the naïve approach is optimistic about the risk-return trade-offs for this universe of 12 technology stocks. The proof, however, lies in the portfolio weights. To view the weights, enter

```

Assets
ECMwts
NaNwts

```

which generates

```

>> Assets
ans =
      'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'
>> ECMWts
ans =
      0.0358  0.0011  -0.0000  0.0000  0.0000  0.0989
      0.0654  0.0110  0.0000  0.0000  0.0000  0.1877
      0.0923  0.0194  0.0000  0.0000  0.0000  0.2784
      0.1165  0.0264  0.0000  -0.0000  0.0000  0.3712
      0.1407  0.0334  -0.0000  0.0000  0.0000  0.4639
      0.1648  0.0403  0.0000  0.0000  -0.0000  0.5566
      0.1755  0.0457  0.0000  -0.0000  -0.0000  0.6532
      0.1845  0.0509  0.0000  0.0000  -0.0000  0.7502
      0.1093  0.0174  -0.0000  0.0000  0.0000  0.8733
      0.0000  0.0000  -0.0000  0.0000  0.0000  1.0000

>> NaNWts
ans =
      -0.0000  0.0000  -0.0000  0.1185  0.0000  0.0522
      0.0576  -0.0000  -0.0000  0.1219  0.0000  0.0854
      0.1248  -0.0000  -0.0000  0.0952  -0.0000  0.1195
      0.1969  -0.0000  -0.0000  0.0529  -0.0000  0.1551
      0.2690  -0.0000  -0.0000  0.0105  0.0000  0.1906
      0.3414  0.0000  -0.0000  -0.0000  -0.0000  0.2265
      0.4235  0.0000  -0.0000  -0.0000  -0.0000  0.2639
      0.5245  0.0000  -0.0000  -0.0000  -0.0000  0.3034
      0.6269  -0.0000  -0.0000  -0.0000  -0.0000  0.3425
      1.0000  -0.0000  -0.0000  0.0000  -0.0000  0

Assets (continued)
      'HPQ'  'IBM'  'INTC'  'MSFT'  'ORCL'  'YHOO'
ECMWts (continued)
      0.0535  0.4676  0.0000  0.3431  -0.0000  0.0000
      0.0179  0.3899  -0.0000  0.3282  0.0000  -0.0000
      0.0000  0.3025  -0.0000  0.3074  0.0000  -0.0000
      0.0000  0.2054  -0.0000  0.2806  0.0000  0.0000
      0.0000  0.1083  -0.0000  0.2538  -0.0000  0.0000
      0.0000  0.0111  -0.0000  0.2271  -0.0000  0.0000
      0.0000  0.0000  -0.0000  0.1255  -0.0000  0.0000
      0.0000  0.0000  -0.0000  0.0143  -0.0000  -0.0000
      0.0000  -0.0000  -0.0000  -0.0000  -0.0000  0.0000
      0.0000  -0.0000  -0.0000  -0.0000  -0.0000  0.0000

NaNWts (continued)
      0.0824  0.1779  0.0000  0.5691  -0.0000  0.0000
      0.1274  0.0460  0.0000  0.5617  -0.0000  -0.0000
      0.1674  -0.0000  0.0000  0.4802  0.0129  -0.0000
      0.2056  -0.0000  0.0000  0.3621  0.0274  -0.0000
      0.2438  -0.0000  0.0000  0.2441  0.0419  -0.0000
      0.2782  -0.0000  0.0000  0.0988  0.0551  -0.0000

```


0.2788	-0.0000	0.0000	-0.0000	0.0337	-0.0000
0.1721	-0.0000	0.0000	-0.0000	-0.0000	-0.0000
0.0306	-0.0000	0.0000	0.0000	0	-0.0000
0	0.0000	0.0000	-0.0000	-0.0000	-0.0000

The naïve portfolios in `NaNWts` tend to favor Apple Computer (AAPL), which happened to do well over the period from the Google IPO to the end of the estimation period, while the ECM portfolios in `ECMWts` tend to underweight Apple Computer and to recommend increased weights in Google relative to the naïve weights.

- 8** To evaluate the impact of estimation error and, in particular, the effect of missing data, use `ecmnstd` to calculate standard errors. Although it is possible to estimate the standard errors for both the mean and covariance, the standard errors for the mean estimates alone are usually the main quantities of interest.

```
StdMeanF = ecmnstd(Data,ECMMean,ECMCovar, 'fisher');
```

- 9** Calculate standard errors that use the data-generated Hessian matrix (which accounts for the possible loss of information due to missing data) with the option `HESSIAN`.

```
StdMeanH = ecmnstd(Data,ECMMean,ECMCovar, 'hessian');
```

The difference in the standard errors shows the increase in uncertainty of estimation of asset expected returns due to missing data. This can be viewed by entering

```
Assets
StdMeanH'
StdMeanF'
StdMeanH' - StdMeanF'
```

The two assets with missing data, AMZN and GOOG, are the only assets to have differences due to missing information.

Valuation with Missing Data

In this section...

“Introduction” on page 9-30

“Capital Asset Pricing Model” on page 9-30

“Estimation of the CAPM” on page 9-31

“Estimation with Missing Data” on page 9-32

“Estimation of Some Technology Stock Betas” on page 9-32

“Grouped Estimation of Some Technology Stock Betas” on page 9-34

“References” on page 9-37

Introduction

The Capital Asset Pricing Model (CAPM) is a venerable but often maligned tool to characterize comovements between asset and market prices. Although many issues arise in CAPM implementation and interpretation, one problem that practitioners face is to estimate the coefficients of the CAPM with incomplete stock price data.

This example shows how to use the missing data regression functions to estimate the coefficients of the CAPM. You can run the example directly using `CAPMdemo.m` located at `matlabroot/toolbox/finance/findemos`.

Capital Asset Pricing Model

Given a host of assumptions that can be found in the references (see Sharpe [11], Lintner [6], Jarrow [5], and Sharpe, et. al. [12]), the CAPM concludes that asset returns have a linear relationship with market returns. Specifically, given the return of all stocks that constitute a market denoted as M and the return of a riskless asset denoted as C , the CAPM states that the return of each asset R_i in the market has the expectational form

$$E[R_i] = \alpha_i + C + \beta_i(E[M] - C)$$

for assets $i = 1, \dots, n$, where β_i is a parameter that specifies the degree of comovement between a given asset and the underlying market. In other words, the expected return of each asset is equal to the return on a riskless asset plus a risk-adjusted expected market return net of riskless asset returns. The collection of parameters β_1, \dots, β_n is called asset betas.

Note that the beta of an asset has the form

$$\beta_i = \frac{\text{cov}(R_i, M)}{\text{var}(M)},$$

which is the ratio of the covariance between asset and market returns divided by the variance of market returns. If an asset has a beta = 1, the asset is said to move with the market; if an asset has a beta > 1, the asset is said to be more volatile than the market. Conversely, if an asset has a beta < 1, the asset is said to be less volatile than the market.

Estimation of the CAPM

The standard CAPM model is a linear model with additional parameters for each asset to characterize residual errors. For each of n assets with m samples of observed asset returns $R_{k,i}$, market returns M_k , and riskless asset returns C_k , the estimation model has the form

$$R_{k,i} = \alpha_i + C_k + \beta_i(M_k - C_k) + V_{k,i}$$

for samples $k = 1, \dots, m$ and assets $i = 1, \dots, n$, where α_i is a parameter that specifies the nonsystematic return of an asset, β_i is the asset beta, and $V_{k,i}$ is the residual error for each asset with associated random variable V_i .

The collection of parameters $\alpha_1, \dots, \alpha_n$ are called asset alphas. The strict form of the CAPM specifies that alphas must be zero and that deviations from zero are the result of temporary disequilibria. In practice, however, assets may have nonzero alphas, where much of active investment management is devoted to the search for assets with exploitable nonzero alphas.

To allow for the possibility of nonzero alphas, the estimation model generally seeks to estimate alphas and to perform tests to determine if the alphas are statistically equal to zero.

The residual errors V_i are assumed to have moments

$$E[V_i] = 0$$

and

$$E[V_i V_j] = S_{ij}$$

for assets $i, j = 1, \dots, n$, where the parameters S_{11}, \dots, S_{nn} are called residual or nonsystematic variances/covariances.

The square root of the residual variance of each asset, for example, $\text{sqrt}(S_{ii})$ for $i = 1, \dots, n$, is said to be the residual or nonsystematic risk of the asset since it characterizes the residual variation in asset prices that are not explained by variations in market prices.

Estimation with Missing Data

Although betas can be estimated for companies with sufficiently long histories of asset returns, it is difficult to estimate betas for recent IPOs. However, if a collection of sufficiently observable companies exists that can be expected to have some degree of correlation with the new company's stock price movements, that is, companies within the same industry as the new company, it is possible to obtain imputed estimates for new company betas with the missing-data regression routines.

Estimation of Some Technology Stock Betas

To illustrate how to use the missing-data regression routines, estimate betas for 12 technology stocks, where a single stock (GOOG) is an IPO.

- 1 Load dates, total returns, and ticker symbols for the 12 stocks from the MAT-file CAPMuniverse.

```
load CAPMuniverse
whos Assets Data Dates
```

Name	Size	Bytes	Class
Assets	1x14	952	cell array
Data	1471x14	164752	double array
Dates	1471x1	11768	double array

```
Grand total is 22135 elements using 177472 bytes
```

The assets in the model have the following symbols, where the last two series are proxies for the market and the riskless asset:

```
Assets(1:7)
Assets(8:14)
```

```
ans =
    'AAPL'    'AMZN'    'CSCO'    'DELL'    'EBAY'    'GOOG'    'HPQ'

ans =
    'IBM'    'INTC'    'MSFT'    'ORCL'    'YHOO'    'MARKET'    'CASH'
```

The data covers the period from January 1, 2000 to November 7, 2005 with daily total returns. Two stocks in this universe have missing values that are represented by NaNs. One of the two stocks had an IPO during this period and, consequently, has significantly less data than the other stocks.

- 2 Compute separate regressions for each stock, where the stocks with missing data will have estimates that reflect their reduced observability.

```
[NumSamples, NumSeries] = size(Data);
NumAssets = NumSeries - 2;

StartDate = Dates(1);
EndDate = Dates(end);

fprintf(1, 'Separate regressions with ');
fprintf(1, 'daily total return data from %s to %s ...\n', ...
    datestr(StartDate,1), datestr(EndDate,1));
fprintf(1, ' %4s %-20s %-20s %-20s\n', ' ', 'Alpha', 'Beta', 'Sigma');
fprintf(1, ' -----\n');
fprintf(1, '-----\n');

for i = 1:NumAssets
% Set up separate asset data and design matrices
    TestData = zeros(NumSamples,1);
    TestDesign = zeros(NumSamples,2);

    TestData(:) = Data(:,i) - Data(:,14);
    TestDesign(:,1) = 1.0;
    TestDesign(:,2) = Data(:,13) - Data(:,14);

% Estimate CAPM for each asset separately
    [Param, Covar] = ecmmvnrmls(TestData, TestDesign);

% Estimate ideal standard errors for covariance parameters
    [StdParam, StdCovar] = ecmmvnrstd(TestData, TestDesign, ...
        Covar, 'fisher');

% Estimate sample standard errors for model parameters
    StdParam = ecmmvnrstd(TestData, TestDesign, Covar, 'hessian');

% Set up results for output
    Alpha = Param(1);
    Beta = Param(2);
    Sigma = sqrt(Covar);

    StdAlpha = StdParam(1);
```

```

StdBeta = StdParam(2);
StdSigma = sqrt(StdCovar);

% Display estimates
fprintf(' %4s %9.4f (%8.4f) %9.4f (%8.4f) %9.4f (%8.4f)\n', ...
        Assets{i},Alpha(1),abs(Alpha(1)/StdAlpha(1)), ...
        Beta(1),abs(Beta(1)/StdBeta(1)),Sigma(1),StdSigma(1));
end

```

This code fragment generates the following table.

```

Separate regressions with daily total return data from 03-Jan-2000
to 07-Nov-2005 ...

```

	Alpha	Beta	Sigma
AAPL	0.0012 (1.3882)	1.2294 (17.1839)	0.0322 (0.0062)
AMZN	0.0006 (0.5326)	1.3661 (13.6579)	0.0449 (0.0086)
CSCO	-0.0002 (0.2878)	1.5653 (23.6085)	0.0298 (0.0057)
DELL	-0.0000 (0.0368)	1.2594 (22.2164)	0.0255 (0.0049)
EBAY	0.0014 (1.4326)	1.3441 (16.0732)	0.0376 (0.0072)
GOOG	0.0046 (3.2107)	0.3742 (1.7328)	0.0252 (0.0071)
HPQ	0.0001 (0.1747)	1.3745 (24.2390)	0.0255 (0.0049)
IBM	-0.0000 (0.0312)	1.0807 (28.7576)	0.0169 (0.0032)
INTC	0.0001 (0.1608)	1.6002 (27.3684)	0.0263 (0.0050)
MSFT	-0.0002 (0.4871)	1.1765 (27.4554)	0.0193 (0.0037)
ORCL	0.0000 (0.0389)	1.5010 (21.1855)	0.0319 (0.0061)
YHOO	0.0001 (0.1282)	1.6543 (19.3838)	0.0384 (0.0074)

The Alpha column contains alpha estimates for each stock that are near zero as expected. In addition, the t-statistics (which are enclosed in parentheses) generally reject the hypothesis that the alphas are nonzero at the 99.5% level of significance.

The Beta column contains beta estimates for each stock that also have t-statistics enclosed in parentheses. For all stocks but GOOG, the hypothesis that the betas are nonzero is accepted at the 99.5% level of significance. It seems, however, that GOOG does not have enough data to obtain a meaningful estimate for beta since its t-statistic would imply rejection of the hypothesis of a nonzero beta.

The Sigma column contains residual standard deviations, that is, estimates for nonsystematic risks. Instead of t-statistics, the associated standard errors for the residual standard deviations are enclosed in parentheses.

Grouped Estimation of Some Technology Stock Betas

To estimate stock betas for all 12 stocks, set up a joint regression model that groups all 12 stocks within a single design. (Since each stock has the same design matrix, this model is actually an example of seemingly unrelated regression.) The routine to

estimate model parameters is `ecmmvnrml`, and the routine to estimate standard errors is `ecmmvnrstd`.

Because GOOG has a significant number of missing values, a direct use of the missing data routine `ecmmvnrml` takes 482 iterations to converge. This can take a long time to compute. For the sake of brevity, the parameter and covariance estimates after the first 480 iterations are contained in a MAT-file and are used as initial estimates to compute stock betas.

```
load CAPMgroupparam
whos Param0 Covar0
```

```
Name          Size          Bytes  Class
Covar0        12x12          1152  double array
Param0        24x1           192   double array
```

Grand total is 168 elements using 1344 bytes

Now estimate the parameters for the collection of 12 stocks.

```
fprintf(1, '\n');
fprintf(1, 'Grouped regression with ');
fprintf(1, 'daily total return data from %s to %s ...\n', ...
    datestr(StartDate,1), datestr(EndDate,1));
fprintf(1, ' %4s %-20s %-20s %-20s\n', '', 'Alpha', 'Beta', 'Sigma');
fprintf(1, ' ----- ');
fprintf(1, '-----\n');

NumParams = 2 * NumAssets;

% Set up grouped asset data and design matrices
TestData = zeros(NumSamples, NumAssets);
TestDesign = cell(NumSamples, 1);
Design = zeros(NumAssets, NumParams);

for k = 1:NumSamples
    for i = 1:NumAssets
        TestData(k,i) = Data(k,i) - Data(k,14);
        Design(i,2*i - 1) = 1.0;
        Design(i,2*i) = Data(k,13) - Data(k,14);
    end
    TestDesign{k} = Design;
end

% Estimate CAPM for all assets together with initial parameter
% estimates
[Param, Covar] = ecmmvnrml(TestData, TestDesign, [], [], [],...
    Param0, Covar0);

% Estimate ideal standard errors for covariance parameters
[StdParam, StdCovar] = ecmmvnrstd(TestData, TestDesign, Covar,...
```

```

    'fisher');

% Estimate sample standard errors for model parameters
StdParam = ecmmvnrstd(TestData, TestDesign, Covar, 'hessian');

% Set up results for output
Alpha = Param(1:2:end-1);
Beta = Param(2:2:end);
Sigma = sqrt(diag(Covar));

StdAlpha = StdParam(1:2:end-1);
StdBeta = StdParam(2:2:end);
StdSigma = sqrt(diag(StdCovar));

% Display estimates
for i = 1:NumAssets
    fprintf(' %4s %9.4f (%8.4f) %9.4f (%8.4f) %9.4f (%8.4f)\n', ...
        Assets{i}, Alpha(i), abs(Alpha(i)/StdAlpha(i)), ...
        Beta(i), abs(Beta(i)/StdBeta(i)), Sigma(i), StdSigma(i));
end

```

This code fragment generates the following table.

```

Grouped regression with daily total return data from 03-Jan-2000
to 07-Nov-2005 ...

```

	Alpha	Beta	Sigma
AAPL	0.0012 (1.3882)	1.2294 (17.1839)	0.0322 (0.0062)
AMZN	0.0007 (0.6086)	1.3673 (13.6427)	0.0450 (0.0086)
CSCO	-0.0002 (0.2878)	1.5653 (23.6085)	0.0298 (0.0057)
DELL	-0.0000 (0.0368)	1.2594 (22.2164)	0.0255 (0.0049)
EBAY	0.0014 (1.4326)	1.3441 (16.0732)	0.0376 (0.0072)
GOOG	0.0041 (2.8907)	0.6173 (3.1100)	0.0337 (0.0065)
HPQ	0.0001 (0.1747)	1.3745 (24.2390)	0.0255 (0.0049)
IBM	-0.0000 (0.0312)	1.0807 (28.7576)	0.0169 (0.0032)
INTC	0.0001 (0.1608)	1.6002 (27.3684)	0.0263 (0.0050)
MSFT	-0.0002 (0.4871)	1.1765 (27.4554)	0.0193 (0.0037)
ORCL	0.0000 (0.0389)	1.5010 (21.1855)	0.0319 (0.0061)
YHOO	0.0001 (0.1282)	1.6543 (19.3838)	0.0384 (0.0074)

Although the results for complete-data stocks are the same, note that the beta estimates for AMZN and GOOG (the two stocks with missing values) are different from the estimates derived for each stock separately. Since AMZN has few missing values, the differences in the estimates are small. With GOOG, however, the differences are more pronounced.

The t-statistic for the beta estimate of GOOG is now significant at the 99.5% level of significance. Note, however, that the t-statistics for beta estimates are based on standard errors from the sample Hessian which, in contrast to the Fisher information matrix, accounts for the increased uncertainty in an estimate due to missing values. If the t-statistic is obtained from the more optimistic Fisher information matrix, the t-statistic

for GOOG is 8.25. Thus, despite the increase in uncertainty due to missing data, GOOG nonetheless has a statistically significant estimate for beta.

Finally, note that the beta estimate for GOOG is 0.62 —a value that may require some explanation. Although the market has been volatile over this period with sideways price movements, GOOG has steadily appreciated in value. Consequently, it is less tightly correlated with the market, implying that it is less volatile than the market ($\beta < 1$).

References

- [1] Caines, Peter E. *Linear Stochastic Systems*. John Wiley & Sons, Inc., 1988.
- [2] Cramér, Harald. *Mathematical Methods of Statistics*. Princeton University Press, 1946.
- [3] Dempster, A.P, N.M. Laird, and D.B Rubin. “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, 1977, pp. 1-37.
- [4] Greene, William H. *Econometric Analysis*, 5th ed., Pearson Education, Inc., 2003.
- [5] Jarrow, R.A. *Finance Theory*, Prentice-Hall, Inc., 1988.
- [6] Lintner, J. “The Valuation of Risk Assets and the Selection of Risky Investments in Stocks,” *Review of Economics and Statistics*, Vol. 14, 1965, pp. 13-37.
- [7] Little, Roderick J. A and Donald B. Rubin. *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.
- [8] Meng, Xiao-Li and Donald B. Rubin. “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.
- [9] Sexton, Joe and Anders Rygh Swensen. “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.
- [10] Shafer, J. L. *Analysis of Incomplete Multivariate Data*, Chapman & Hall/CRC, 1997.
- [11] Sharpe, W. F. “Capital Asset Prices: A Theory of Market Equilibrium Under Conditions of Risk,” *Journal of Finance*, Vol. 19, 1964, pp. 425-442.
- [12] Sharpe, W. F., G. J. Alexander, and J. V. Bailey. *Investments*, 6th ed., Prentice-Hall, Inc., 1999.

Solving Sample Problems

- “Introduction” on page 10-2
- “Common Problems in Finance” on page 10-3
- “Producing Graphics with the Toolbox” on page 10-21

Introduction

This section shows how Financial Toolbox functions solve real-world problems. The examples ship with the toolbox as MATLAB files. Try them by entering the commands directly or by executing the code.

This section contains two major topics:

- “Common Problems in Finance” on page 10-3

Shows how the toolbox solves real-world financial problems, specifically:

- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-8
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-17
- “Producing Graphics with the Toolbox” on page 10-21

Shows how the toolbox produces presentation-quality graphics by solving these problems:

- “Plotting an Efficient Frontier” on page 10-21
- “Plotting Sensitivities of an Option” on page 10-24
- “Plotting Sensitivities of a Portfolio of Options” on page 10-26

Common Problems in Finance

In this section...

“Sensitivity of Bond Prices to Interest Rates” on page 10-3

“Bond Portfolio for Hedging Duration and Convexity” on page 10-6

“Bond Prices and Yield Curve Parallel Shifts” on page 10-8

“Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12

“Greek-Neutral Portfolios of European Stock Options” on page 10-14

“Term Structure Analysis and Interest-Rate Swaps” on page 10-17

Sensitivity of Bond Prices to Interest Rates

Macaulay and *modified duration* measure the sensitivity of a bond's price to changes in the level of interest rates. *Convexity* measures the change in duration for small shifts in the yield curve, and thus measures the second-order price sensitivity of a bond. Both measures can gauge the vulnerability of a bond portfolio's value to changes in the level of interest rates.

Alternatively, analysts can use duration and convexity to construct a bond portfolio that is partly hedged against small shifts in the term structure. If you combine bonds in a portfolio whose duration is zero, the portfolio is insulated, to some extent, against interest rate changes. If the portfolio convexity is also zero, this insulation is even better. However, since hedging costs money or reduces expected return, you need to know how much protection results from hedging duration alone compared to hedging both duration and convexity.

This example demonstrates a way to analyze the relative importance of duration and convexity for a bond portfolio using some of the SIA-compliant bond functions in Financial Toolbox software. Using duration, it constructs a first-order approximation of the change in portfolio price to a level shift in interest rates. Then, using convexity, it calculates a second-order approximation. Finally, it compares the two approximations with the true price change resulting from a change in the yield curve.

Step 1

Define three bonds using values for the settlement date, maturity date, face value, and coupon rate. For simplicity, accept default values for the coupon payment periodicity (semiannual), end-of-month payment rule (rule in effect), and day-count basis (actual/

actual). Also, synchronize the coupon payment structure to the maturity date (no odd first or last coupon dates). Any inputs for which defaults are accepted are set to empty matrices ([]) as placeholders where appropriate.

```
Settle      = '19-Aug-1999';  
Maturity    = ['17-Jun-2010'; '09-Jun-2015'; '14-May-2025'];  
Face        = [100; 100; 1000];  
CouponRate = [0.07; 0.06; 0.045];
```

Also, specify the yield curve information.

```
Yields = [0.05; 0.06; 0.065];
```

Step 2

Use Financial Toolbox functions to calculate the price, modified duration in years, and convexity in years of each bond.

The true price is quoted (clean) price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields, CouponRate,...  
Settle, Maturity, 2, 0, [], [], [], [], Face);  
  
Durations = bnddury(Yields, CouponRate, Settle, Maturity, 2, 0,...  
[], [], [], [], [], Face);  
  
Convexities = bndconvy(Yields, CouponRate, Settle, Maturity, 2, 0,...  
[], [], [], [], [], Face);  
  
Prices = CleanPrice + AccruedInterest;
```

Step 3

Choose a hypothetical amount by which to shift the yield curve (here, 0.2 percentage point or 20 basis points).

```
dY = 0.002;
```

Weight the three bonds equally, and calculate the actual quantity of each bond in the portfolio, which has a total value of \$100,000.

```
PortfolioPrice = 100000;  
PortfolioWeights = ones(3,1)/3;  
PortfolioAmounts = PortfolioPrice * PortfolioWeights ./ Prices;
```

Step 4

Calculate the modified duration and convexity of the portfolio. Note that the portfolio duration or convexity is a weighted average of the durations or convexities of the

individual bonds. Calculate the first- and second-order approximations of the percent price change as a function of the change in the level of interest rates.

```
PortfolioDuration = PortfolioWeights' * Durations;
PortfolioConvexity = PortfolioWeights' * Convexities;
PercentApprox1 = -PortfolioDuration * dY * 100;
```

```
PercentApprox2 = PercentApprox1 + ...
PortfolioConvexity*dY^2*100/2.0;
```

Step 5

Estimate the new portfolio price using the two estimates for the percent price change.

```
PriceApprox1 = PortfolioPrice + ...
PercentApprox1 * PortfolioPrice/100;
```

```
PriceApprox2 = PortfolioPrice + ...
PercentApprox2 * PortfolioPrice/100;
```

Step 6

Calculate the true new portfolio price by shifting the yield curve.

```
[CleanPrice, AccruedInterest] = bndprice(Yields + dY,...
CouponRate, Settle, Maturity, 2, 0, [], [], [], [], [],...
Face);
```

```
NewPrice = PortfolioAmounts' * (CleanPrice + AccruedInterest);
```

Step 7

Compare the results. The analysis results are as follows:

- The original portfolio price was \$100,000.
- The yield curve shifted up by 0.2 percentage point or 20 basis points.
- The portfolio duration and convexity are 10.3181 and 157.6346, respectively. These will be needed for “Bond Portfolio for Hedging Duration and Convexity” on page 10-6.
- The first-order approximation, based on modified duration, predicts the new portfolio price (`PriceApprox1`) will be \$97,936.37.
- The second-order approximation, based on duration and convexity, predicts the new portfolio price (`PriceApprox2`) will be \$97,967.90.

- The true new portfolio price (**NewPrice**) for this yield curve shift is \$97,967.51.
- The estimate using duration and convexity is quite good (at least for this fairly small shift in the yield curve), but only slightly better than the estimate using duration alone. The importance of convexity increases as the magnitude of the yield curve shift increases. Try a larger shift (dY) to see this effect.

The approximation formulas in this example consider only parallel shifts in the term structure, because both formulas are functions of dY , the change in yield. The formulas are not well-defined unless each yield changes by the same amount. In actual financial markets, changes in yield curve level typically explain a substantial portion of bond price movements. However, other changes in the yield curve, such as slope, may also be important and are not captured here. Also, both formulas give local approximations whose accuracy deteriorates as dY increases in size. You can demonstrate this by running the program with larger values of dY .

Bond Portfolio for Hedging Duration and Convexity

This example constructs a bond portfolio to hedge the portfolio of “Sensitivity of Bond Prices to Interest Rates” on page 10-3. It assumes a long position in (holding) the portfolio, and that three other bonds are available for hedging. It chooses weights for these three other bonds in a new portfolio so that the duration and convexity of the new portfolio match those of the original portfolio. Taking a short position in the new portfolio, in an amount equal to the value of the first portfolio, partially hedges against parallel shifts in the yield curve.

Recall that portfolio duration or convexity is a weighted average of the durations or convexities of the individual bonds in a portfolio. As in the previous example, this example uses modified duration in years and convexity in years. The hedging problem therefore becomes one of solving a system of linear equations, which is an easy to do in MATLAB software.

Step 1

Define three bonds available for hedging the original portfolio. Specify values for the settlement date, maturity date, face value, and coupon rate. For simplicity, accept default values for the coupon payment periodicity (semiannual), end-of-month payment rule (rule in effect), and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (that is, no odd first or last coupon dates). Set any inputs for which defaults are accepted to empty matrices (`[]`) as placeholders where

appropriate. The intent is to hedge against duration and convexity and constrain total portfolio price.

```
Settle      = '19-Aug-1999';
Maturity    = ['15-Jun-2005'; '02-Oct-2010'; '01-Mar-2025'];
Face        = [500; 1000; 250];
CouponRate  = [0.07; 0.066; 0.08];
```

Also, specify the yield curve for each bond.

```
Yields = [0.06; 0.07; 0.075];
```

Step 2

Use Financial Toolbox functions to calculate the price, modified duration in years, and convexity in years of each bond.

The true price is quoted (clean price plus accrued interest).

```
[CleanPrice, AccruedInterest] = bndprice(Yields, CouponRate, ...
Settle, Maturity, 2, 0, [], [], [], [], [], Face);
```

```
Durations = bnddury(Yields, CouponRate, Settle, Maturity, ...
2, 0, [], [], [], [], [], Face);
```

```
Convexities = bndconvy(Yields, CouponRate, Settle, ...
Maturity, 2, 0, [], [], [], [], [], Face);
```

```
Prices = CleanPrice + AccruedInterest;
```

Step 3

Set up and solve the system of linear equations whose solution is the weights of the new bonds in a new portfolio with the same duration and convexity as the original portfolio. In addition, scale the weights to sum to 1; that is, force them to be portfolio weights. You can then scale this unit portfolio to have the same price as the original portfolio. Recall that the original portfolio duration and convexity are 10.3181 and 157.6346, respectively. Also, note that the last row of the linear system ensures that the sum of the weights is unity.

```
A = [Durations'
      Convexities'
      1 1 1];
```

```
b = [ 10.3181
```

```
157.6346  
1];
```

```
Weights = A\b;
```

Step 4

Compute the duration and convexity of the hedge portfolio, which should now match the original portfolio.

```
PortfolioDuration = Weights' * Durations;  
PortfolioConvexity = Weights' * Convexities;
```

Step 5

Finally, scale the unit portfolio to match the value of the original portfolio and find the number of bonds required to insulate against small parallel shifts in the yield curve.

```
PortfolioValue = 100000;  
HedgeAmounts = Weights ./ Prices * PortfolioValue;
```

Step 6

Compare the results.

- As required, the duration and convexity of the new portfolio are **10.3181** and **157.6346**, respectively.
- The hedge amounts for bonds 1, 2, and 3 are **-57.37**, **71.70**, and **216.27**, respectively.

Notice that the hedge matches the duration, convexity, and value (\$100,000) of the original portfolio. If you are holding that first portfolio, you can hedge by taking a short position in the new portfolio.

Just as the approximations of the first example are appropriate only for small parallel shifts in the yield curve, the hedge portfolio is appropriate only for reducing the impact of small level changes in the term structure.

Bond Prices and Yield Curve Parallel Shifts

Often bond portfolio managers want to consider more than just the sensitivity of a portfolio's price to a small shift in the yield curve, particularly if the investment horizon is long. This example shows how MATLAB software can help you to visualize the price

behavior of a portfolio of bonds over a wide range of yield curve scenarios, and as time progresses toward maturity.

This example uses Financial Toolbox bond pricing functions to evaluate the impact of time-to-maturity and yield variation on the price of a bond portfolio. It plots the portfolio value and shows the behavior of bond prices as yield and time vary.

Step 1

Specify values for the settlement date, maturity date, face value, coupon rate, and coupon payment periodicity of a four-bond portfolio. For simplicity, accept default values for the end-of-month payment rule (rule in effect) and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (no odd first or last coupon dates). Any inputs for which defaults are accepted are set to empty matrices ([]) as placeholders where appropriate.

```
Settle      = '15-Jan-1995';
Maturity    = datenum(['03-Apr-2020'; '14-May-2025'; ...
                    '09-Jun-2019'; '25-Feb-2019']);
Face        = [1000; 1000; 1000; 1000];
CouponRate  = [0; 0.05; 0; 0.055];
Periods     = [0; 2; 0; 2];
```

Also, specify the points on the yield curve for each bond.

```
Yields = [0.078; 0.09; 0.075; 0.085];
```

Step 2

Use Financial Toolbox functions to calculate the true bond prices as the sum of the quoted price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields, ...
    CouponRate, Settle, Maturity, Periods, ...
    [], [], [], [], [], [], Face);
Prices = CleanPrice + AccruedInterest;
```

Step 3

Assume the value of each bond is \$25,000, and determine the quantity of each bond such that the portfolio value is \$100,000.

```
BondAmounts = 25000 ./ Prices;
```

Step 4

Compute the portfolio price for a rolling series of settlement dates over a range of yields. The evaluation dates occur annually on January 15, beginning on 15-Jan-1995 (settlement) and extending out to 15-Jan-2018. Thus, this step evaluates portfolio price on a grid of time of progression (dT) and interest rates (dY).

```
dy = -0.05:0.005:0.05; % Yield changes

D = datevec(Settle); % Get date components
dt = datenum(D(1):2018, D(2), D(3)); % Get evaluation dates

[dT, dY] = meshgrid(dt, dy); % Create grid

NumTimes = length(dt); % Number of time steps
NumYields = length(dy); % Number of yield changes
NumBonds = length(Maturity); % Number of bonds

% Preallocate vector
Prices = zeros(NumTimes*NumYields, NumBonds);
```

Now that the grid and price vectors have been created, compute the price of each bond in the portfolio on the grid one bond at a time.

```
for i = 1:NumBonds

    [CleanPrice, AccruedInterest] = bndprice(Yields(i)+...
        dY(:), CouponRate(i), dT(:), Maturity(i), Periods(i),...
        [], [], [], [], [], [], Face(i));

    Prices(:,i) = CleanPrice + AccruedInterest;
```

```
end
```

Scale the bond prices by the quantity of bonds.

```
Prices = Prices * BondAmounts;
```

Reshape the bond values to conform to the underlying evaluation grid.

```
Prices = reshape(Prices, NumYields, NumTimes);
```

Step 5

Plot the price of the portfolio as a function of settlement date and a range of yields, and as a function of the change in yield (dY). This plot illustrates the interest rate sensitivity

of the portfolio as time progresses (dt), under a range of interest rate scenarios. With the following graphics commands, you can visualize the three-dimensional surface relative to the current portfolio value (that is, \$100,000).

```
figure                                % Open a new figure window
surf(dt, dy, Prices)                 % Draw the surface
```

Add the base portfolio value to the existing surface plot.

```
hold on                               % Add the current value for reference
basemesh = mesh(dt, dy, 100000*ones(NumYields, NumTimes));
```

Make it transparent, plot it so the price surface shows through, and draw a box around the plot.

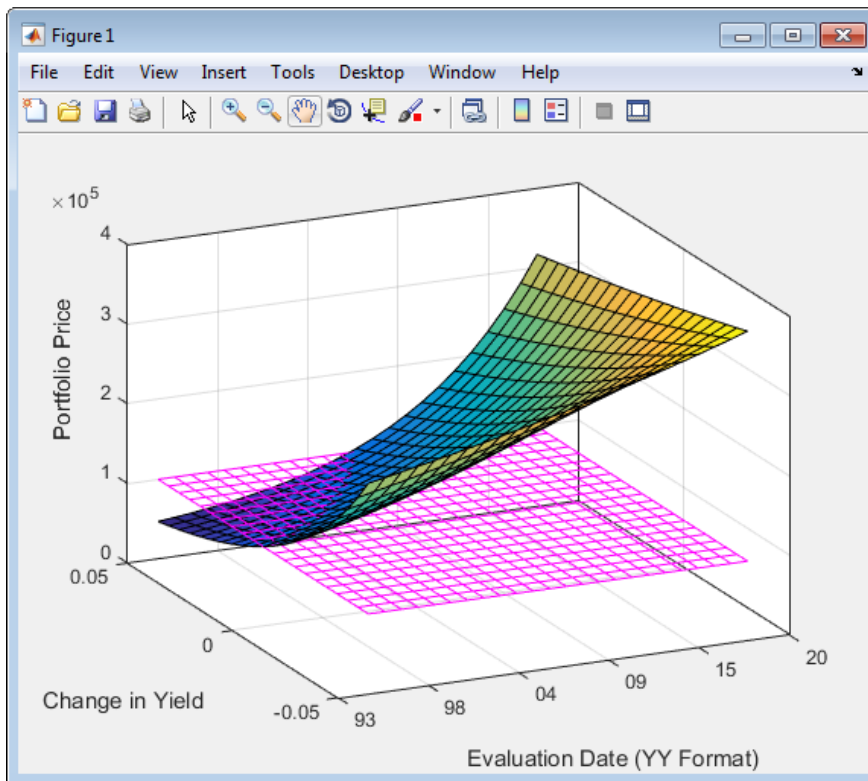
```
set(basemesh, 'facecolor', 'none');
set(basemesh, 'edgecolor', 'm');
set(gca, 'box', 'on');
```

Plot the x -axis using two-digit year (YY format) labels for ticks.

```
dateaxis('x', 11);
```

Add axis labels and set the three-dimensional viewpoint. MATLAB produces the figure.

```
xlabel('Evaluation Date (YY Format)');
ylabel('Change in Yield');
zlabel('Portfolio Price');
hold off
view(-25,25);
```



MATLAB three-dimensional graphics allow you to visualize the interest rate risk experienced by a bond portfolio over time. This example assumed parallel shifts in the term structure, but it might similarly have allowed other components to vary, such as the level and slope.

Bond Prices and Yield Curve Nonparallel Shifts

Key rate duration enables you to determine the sensitivity of the price of a bond to nonparallel shifts in the yield curve. This example uses `bndkrdur` to construct a portfolio to hedge the interest rate risk of a U.S. Treasury bond maturing in 20 years.

```
Settle = datenum('2-Dec-2008');
CouponRate = 5.500/100;
Maturity = datenum('15-Aug-2028');
Price = 128.68;
```

The interest rate risk of this bond is hedged with the following four on-the-run Treasury bonds:

The 30-year bond.

```
Maturity_30 = datenum('15-May-2038');
Coupon_30 = .045;
Price_30 = 124.69;
```

The ten-year note.

```
Maturity_10 = datenum('15-Nov-2018');
Coupon_10 = .0375;
Price_10 = 109.35;
```

The five-year note.

```
Maturity_05 = datenum('30-Nov-2013');
Coupon_05 = .02;
Price_05 = 101.67;
```

The two-year note.

```
Maturity_02 = datenum('30-Nov-2010');
Coupon_02 = .01250;
Price_02 = 100.72;
```

You can get the Treasury spot or zero curve from: <http://www.treas.gov/offices/domestic-finance/debt-management/interest-rate/yield.shtml>.

```
ZeroDates = daysadd(Settle,[30 90 180 360 360*2 360*3 360*5 ...
360*7 360*10 360*20 360*30]);
ZeroRates = ([0.09 0.07 0.44 0.81 0.90 1.16 1.71 2.13 2.72 3.51 3.22]/100)';
```

Step 1

Compute the key rate durations for both the bond and the hedging portfolio:

```
BondKRD = bndkrdur([ZeroDates ZeroRates], CouponRate, Settle,...
Maturity,'keyrates',[2 5 10 20]);
HedgeMaturity = [Maturity_02;Maturity_05;Maturity_10;Maturity_30];
HedgeCoupon = [Coupon_02;Coupon_05;Coupon_10;Coupon_30];
HedgeKRD = bndkrdur([ZeroDates ZeroRates], HedgeCoupon,...
Settle, HedgeMaturity, 'keyrates',[2 5 10 20]);
```

Step 2

Compute the dollar durations for each of the instruments and each of the key rates (assuming holding 100 bonds):

```
PortfolioDD = 100*Price* BondKRD;  
HedgeDD = bsxfun(@times, HedgeKRD,[Price_30;Price_10;Price_05;Price_02]);
```

Step 3

Compute the number of bonds to sell short to obtain a key rate duration that is 0 for the entire portfolio:

```
NumBonds = PortfolioDD/HedgeDD;
```

```
NumBonds =
```

```
    3.8973    6.1596   23.0282   80.0522
```

These results indicate selling 4, 6, 23 and 80 bonds respectively of the 2-, 5-, 10-, and 30-year bonds achieves a portfolio that is neutral with respect to the 2-, 5-, 10-, and 30-year spot rates.

Greek-Neutral Portfolios of European Stock Options

The option sensitivity measures familiar to most option traders are often referred to as the *greeks*: *delta*, *gamma*, *vega*, *lambda*, *rho*, and *theta*. Delta is the price sensitivity of an option with respect to changes in the price of the underlying asset. It represents a first-order sensitivity measure analogous to duration in fixed income markets. Gamma is the sensitivity of an option's delta to changes in the price of the underlying asset, and represents a second-order price sensitivity analogous to convexity in fixed income markets. Vega is the price sensitivity of an option with respect to changes in the volatility of the underlying asset. See “Pricing and Analyzing Equity Derivatives” on page 2-41 or the Glossary for other definitions.

The greeks of a particular option are a function of the model used to price the option. However, given enough different options to work with, a trader can construct a portfolio with any desired values for its greeks. For example, to insulate the value of an option portfolio from small changes in the price of the underlying asset, one trader might construct an option portfolio whose delta is zero. Such a portfolio is then said to be “delta neutral.” Another trader may want to protect an option portfolio from larger changes in the price of the underlying asset, and so might construct a portfolio whose delta and gamma are both zero. Such a portfolio is both delta and gamma neutral. A third trader may want to construct a portfolio insulated from small changes in the volatility of the underlying asset in addition to delta and gamma neutrality. Such a portfolio is then delta, gamma, and vega neutral.

Using the Black-Scholes model for European options, this example creates an equity option portfolio that is simultaneously delta, gamma, and vega neutral. The value of a particular greek of an option portfolio is a weighted average of the corresponding greek of each individual option. The weights are the quantity of each option in the portfolio. Hedging an option portfolio thus involves solving a system of linear equations, an easy process in MATLAB.

Step 1

Create an input data matrix to summarize the relevant information. Each row of the matrix contains the standard inputs to Financial Toolbox Black-Scholes suite of functions: column 1 contains the current price of the underlying stock; column 2 the strike price of each option; column 3 the time to-expiry of each option in years; column 4 the annualized stock price volatility; and column 5 the annualized dividend rate of the underlying asset. Note that rows 1 and 3 are data related to call options, while rows 2 and 4 are data related to put options.

```
DataMatrix = [100.000  100  0.2  0.3  0          % Call
              119.100  125  0.2  0.2  0.025     % Put
              87.200   85  0.1  0.23  0         % Call
              301.125  315  0.5  0.25  0.0333] % Put
```

Also, assume the annualized risk-free rate is 10% and is constant for all maturities of interest.

```
RiskFreeRate = 0.10;
```

For clarity, assign each column of `DataMatrix` to a column vector whose name reflects the type of financial data in the column.

```
StockPrice   = DataMatrix(:,1);
StrikePrice  = DataMatrix(:,2);
ExpiryTime   = DataMatrix(:,3);
Volatility    = DataMatrix(:,4);
DividendRate = DataMatrix(:,5);
```

Step 2

Based on the Black-Scholes model, compute the prices, and the delta, gamma, and vega sensitivity greeks of each of the four options. Note that the functions `blsprice` and `blsdelta` have two outputs, while `blsgamma` and `blsvega` have only one. The price and delta of a call option differ from the price and delta of an otherwise equivalent put option, in contrast to the gamma and vega sensitivities, which are valid for both calls and puts.

```
[CallPrices, PutPrices] = blsprice(StockPrice, StrikePrice,...  
RiskFreeRate, ExpiryTime, Volatility, DividendRate);
```

```
[CallDeltas, PutDeltas] = blsdelta(StockPrice,...  
StrikePrice, RiskFreeRate, ExpiryTime, Volatility,...  
DividendRate);
```

```
Gammas = blsgamma(StockPrice, StrikePrice, RiskFreeRate,...  
ExpiryTime, Volatility , DividendRate)';
```

```
Vegas = blsvega(StockPrice, StrikePrice, RiskFreeRate,...  
ExpiryTime, Volatility , DividendRate)';
```

Extract the prices and deltas of interest to account for the distinction between call and puts.

```
Prices = [CallPrices(1) PutPrices(2) CallPrices(3)...  
PutPrices(4)];
```

```
Deltas = [CallDeltas(1) PutDeltas(2) CallDeltas(3)...  
PutDeltas(4)];
```

Step 3

Now, assuming an arbitrary portfolio value of \$17,000, set up and solve the linear system of equations such that the overall option portfolio is simultaneously delta, gamma, and vega-neutral. The solution computes the value of a particular greek of a portfolio of options as a weighted average of the corresponding greek of each individual option in the portfolio. The system of equations is solved using the back slash (\) operator discussed in “Solving Simultaneous Linear Equations” on page 1-13.

```
A = [Deltas; Gammas; Vegas; Prices];  
b = [0; 0; 0; 17000];  
OptionQuantities = A\b; % Quantity (number) of each option.
```

Step 4

Finally, compute the market value, delta, gamma, and vega of the overall portfolio as a weighted average of the corresponding parameters of the component options. The weighted average is computed as an inner product of two vectors.

```
PortfolioValue = Prices * OptionQuantities;  
PortfolioDelta = Deltas * OptionQuantities;  
PortfolioGamma = Gammas * OptionQuantities;  
PortfolioVega = Vegas * OptionQuantities;
```

The output for these computations is:

Option	Price	Delta	Gamma	Vega	Quantity
1	6.3441	0.5856	0.0290	17.4293	22332.6131
2	6.6035	-0.6255	0.0353	20.0347	6864.0731
3	4.2993	0.7003	0.0548	9.5837	-15654.8657
4	22.7694	-0.4830	0.0074	83.5225	-4510.5153

Portfolio Value: \$17000.00
 Portfolio Delta: 0.00
 Portfolio Gamma: -0.00
 Portfolio Vega : 0.00

You can verify that the portfolio value is \$17,000 and that the option portfolio is indeed delta, gamma, and vega neutral, as desired. Hedges based on these measures are effective only for small changes in the underlying variables.

Term Structure Analysis and Interest-Rate Swaps

This example illustrates some of the term-structure analysis functions found in Financial Toolbox software. Specifically, it illustrates how to derive implied zero (*spot*) and forward curves from the observed market prices of coupon-bearing bonds. The zero and forward curves implied from the market data are then used to price an interest rate swap agreement.

In an interest rate swap, two parties agree to a periodic exchange of cash flows. One of the cash flows is based on a fixed interest rate held constant throughout the life of the swap. The other cash flow stream is tied to some variable index rate. Pricing a swap at inception amounts to finding the fixed rate of the swap agreement. This fixed rate, appropriately scaled by the notional principal of the swap agreement, determines the periodic sequence of fixed cash flows.

In general, interest rate swaps are priced from the forward curve such that the variable cash flows implied from the series of forward rates and the periodic sequence of fixed-rate cash flows have the same current value. Thus, interest rate swap pricing and term structure analysis are intimately related.

Step 1

Specify values for the settlement date, maturity dates, coupon rates, and market prices for 10 U.S. Treasury Bonds. This data allows you to price a five-year swap with net cash flow payments exchanged every six months. For simplicity, accept default values for the

end-of-month payment rule (rule in effect) and day-count basis (actual/actual). To avoid issues of accrued interest, assume that all Treasury Bonds pay semiannual coupons and that settlement occurs on a coupon payment date.

```
Settle = datenum('15-Jan-1999');  
  
BondData = { '15-Jul-1999' 0.06000 99.93  
             '15-Jan-2000' 0.06125 99.72  
             '15-Jul-2000' 0.06375 99.70  
             '15-Jan-2001' 0.06500 99.40  
             '15-Jul-2001' 0.06875 99.73  
             '15-Jan-2002' 0.07000 99.42  
             '15-Jul-2002' 0.07250 99.32  
             '15-Jan-2003' 0.07375 98.45  
             '15-Jul-2003' 0.07500 97.71  
             '15-Jan-2004' 0.08000 98.15};
```

`BondData` is an instance of a MATLAB *cell array*, indicated by the curly braces (`{}`).

Next assign the date stored in the cell array to `Maturity`, `CouponRate`, and `Prices` vectors for further processing.

```
Maturity = datenum(char(BondData{:},1));  
CouponRate = [BondData{:},2]';  
Prices = [BondData{:},3]';  
Period = 2; % semiannual coupons
```

Step 2

Now that the data has been specified, use the term structure function `zbtprice` to bootstrap the zero curve implied from the prices of the coupon-bearing bonds. This implied zero curve represents the series of zero-coupon Treasury rates consistent with the prices of the coupon-bearing bonds such that arbitrage opportunities will not exist.

```
ZeroRates = zbtprice([Maturity CouponRate], Prices, Settle);
```

The zero curve, stored in `ZeroRates`, is quoted on a semiannual bond basis (the periodic, six-month, interest rate is doubled to annualize). The first element of `ZeroRates` is the annualized rate over the next six months, the second element is the annualized rate over the next 12 months, and so on.

Step 3

From the implied zero curve, find the corresponding series of implied forward rates using the term structure function `zero2fwd`.

```
ForwardRates = zero2fwd(ZeroRates, Maturity, Settle);
```

The forward curve, stored in `ForwardRates`, is also quoted on a semiannual bond basis. The first element of `ForwardRates` is the annualized rate applied to the interval between settlement and six months after settlement, the second element is the annualized rate applied to the interval from six months to 12 months after settlement, and so on. This implied forward curve is also consistent with the observed market prices such that arbitrage activities will be unprofitable. Since the first forward rate is also a zero rate, the first element of `ZeroRates` and `ForwardRates` are the same.

Step 4

Now that you have derived the zero curve, convert it to a sequence of discount factors with the term structure function `zero2disc`.

```
DiscountFactors = zero2disc(ZeroRates, Maturity, Settle);
```

Step 5

From the discount factors, compute the present value of the variable cash flows derived from the implied forward rates. For plain interest rate swaps, the notional principle remains constant for each payment date and cancels out of each side of the present value equation. The next line assumes unit notional principle.

```
PresentValue = sum((ForwardRates/Period) .* DiscountFactors);
```

Step 6

Compute the swap's price (the fixed rate) by equating the present value of the fixed cash flows with the present value of the cash flows derived from the implied forward rates. Again, since the notional principle cancels out of each side of the equation, it is simply assumed to be 1.

```
SwapFixedRate = Period * PresentValue / sum(DiscountFactors);
```

The output for these computations is:

Zero Rates	Forward Rates
0.0614	0.0614
0.0642	0.0670
0.0660	0.0695
0.0684	0.0758
0.0702	0.0774
0.0726	0.0846
0.0754	0.0925

0.0795	0.1077
0.0827	0.1089
0.0868	0.1239

Swap Price (Fixed Rate) = 0.0845

All rates are in decimal format. The swap price, 8.45%, would likely be the mid-point between a market-maker's bid/ask quotes.

Producing Graphics with the Toolbox

In this section...

“Introduction” on page 10-21

“Plotting an Efficient Frontier” on page 10-21

“Plotting Sensitivities of an Option” on page 10-24

“Plotting Sensitivities of a Portfolio of Options” on page 10-26

Introduction

Financial Toolbox and MATLAB graphics functions work together to produce presentation quality graphics, as these examples show. The examples ship with the toolbox as MATLAB files. Try them by entering the commands directly or by executing the code. For help using MATLAB plotting functions, see “Creating Line Plots” in the MATLAB documentation.

Plotting an Efficient Frontier

This example plots the efficient frontier of a hypothetical portfolio of three assets. It illustrates how to specify the expected returns, standard deviations, and correlations of a portfolio of assets, how to convert standard deviations and correlations into a covariance matrix, and how to compute and plot the efficient frontier from the returns and covariance matrix. The example also illustrates how to randomly generate a set of portfolio weights, and how to add the random portfolios to an existing plot for comparison with the efficient frontier. The file for this example is `ftgex1.m`.

First, specify the expected returns, standard deviations, and correlation matrix for a hypothetical portfolio of three assets.

```
Returns      = [0.1 0.15 0.12];
STDs         = [0.2 0.25 0.18];
```

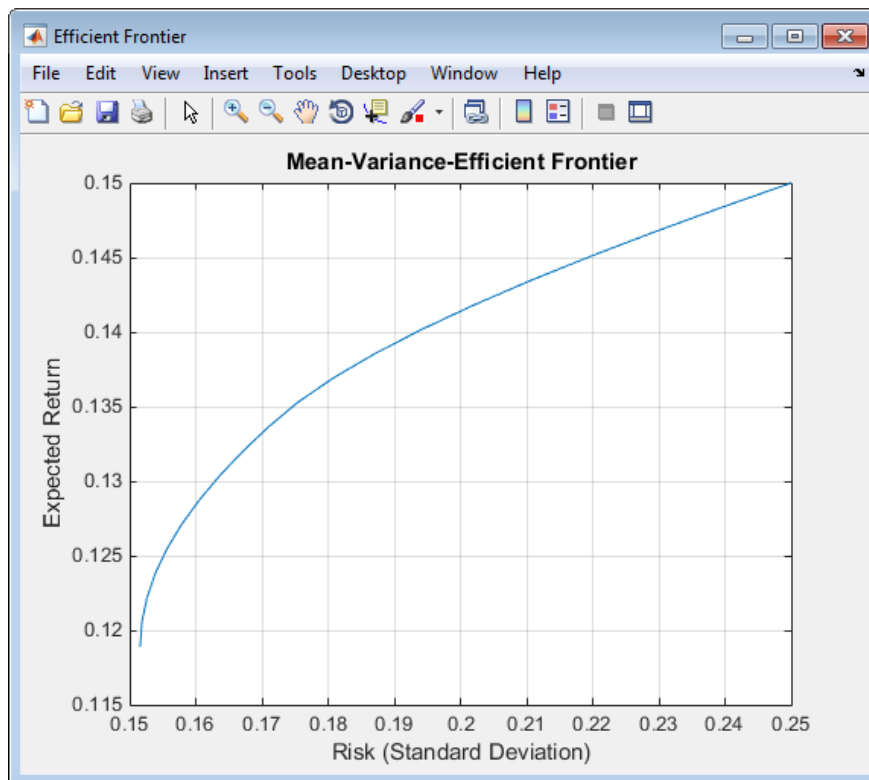
```
Correlations = [ 1  0.3  0.4
                 0.3  1   0.3
                 0.4 0.3  1  ];
```

Convert the standard deviations and correlation matrix into a variance-covariance matrix with the Financial Toolbox function `corr2cov`.

```
Covariances = corr2cov(STDs, Correlations);
```

Evaluate and plot the efficient frontier at 20 points along the frontier, using the function `portopt` and the expected returns and corresponding covariance matrix. Although rather elaborate constraints can be placed on the assets in a portfolio, for simplicity accept the default constraints and scale the total value of the portfolio to 1 and constrain the weights to be positive (no short-selling).

```
portopt>Returns, Covariances, 20)
```



Now that the efficient frontier is displayed, randomly generate the asset weights for 1000 portfolios starting from the MATLAB initial state.

```
rng('default')  
Weights = rand(1000, 3);
```


The previous line of code generates three columns of uniformly distributed random weights, but does not guarantee they sum to 1. The following code segment normalizes the weights of each portfolio so that the total of the three weights represent a valid portfolio.

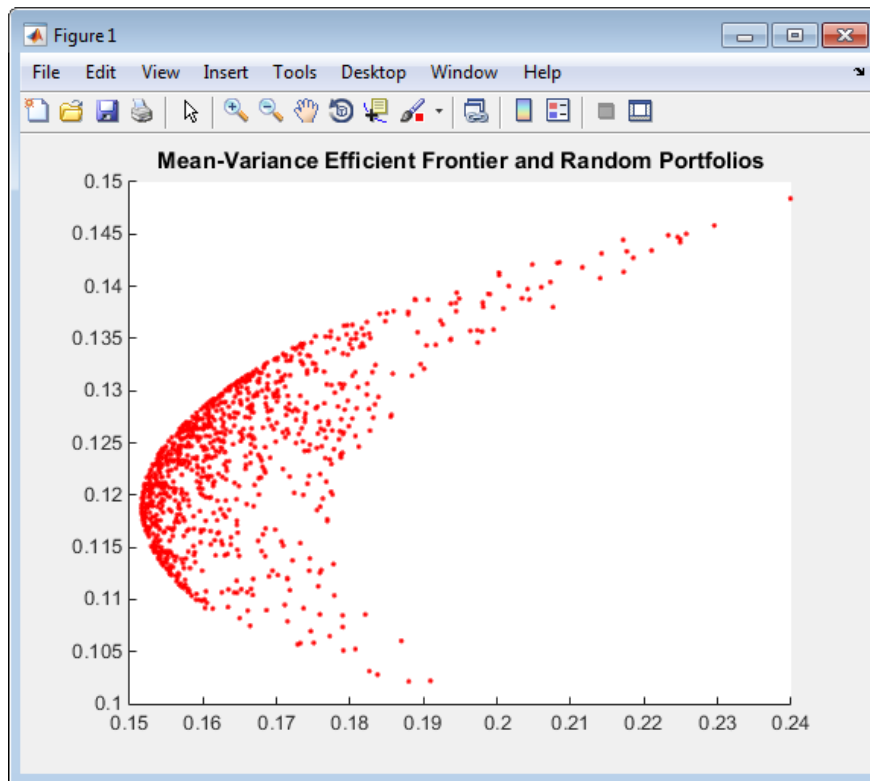
```
Total = sum(Weights, 2);    % Add the weights
Total = Total(:,ones(3,1)); % Make size-compatible matrix
Weights = Weights./Total;  % Normalize so sum = 1
```

Given the 1000 random portfolios just created, compute the expected return and risk of each portfolio associated with the weights.

```
[PortRisk, PortReturn] = portstats>Returns, Covariances, ...
                             Weights);
```

Finally, hold the current graph, and plot the returns and risks of each portfolio on top of the existing efficient frontier for comparison. After plotting, annotate the graph with a title and return the graph to default holding status (any subsequent plots will erase the existing data). The efficient frontier appears in blue, while the 1000 random portfolios appear as a set of red dots on or below the frontier.

```
hold on
plot (PortRisk, PortReturn, '.r')
title('Mean-Variance Efficient Frontier and Random Portfolios')
hold off
```



Plotting Sensitivities of an Option

This example creates a three-dimensional plot showing how gamma changes relative to price for a Black-Scholes option. Recall that gamma is the second derivative of the option price relative to the underlying security price. The plot shows a three-dimensional surface whose z -value is the gamma of an option as price (x -axis) and time (y -axis) vary. It adds yet a fourth dimension by showing option delta (the first derivative of option price to security price) as the color of the surface. The file for this example is `ftgex2.m`.

First set the price range of the options, and set the time range to one year divided into half-months and expressed as fractions of a year.

```
Range = 10:70;  
Span = length(Range);
```

```
j = 1:0.5:12;
Newj = j(ones(Span,1),:)' / 12;
```

For each time period create a vector of prices from 10 to 70 and create a matrix of all ones.

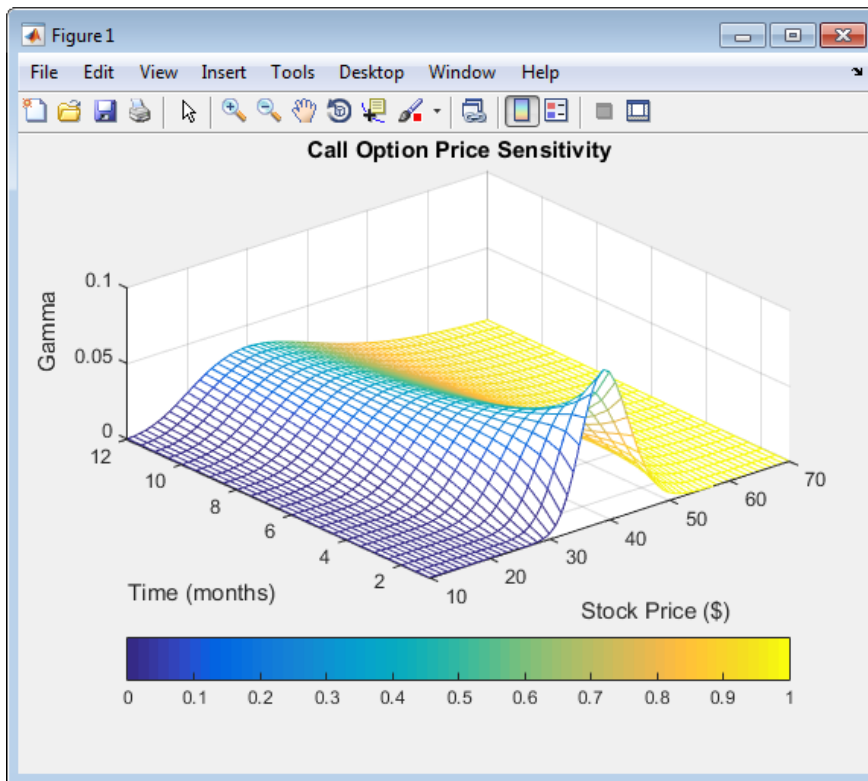
```
JSpan = ones(length(j),1);
NewRange = Range(JSpan,:);
Pad = ones(size(Newj));
```

Calculate the toolbox gamma and delta sensitivity functions (greeks). (Recall that gamma is the second derivative of the option price with respect to the stock price, and delta is the first derivative of the option price with respect to the stock price.) The exercise price is \$40, the risk-free interest rate is 10%, and volatility is 0.35 for all prices and periods.

```
ZVal = blsgamma(NewRange, 40*Pad, 0.1*Pad, Newj, 0.35*Pad);
Color = blsdelta(NewRange, 40*Pad, 0.1*Pad, Newj, 0.35*Pad);
```

Display the greeks as a function of price and time. Gamma is the z-axis; delta is the color.

```
mesh(Range, j, ZVal, Color);
xlabel('Stock Price ($)');
ylabel('Time (months)');
zlabel('Gamma');
title('Call Option Price Sensitivity');
axis([10 70 1 12 -inf inf]);
view(-40, 50);
colorbar('horiz');
```



Plotting Sensitivities of a Portfolio of Options

This example plots gamma as a function of price and time for a portfolio of 10 Black-Scholes options. The plot shows a three-dimensional surface. For each point on the surface, the height (z -value) represents the sum of the gammas for each option in the portfolio weighted by the amount of each option. The x -axis represents changing price, and the y -axis represents time. The plot adds a fourth dimension by showing delta as surface color. This has applications in hedging.

The file for this example is `ftgex3.m`.

First set up the portfolio with arbitrary data. Current prices range from \$20 to \$90 for each option. Set corresponding exercise prices for each option.

```
Range = 20:90;
```

```
PLen = length(Range);
ExPrice = [75 70 50 55 75 50 40 75 60 35];
```

Set all risk-free interest rates to 10%, and set times to maturity in days. Set all volatilities to 0.35. Set the number of options of each instrument, and allocate space for matrices.

```
Rate = 0.1*ones(10,1);
Time = [36 36 36 27 18 18 18 9 9 9];
Sigma = 0.35*ones(10,1);
NumOpt = 1000*[4 8 3 5 5.5 2 4.8 3 4.8 2.5];
ZVal = zeros(36, PLen);
Color = zeros(36, PLen);
```

For each instrument, create a matrix (of size `Time` by `PLen`) of prices for each period.

```
for i = 1:10
    Pad = ones(Time(i),PLen);
    NewR = Range(ones(Time(i),1),:);
```

Create a vector of time periods 1 to `Time`; and a matrix of times, one column for each price.

```
T = (1:Time(i))';
NewT = T(:,ones(PLen,1));
```

Call the toolbox gamma and delta sensitivity functions to compute gamma and delta.

```
ZVal(36-Time(i)+1:36,:) = ZVal(36-Time(i)+1:36,:) ...
    + NumOpt(i) * blsgamma(NewR, ExPrice(i)*Pad, ...
    Rate(i)*Pad, NewT/36, Sigma(i)*Pad);

Color(36-Time(i)+1:36,:) = Color(36-Time(i)+1:36,:) ...
    + NumOpt(i) * blsdelta(NewR, ExPrice(i)*Pad, ...
    Rate(i)*Pad, NewT/36, Sigma(i)*Pad);
```

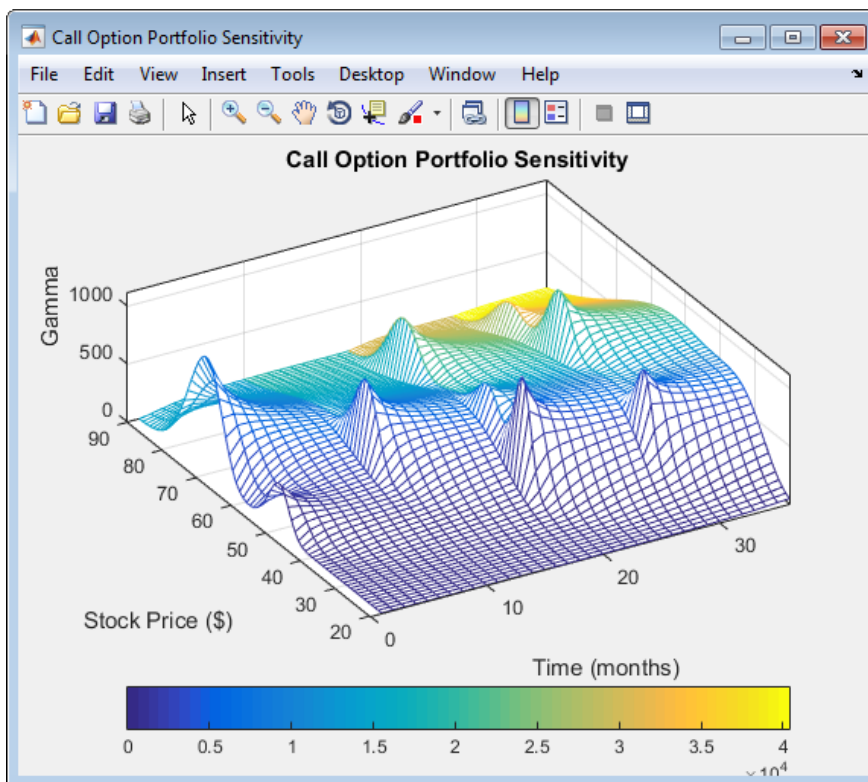
```
end
```

Draw the surface as a mesh, set the viewpoint, and reverse the x -axis because of the viewpoint. The axes range from 20 to 90, 0 to 36, and $-\infty$ to ∞ .

```
mesh(Range, 1:36, ZVal, Color);
view(60,60);
set(gca, 'xdir','reverse', 'tag', 'mesh_axes_3');
axis([20 90 0 36 -inf inf]);
```

Add a title and axis labels and draw a box around the plot. Annotate the colors with a bar and label the colorbar.

```
title('Call Option Portfolio Sensitivity');  
xlabel('Stock Price ($)');  
ylabel('Time (months)');  
zlabel('Gamma');  
set(gca, 'box', 'on');  
colorbar('horiz');
```



Financial Time Series Analysis

- “Analyzing Financial Time Series” on page 11-2
- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16

Analyzing Financial Time Series

Financial Toolbox software provides a collection of tools for the analysis of time series data in the financial markets. The toolbox contains a financial time series object constructor and several methods that operate on and analyze the object. Financial engineers working with time series data, such as equity prices or daily interest fluctuations, can use these tools for more intuitive data management than by using regular vectors or matrices.

This section discusses how to create a financial time series object in one of two ways:

- “Using the Constructor” on page 11-3
- “Transforming a Text File” on page 11-13

`chartfts` is a graphical tool for visualizing financial time series objects. You can find this discussion in “Visualizing Financial Time Series Objects” on page 11-16.

Creating Financial Time Series Objects

In this section...

“Introduction” on page 11-3

“Using the Constructor” on page 11-3

“Transforming a Text File” on page 11-13

Introduction

Financial Toolbox software provides two ways to create a financial time series object:

- At the command line using the object constructor `fints`
- From a text data file through the function `ascii2fts`

The structure of the object minimally consists of a description field, a frequency indicator field, the date vector field, and at least one data series vector. The names for the fields are fixed for the first three fields: `desc`, `freq`, and `dates`. You can specify names of your choice for any data series vectors. If you do not specify names, the object uses the default names `series1`, `series2`, `series3`, and so on.

If time-of-day information is incorporated in the date vector, the object contains an additional field named `times`.

Using the Constructor

The object constructor function `fints` has five different syntaxes. These forms exist to simplify object construction. The syntaxes vary according to the types of input arguments presented to the constructor. The syntaxes are

- Single Matrix Input
 - See “Time-of-Day Information Excluded” on page 11-4.
 - See “Time-of-Day Information Included” on page 11-6.
- Separate Vector Input
 - See “Time-of-Day Information Excluded” on page 11-7.

- See “Time-of-Day Information Included” on page 11-8.
- See “Data Name Input” on page 11-9.
- See “Frequency Indicator Input” on page 11-11.
- See “Description Field Input” on page 11-12.

Single Matrix Input

The date information provided with this syntax must be in serial date number format. The date number may or may not include time-of-day information.

Note If you are unfamiliar with the concepts of date strings and serial date numbers, consult “Handle and Convert Dates” on page 2-4.

Time-of-Day Information Excluded

```
fts = fints(dates_and_data)
```

In this simplest form of syntax, the input must be at least a two-column matrix. The first column contains the dates in serial date format; the second column is the data series. The input matrix can have more than two columns, each additional column representing a different data series or set of observations.

If the input is a two-column matrix, the output object contains four fields: `desc`, `freq`, `dates`, and `series1`. The description field, `desc`, defaults to blanks ' ', and the frequency indicator field, `freq`, defaults to 0. The dates field, `dates`, contains the serial dates from the first column of the input matrix, while the data series field, `series1`, has the data from the second column of the input matrix.

The first example makes two financial time series objects. The first one has only one data series, while the other has more than one. A random vector provides the values for the data series. The range of dates is arbitrarily chosen using the `today` function:

```
date_series = (today:today+100)';  
data_series = exp(randn(1, 101))';  
dates_and_data = [date_series data_series];  
fts1 = fints(dates_and_data);
```

Examine the contents of the object `fts1` create. The actual date series you observe will vary according to the day when you run the example (the value of `today`). Also, your

values in `series1` will differ from those shown, depending upon the sequence of random numbers generated:

```
fts1 =
  desc: (none)
  freq: Unknown (0)

  'dates: (101)'   'series1: (101)'
  '12-Jul-1999'   [         0.3124]
  '13-Jul-1999'   [         3.2665]
  '14-Jul-1999'   [         0.9847]
  '15-Jul-1999'   [         1.7095]
  '16-Jul-1999'   [         0.4885]
  '17-Jul-1999'   [         0.5192]
  '18-Jul-1999'   [         1.3694]
  '19-Jul-1999'   [         1.1127]
  '20-Jul-1999'   [         6.3485]
  '21-Jul-1999'   [         0.7595]
  '22-Jul-1999'   [         9.1390]
  '23-Jul-1999'   [         4.5201]
  '24-Jul-1999'   [         0.1430]
  '25-Jul-1999'   [         0.1863]
  '26-Jul-1999'   [         0.5635]
  '27-Jul-1999'   [         0.8304]
  '28-Jul-1999'   [         1.0090]...
```

The output is truncated for brevity. There are actually 101 data points in the object.

Note that the `desc` field displays as `(none)` instead of `' '`, and that the contents of the object display as cell array elements. Although the object displays as such, it should be thought of as a MATLAB structure containing the default field names for a single data series object: `desc`, `freq`, `dates`, and `series1`.

Now create an object with more than one data series in it:

```
date_series = (today:today+100)';
data_series1 = exp(randn(1, 101))';
data_series2 = exp(randn(1, 101))';
dates_and_data = [date_series data_series1 data_series2];
fts2 = fints(dates_and_data);
```

Now look at the object created (again in abbreviated form):

```
fts2 =
  desc: (none)
  freq: Unknown (0)

  'dates: (101)'   'series1: (101)'   'series2: (101)'
  '12-Jul-1999'   [         0.5816]   [         1.2816]
```

```
'13-Jul-1999' [ 5.1253] [ 0.9262]
'14-Jul-1999' [ 2.2824] [ 5.6869]
'15-Jul-1999' [ 1.2596] [ 5.0631]
'16-Jul-1999' [ 1.9574] [ 1.8709]
'17-Jul-1999' [ 0.6017] [ 1.0962]
'18-Jul-1999' [ 2.3546] [ 0.4459]
'19-Jul-1999' [ 1.3080] [ 0.6304]
'20-Jul-1999' [ 1.8682] [ 0.2451]
'21-Jul-1999' [ 0.3509] [ 0.6876]
'22-Jul-1999' [ 4.6444] [ 0.6244]
'23-Jul-1999' [ 1.5441] [ 5.7621]
'24-Jul-1999' [ 0.1470] [ 2.1238]
'25-Jul-1999' [ 1.5999] [ 1.0671]
'26-Jul-1999' [ 3.5764] [ 0.7462]
'27-Jul-1999' [ 1.8937] [ 1.0863]
'28-Jul-1999' [ 3.9780] [ 2.1516]...
```

The second data series name defaults to `series2`, as expected.

Before you can perform any operations on the object, you must set the frequency indicator field `freq` to the valid frequency of the data series contained in the object. You can leave the description field `desc` blank.

To set the frequency indicator field to a daily frequency, enter

```
fts2.freq = 1, or
```

```
fts2.freq = 'daily'
```

For more information, see `fints`.

Time-of-Day Information Included

The serial date number used with this form of the `fints` function can incorporate time-of-day information. When time-of-day information is present, the output of the function contains a field `times` that indicates the time of day.

If you recode the previous example to include time-of-day information, you can see the additional column present in the output object:

```
time_series = (now:now+100)';
data_series = exp(randn(1, 101))';
times_and_data = [time_series data_series];
fts1 = fints(times_and_data);
```

```
fts1 =
```

```
desc: (none)
freq: Unknown (0)
```

```
'dates: (101)'      'times: (101)'      'series1: (101)'
'29-Nov-2001'      '14:57'             [          0.5816]
'30-Nov-2001'      '14:57'             [          5.1253]
'01-Dec-2001'      '14:57'             [          2.2824]
'02-Dec-2001'      '14:57'             [          1.2596]...
```

Separate Vector Input

The date information provided with this syntax can be in serial date number or date string format. The date information may or may not include time-of-day information.

Time-of-Day Information Excluded

```
fts = fints(dates, data)
```

In this second syntax the dates and data series are entered as separate vectors to `fints`, the financial time series object constructor function. The `dates` vector must be a column vector, while the data series `data` can be a column vector (if there is only one data series) or a column-oriented matrix (for multiple data series). A column-oriented matrix, in this context, indicates that each column is a set of observations. Different columns are different sets of data series.

Here is an example:

```
dates = (today:today+100)';
data_series1 = exp(randn(1, 101))';
data_series2 = exp(randn(1, 101))';
data = [data_series1 data_series2]';
fts = fints(dates, data)
```

```
fts =
desc: (none)
freq: Unknown (0)

'dates: (101)'      'series1: (101)'      'series2: (101)'
'12-Jul-1999'      [          0.5816]      [          1.2816]
'13-Jul-1999'      [          5.1253]      [          0.9262]
'14-Jul-1999'      [          2.2824]      [          5.6869]
'15-Jul-1999'      [          1.2596]      [          5.0631]
'16-Jul-1999'      [          1.9574]      [          1.8709]
'17-Jul-1999'      [          0.6017]      [          1.0962]
'18-Jul-1999'      [          2.3546]      [          0.4459]
'19-Jul-1999'      [          1.3080]      [          0.6304]
'20-Jul-1999'      [          1.8682]      [          0.2451]
'21-Jul-1999'      [          0.3509]      [          0.6876]
'22-Jul-1999'      [          4.6444]      [          0.6244]
'23-Jul-1999'      [          1.5441]      [          5.7621]
'24-Jul-1999'      [          0.1470]      [          2.1238]
'25-Jul-1999'      [          1.5999]      [          1.0671]
'26-Jul-1999'      [          3.5764]      [          0.7462]
'27-Jul-1999'      [          1.8937]      [          1.0863]
```

```
'28-Jul-1999' [ 3.9780] [ 2.1516]...
```

The result is exactly the same as the first syntax. The only difference between the first and second syntax is the way the inputs are entered into the constructor function.

Time-of-Day Information Included

With this form of the function you can enter the time-of-day information either as a serial date number or as a date string. If more than one serial date and time are present, the entry must be in the form of a column-oriented matrix. If more than one string date and time are present, the entry must be a column-oriented cell array of dates and times.

With date string input the dates and times can initially be separate column-oriented date and time series, but you must concatenate them into a single column-oriented cell array before entering them as the first input to `fints`.

For date string input the allowable formats are

- 'ddmmyy hh:mm' or 'ddmmyyyy hh:mm'
- 'mm/dd/yy hh:mm' or 'mm/dd/yyyy hh:mm'
- 'dd-mmm-yy hh:mm' or 'dd-mmm-yyyy hh:mm'
- 'mmm.dd,yy hh:mm' or 'mmm.dd,yyyy hh:mm'

The next example shows time-of-day information input as serial date numbers in a column-oriented matrix:

```
f = fints([now;now+1],[1:2])
f =
    desc: (none)
    freq: Unknown (0)

    'dates: (2)'    'times: (2)'    'series1: (2)'
    '29-Nov-2001'  '15:22'        [ 1]
    '30-Nov-2001'  '15:22'        [ 2]
```

If the time-of-day information is in date string format, you must provide it to `fints` as a column-oriented cell array:

```
f = fints({'01-Jan-2001 12:00';'02-Jan-2001 12:00'},[1:2])
f =
```

```

desc: (none)
freq: Unknown (0)

'dates: (2)'      'times: (2)'      'series1: (2)'
'01-Jan-2001'    '12:00'          [          1]
'02-Jan-2001'    '12:00'          [          2]

```

If the dates and times are in date string format and contained in separate matrices, you must concatenate them before using the date and time information as input to `fints`:

```

dates = ['01-Jan-2001'; '02-Jan-2001'; '03-Jan-2001'];
times = ['12:00'; '12:00'; '12:00'];
dates_time = cellstr([dates, repmat(' ', size(dates,1),1), times]);
f = fints(dates_time, (1:3))

```

f =

```

desc: (none)
freq: Unknown (0)

'dates: (3)'      'times: (3)'      'series1: (3)'
'01-Jan-2001'    '12:00'          [          1]
'02-Jan-2001'    '12:00'          [          2]
'03-Jan-2001'    '12:00'          [          3]

```

Data Name Input

```
fts = fints(dates, data, datanames)
```

The third syntax lets you specify the names for the data series with the argument `datanames`. The `datanames` argument can be a MATLAB string for a single data series. For multiple data series names, it must be a cell array of strings.

Look at two examples, one with a single data series and a second with two. The first example sets the data series name to the specified name `First`:

```

dates = (today:today+100)';
data = exp(randn(1, 101))';
fts1 = fints(dates, data, 'First')

```

fts1 =

```

desc: (none)
freq: Unknown (0)

```

```
'dates: (101)'      'First: (101)'  
'12-Jul-1999'      [          0.4615]  
'13-Jul-1999'      [          1.1640]  
'14-Jul-1999'      [          0.7140]  
'15-Jul-1999'      [          2.6400]  
'16-Jul-1999'      [          0.8983]  
'17-Jul-1999'      [          2.7552]  
'18-Jul-1999'      [          0.6217]  
'19-Jul-1999'      [          1.0714]  
'20-Jul-1999'      [          1.4897]  
'21-Jul-1999'      [          3.0536]  
'22-Jul-1999'      [          1.8598]  
'23-Jul-1999'      [          0.7500]  
'24-Jul-1999'      [          0.2537]  
'25-Jul-1999'      [          0.5037]  
'26-Jul-1999'      [          1.3933]  
'27-Jul-1999'      [          0.3687]...
```

The second example provides two data series named `First` and `Second`:

```
dates = (today:today+100)';  
data_series1 = exp(randn(1, 101))';  
data_series2 = exp(randn(1, 101))';  
data = [data_series1 data_series2];  
fts2 = fints(dates, data, {'First', 'Second'})
```

```
fts2 =  
desc: (none)  
freq: Unknown (0)  
  
'dates: (101)'      'First: (101)'      'Second: (101)'  
'12-Jul-1999'      [          1.2305]      [          0.7396]  
'13-Jul-1999'      [          1.2473]      [          2.6038]  
'14-Jul-1999'      [          0.3657]      [          0.5866]  
'15-Jul-1999'      [          0.6357]      [          0.4061]  
'16-Jul-1999'      [          4.0530]      [          0.4096]  
'17-Jul-1999'      [          0.6300]      [          1.3214]  
'18-Jul-1999'      [          1.0333]      [          0.4744]  
'19-Jul-1999'      [          2.2228]      [          4.9702]  
'20-Jul-1999'      [          2.4518]      [          1.7758]  
'21-Jul-1999'      [          1.1479]      [          1.3780]  
'22-Jul-1999'      [          0.1981]      [          0.8595]  
'23-Jul-1999'      [          0.1927]      [          1.3713]  
'24-Jul-1999'      [          1.5353]      [          3.8332]  
'25-Jul-1999'      [          0.4784]      [          0.1067]
```



```
'26-Jul-1999' [ 1.7593] [ 3.6434]
'27-Jul-1999' [ 0.2505] [ 0.6849]
'28-Jul-1999' [ 1.5845] [ 1.0025]...
```

Note Data series names must be valid MATLAB variable names. The only allowed nonalphanumeric character is the underscore (_) character.

Because `freq` for `fts2` has not been explicitly indicated, the frequency indicator for `fts2` is set to `Unknown`. Set the frequency indicator field `freq` before you attempt any operations on the object. You will not be able to use the object until the frequency indicator field is set to a valid indicator.

Frequency Indicator Input

```
fts = fints(dates, data, datanames, freq)
```

With the fourth syntax you can set the frequency indicator field when you create the financial time series object. The frequency indicator field `freq` is set as the fourth input argument. You will not be able to use the financial time series object until `freq` is set to a valid indicator. Valid frequency indicators are

```
UNKNOWN, Unknown, unknown, U, u,0
DAILY, Daily, daily, D, d,1
WEEKLY, Weekly, weekly, W, w,2
MONTHLY, Monthly, monthly, M, m,3
QUARTERLY, Quarterly, quarterly, Q, q,4
SEMIANNUAL, Semiannual, semiannual, S, s,5
ANNUAL, Annual, annual, A, a,6
```

The previous example contained sets of daily data. The `freq` field displayed as `Unknown (0)` because the frequency indicator was not explicitly set. The command

```
fts = fints(dates, data, {'First', 'Second'}, 1);
```

sets the `freq` indicator to `Daily(1)` when creating the financial time series object:

```
fts =
  desc: (none)
  freq: Daily (1)
  dates: (101)'   'First: (101)'   'Second: (101)'
```

```
'12-Jul-1999' [ 1.2305] [ 0.7396]
'13-Jul-1999' [ 1.2473] [ 2.6038]
'14-Jul-1999' [ 0.3657] [ 0.5866]
'15-Jul-1999' [ 0.6357] [ 0.4061]
'16-Jul-1999' [ 4.0530] [ 0.4096]
'17-Jul-1999' [ 0.6300] [ 1.3214]
'18-Jul-1999' [ 1.0333] [ 0.4744]...
```

When you create the object using this syntax, you can use the other valid frequency indicators for a particular frequency. For a daily data set you can use `DAILY`, `Daily`, `daily`, `D`, or `d`. Similarly, with the other frequencies, you can use the valid string indicators or their numeric counterparts.

Description Field Input

```
fts = fints(dates, data, datanames, freq, desc)
```

With the fifth syntax, you can explicitly set the description field as the fifth input argument. The description can be anything you want. It is not used in any operations performed on the object.

This example sets the `desc` field to `'Test TS'`.

```
dates = (today:today+100)';
data_series1 = exp(randn(1, 101));
data_series2 = exp(randn(1, 101));
data = [data_series1 data_series2];
fts = fints(dates, data, {'First', 'Second'}, 1, 'Test TS')

fts =
  desc: Test TS
  freq: Daily (1)

  'dates: (101)'   'First: (101)'   'Second: (101)'
  '12-Jul-1999' [ 0.5428] [ 1.2491]
  '13-Jul-1999' [ 0.6649] [ 6.4969]
  '14-Jul-1999' [ 0.2428] [ 1.1163]
  '15-Jul-1999' [ 1.2550] [ 0.6628]
  '16-Jul-1999' [ 1.2312] [ 1.6674]
  '17-Jul-1999' [ 0.4869] [ 0.3015]
  '18-Jul-1999' [ 2.1335] [ 0.9081]...
```

Now the description field is filled with the specified string `'Test TS'` when the constructor is called.

Transforming a Text File

The function `ascii2fts` creates a financial time series object from a text (ASCII) data file provided that the data file conforms to a general format. The general format of the text data file is as follows:

- Can contain header text lines.
- Can contain column header information. The column header information must immediately precede the data series columns unless the `skiprows` argument (see below) is specified.
- Leftmost column must be the date column.
- Dates must be in a valid date string format.
 - `'ddmmyy'` or `'ddmmyyyy'`
 - `'mm/dd/yy'` or `'mm/dd/yyyy'`
 - `'dd-mmm-yy'` or `'dd-mmm-yyyy'`
 - `'mmm.dd,yy'` or `'mmm.dd,yyyy'`
- Each column must be separated either by spaces or a tab.

Several example text data files are included with the toolbox. These files are in the `ftsdata` subdirectory within the directory `matlabroot/toolbox/finance`.

The syntax of the function

```
fts = ascii2fts(filename, descrow, colheadrow, skiprows);
```

takes in the data file name (`filename`), the row number where the text for the description field is (`descrow`), the row number of the column header information (`colheadrow`), and the row numbers of rows to be skipped (`skiprows`). For example, rows need to be skipped when there are intervening rows between the column head row and the start of the time series data.

Look at the beginning of the ASCII file `disney.dat` in the `ftsdata` subdirectory:

```
Walt Disney Company (DIS)
Daily prices (3/29/96 to 3/29/99)
DATE      OPEN      HIGH      LOW      CLOSE      VOLUME
3/29/99   33.0625   33.188    32.75    33.063     6320500
3/26/99   33.3125   33.375    32.75    32.938     5552800
```

```
3/25/99 33.5      33.625  32.875  33.375  7936000
3/24/99 33.0625  33.25   32.625  33.188  6025400...
```

The command line

```
disfts = ascii2fts('disney.dat', 1, 3, 2)
```

uses `disney.dat` to create time series object `disfts`. This example

- Reads the text data file `disney.dat`
- Uses the first line in the file as the content of the description field
- Skips the second line
- Parses the third line in the file for column header (or data series names)
- Parses the rest of the file for the date vector and the data series values

The resulting financial time series object looks like this.

```
disfts =
desc: Walt Disney Company (DIS)
freq: Unknown (0)

'dates: (782)'  'OPEN: (782)'  'HIGH: (782)'  'LOW: (782)'
'29-Mar-1996'  [ 21.1938]    [ 21.6250]    [ 21.2920]
'01-Apr-1996'  [ 21.1120]    [ 21.6250]    [ 21.4170]
'02-Apr-1996'  [ 21.3165]    [ 21.8750]    [ 21.6670]
'03-Apr-1996'  [ 21.4802]    [ 21.8750]    [ 21.7500]
'04-Apr-1996'  [ 21.4393]    [ 21.8750]    [ 21.5000]
'05-Apr-1996'  [      NaN]    [      NaN]    [      NaN]
'09-Apr-1996'  [ 21.1529]    [ 21.5420]    [ 21.2080]
'10-Apr-1996'  [ 20.7387]    [ 21.1670]    [ 20.2500]
'11-Apr-1996'  [ 20.0829]    [ 20.5000]    [ 20.0420]
'12-Apr-1996'  [ 19.9189]    [ 20.5830]    [ 20.0830]
'15-Apr-1996'  [ 20.2878]    [ 20.7920]    [ 20.3750]
'16-Apr-1996'  [ 20.3698]    [ 20.9170]    [ 20.1670]
'17-Apr-1996'  [ 20.4927]    [ 20.9170]    [ 20.7080]
'18-Apr-1996'  [ 20.4927]    [ 21.0420]    [ 20.7920]
```

There are 782 data points in this object. Only the first few lines are shown here. Also, this object has two other data series, the `CLOSE` and `VOLUME` data series, that are not shown here. Note that in creating the financial time series object, `ascii2fts` sorts the data into ascending chronological order.

The frequency indicator field, `freq`, is set to 0 for Unknown frequency. You can manually reset it to the appropriate frequency using structure syntax `disfts.freq = 1` for Daily frequency.

With a slightly different syntax, the function `ascii2fts` can create a financial time series object when time-of-day data is present in the ASCII file. The new syntax has the form

```
fts = ascii2fts(filename, timedata, descrow, colheadrow,  
skiprows);
```

Set `timedata` to 'T' when time-of-day data is present and to 'NT' when there is no time data. For an example using this function with time-of-day data, see the reference page for `ascii2fts`.

Visualizing Financial Time Series Objects

In this section...

“Introduction” on page 11-16

“Using `chartfts`” on page 11-16

“Zoom Tool” on page 11-19

“Combine Axes Tool” on page 11-22

Introduction

Financial Toolbox software contains the function `chartfts`, which provides a visual representation of a financial time series object. `chartfts` is an interactive charting and graphing utility for financial time series objects. With this function, you can observe time series values on the entire range of dates covered by the time series.

Note Interactive charting is also available from the **Graphs** menu of the graphical user interface. See “Interactive Chart” on page 14-16 for additional information.

Using `chartfts`

`chartfts` requires a single input argument, `tsobj`, where `tsobj` is the name of the financial time series object you want to explore. Most equity financial time series objects contain four price series, such as opening, closing, highest, and lowest prices, plus an additional series containing the volume traded. However, `chartfts` is not limited to a time series of equity prices and volume traded. It can be used to display any time series data you may have.

To illustrate the use of `chartfts`, use the equity price and volume traded data for the Walt Disney Corporation (NYSE: DIS) provided in the file `disney.mat`:

```
load disney.mat
```

```
whos
```

Name	Size	Bytes	Class
dis	782x5	39290	fints object

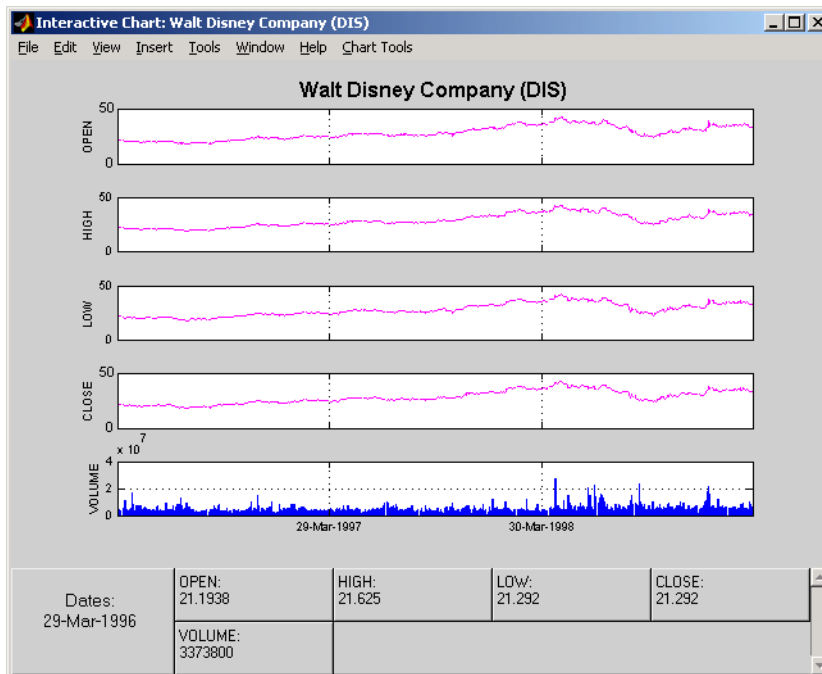
```

dis_CLOSE      782x1          6256  double array
dis_HIGH       782x1          6256  double array
dis_LOW        782x1          6256  double array
dis_OPEN       782x1          6256  double array
dis_VOLUME     782x1          6256  double array
dis_nv         782x4          32930  fints object
q_dis          13x4           2196  fints object

```

For charting purposes look only at the objects `dis` (daily equity data including volume traded) and `dis_nv` (daily data without volume traded). Both objects contain the series OPEN, HIGH, LOW, and CLOSE, but only `dis` contains the additional VOLUME series.

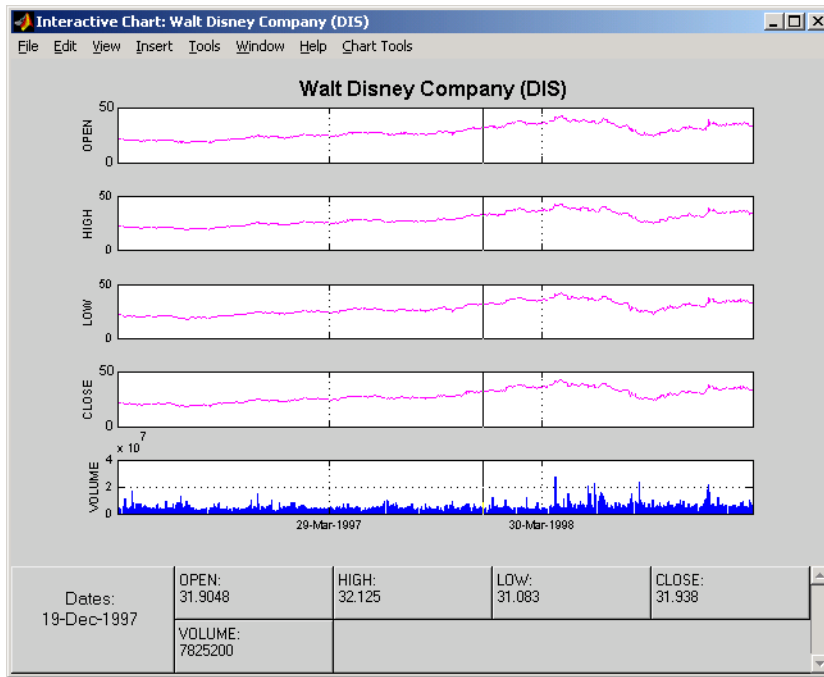
Use `chartfts(dis)` to observe the values.



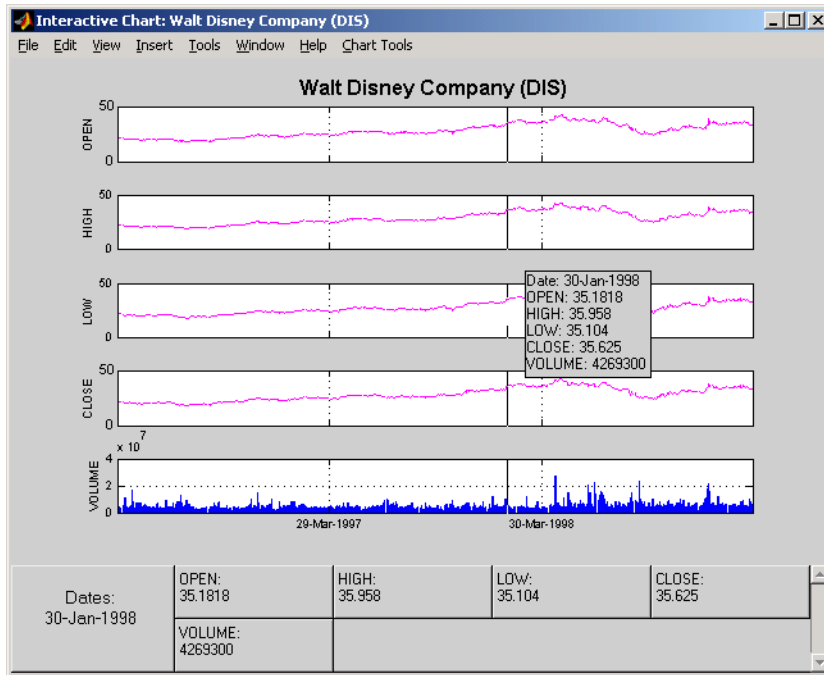
The chart contains five plots, each representing one of the series in the time series object. Boxes indicate the value of each individual plot. The date box is always on the left. The number of data boxes on the right depends upon the number of data series in the time series object, five in this case. The order in which these boxes are arranged (left to right) matches the plots from top to bottom. With more than eight data series in the object, the

scroll bar on the right is activated so that additional data from the other series can be brought into view.

Slide the mouse cursor over the chart. A vertical bar appears across all plots. This bar selects the set of data shown in the boxes below. Move this bar horizontally and the data changes accordingly.

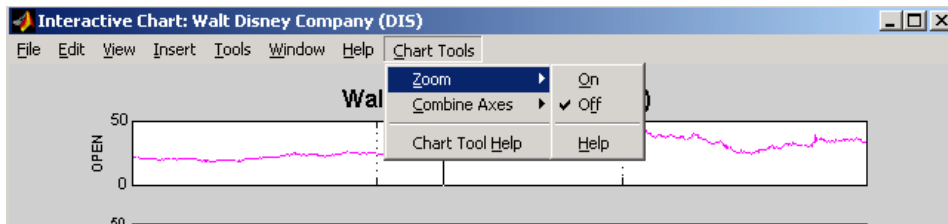


Click the plot. A small information box displays the data at the point where you click the mouse button.



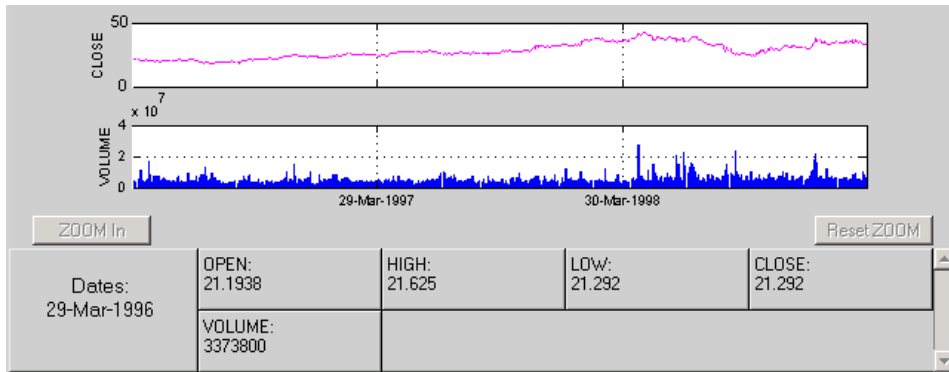
Zoom Tool

The zoom feature of `chartfts` enables a more detailed look at the data during a selected time frame. The Zoom tool is found under the **Chart Tools** menu.

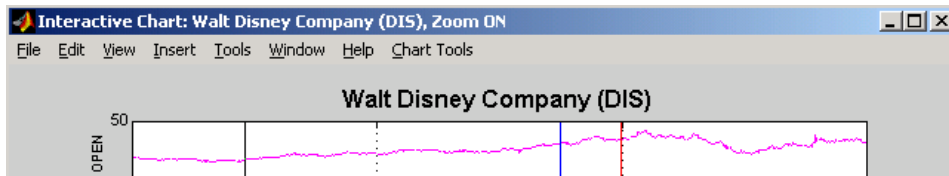


Note Due to the specialized nature of this feature, do not use the MATLAB `zoom` command or **Zoom In** and **Zoom Out** from the **Tools** menu.

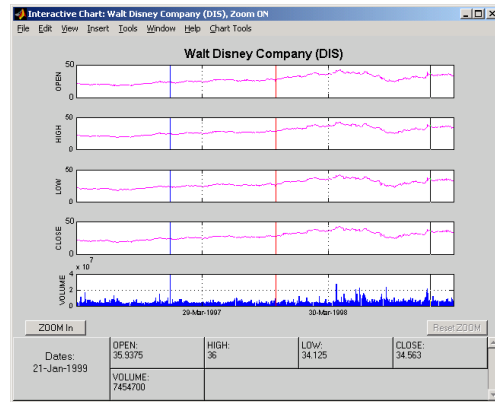
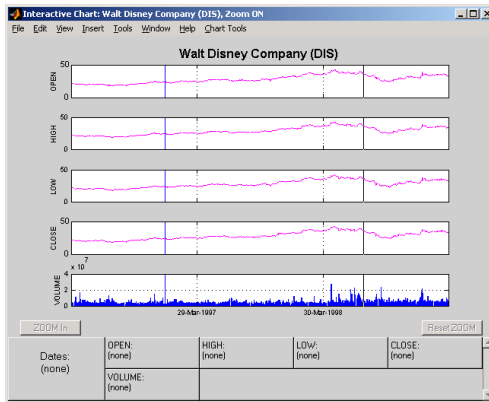
When the feature is turned on, you will see two inactive buttons (**ZOOM In** and **Reset ZOOM**) above the boxes. The buttons become active later after certain actions have been performed.



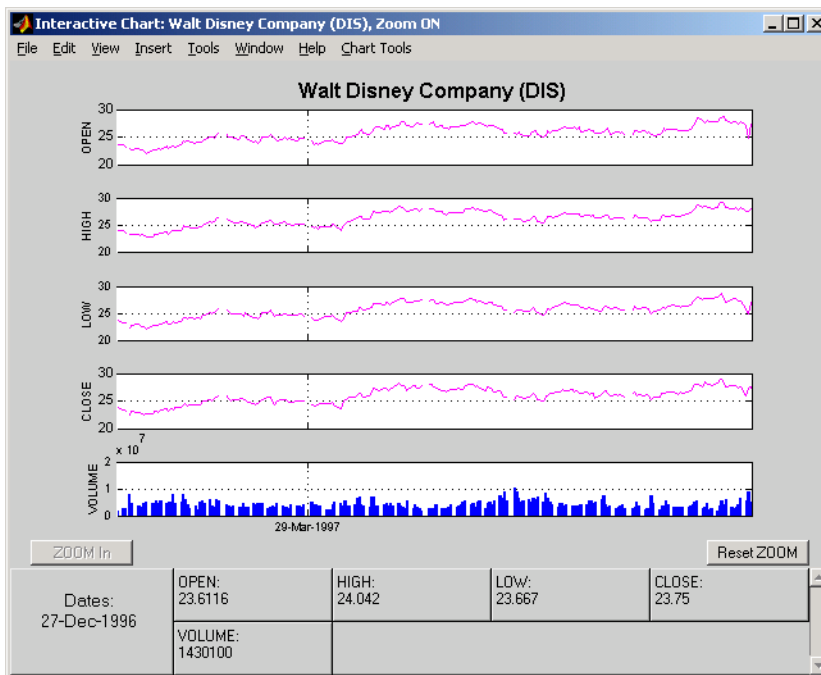
The window title bar displays the status of the chart tool that you are using. With the Zoom tool turned on, you see **Zoom ON** in the title bar in addition to the name of the time series you are working with. When the tool is off, no status is displayed.



To zoom into the chart, you need to define the starting and ending dates. Define the starting date by moving the cursor over the chart until the desired date appears at the bottom-left box and click the mouse button. A blue vertical line indicates the starting date you have selected. Next, again move the cursor over the chart until the desired ending date appears in the box and click the mouse once again. This time, a red vertical line appears and the **ZOOM In** button is activated.



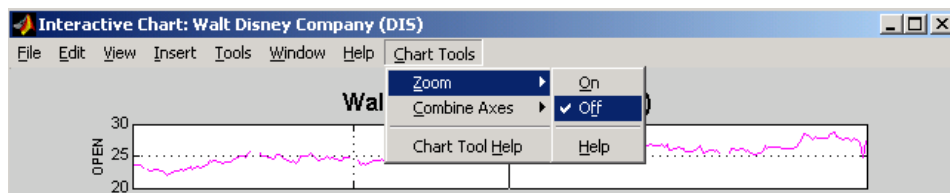
To zoom into the chart, click the **ZOOM In** button.



The chart is zoomed in. Note that the **Reset ZOOM** button now becomes active while the **ZOOM In** button becomes inactive again. To return the chart to its original state (not

zoomed), click the **Reset ZOOM** button. To zoom into the chart even further, repeat the steps above for zooming into the chart.

Turn the Zoom tool off by going back to the **Chart Tools** menu and choosing **Zoom Off**.



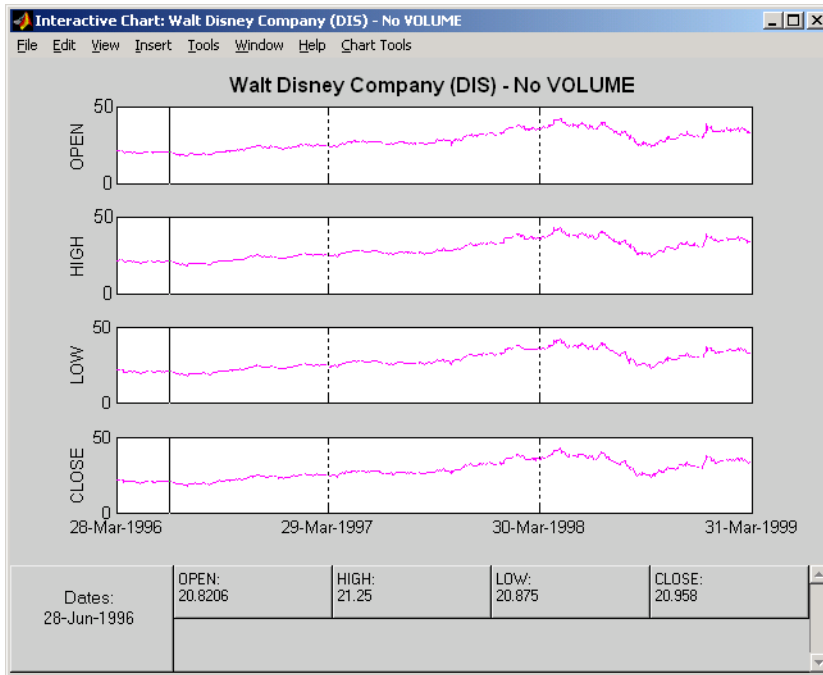
With the tool turned off, the chart stays at the last state that it was in. If you turn it off when the chart is zoomed in, the chart stays zoomed in. If you reset the zoom before turning it off, the chart becomes the original (not zoomed).

Combine Axes Tool

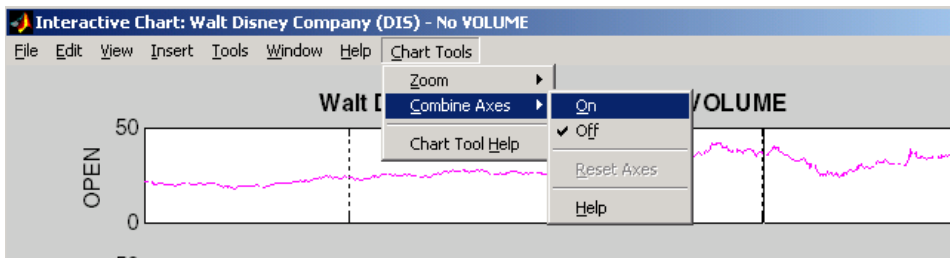
The Combine Axes tool allows you to combine all axes or specific axes into one. With axes combined, you can visually spot any trends that can occur among the data series in a financial time series object.

To illustrate this tool, use `dis_nv`, the financial time series object that does not contain volume traded data:

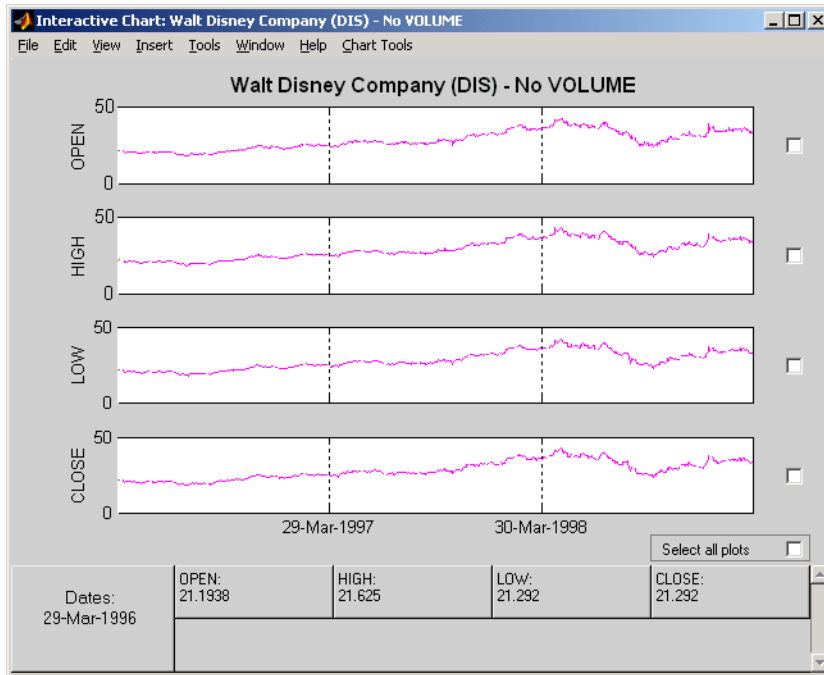
```
chartfts(dis_nv)
```



To combine axes, choose the **Chart Tools** menu, followed by **Combine Axes** and **On**.

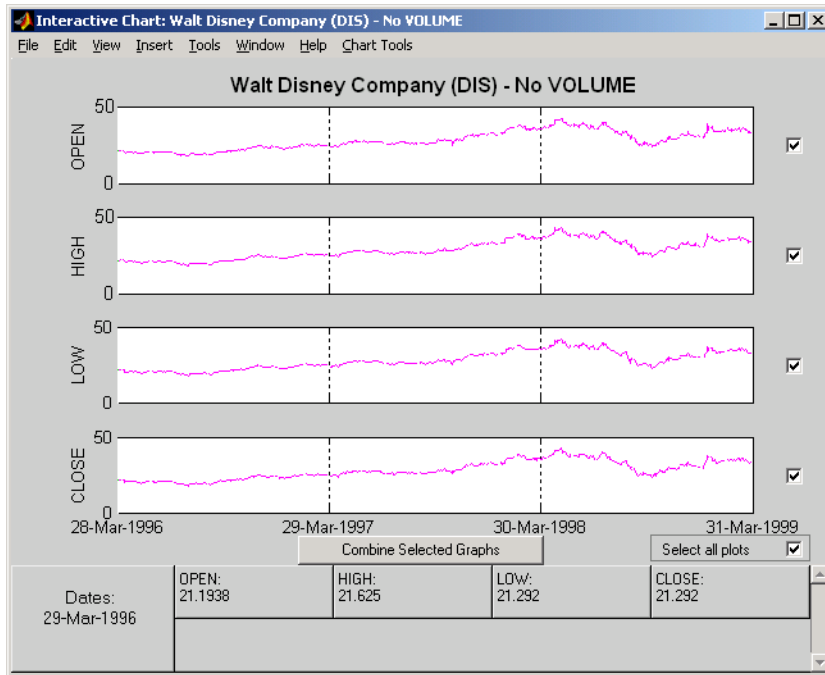


When the Combine Axes tool is on, check boxes appear beside each individual plot. An additional check box enables the combination of all plots.

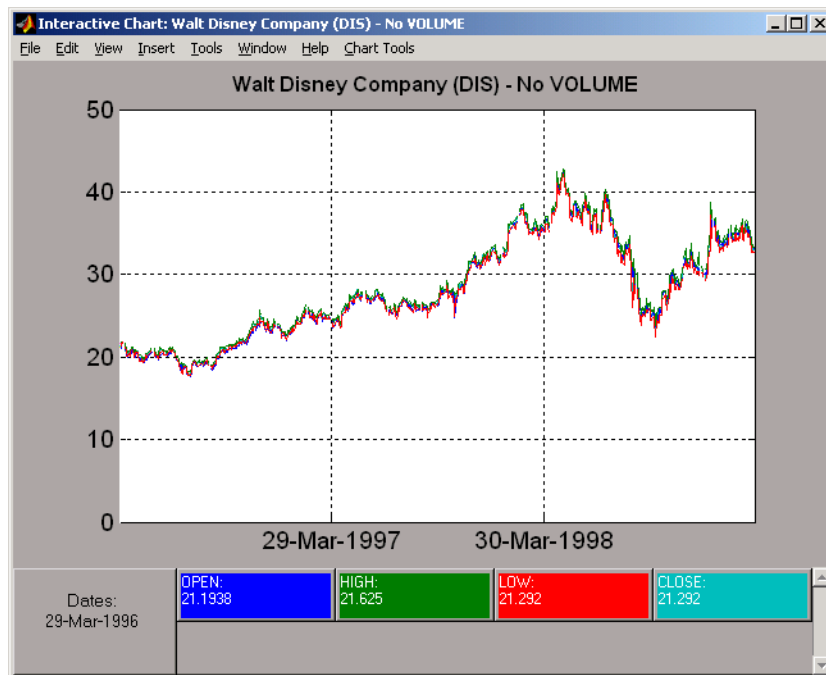


Combining All Axes

To combine all plots, select the **Select all plots** check box.



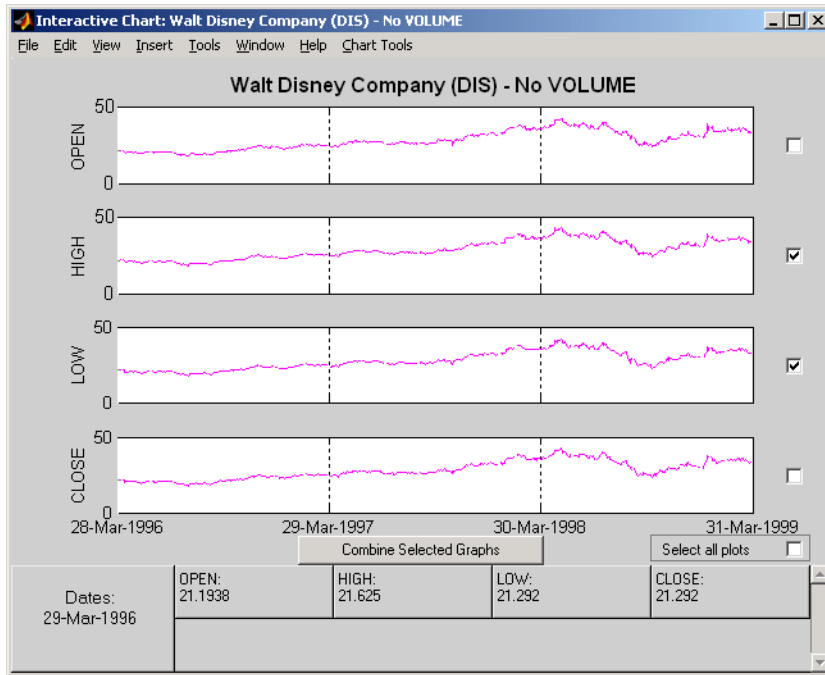
Now click the **Combine Selected Graphs** button to combine the chosen plots. In this case, all plots are combined.



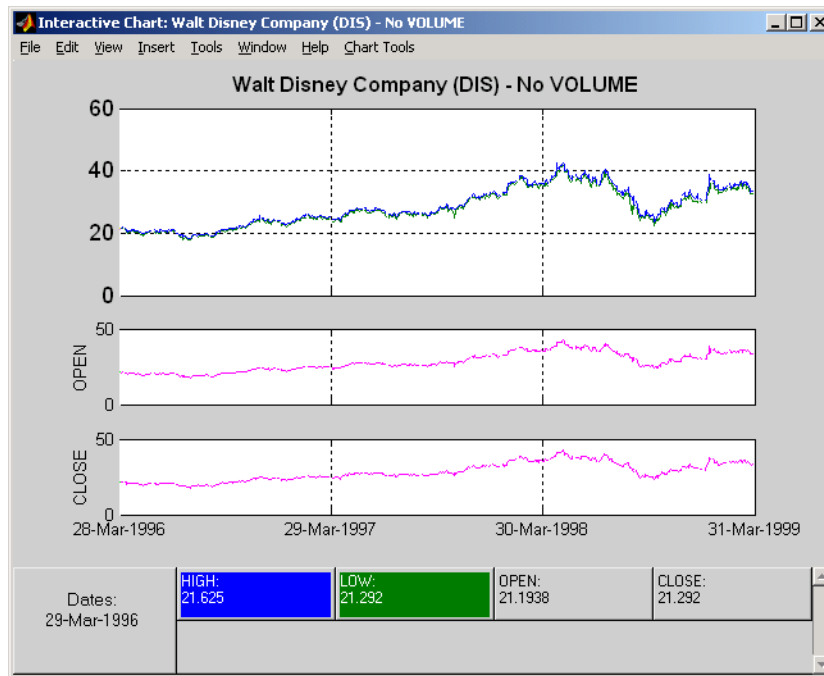
The combined plots have a single plot axis with all data series traced. The background of each data box has changed to the color corresponding to the color of the trace that represents the data series. After the axes are combined, the tool is turned off.

Combining Selected Axes

You can choose any combination of the available axes to combine. For example, combine the HIGH and LOW price series of the Disney time series. Click the check boxes next to the corresponding plots. The **Combine Selected Graphs** button appears and is active.



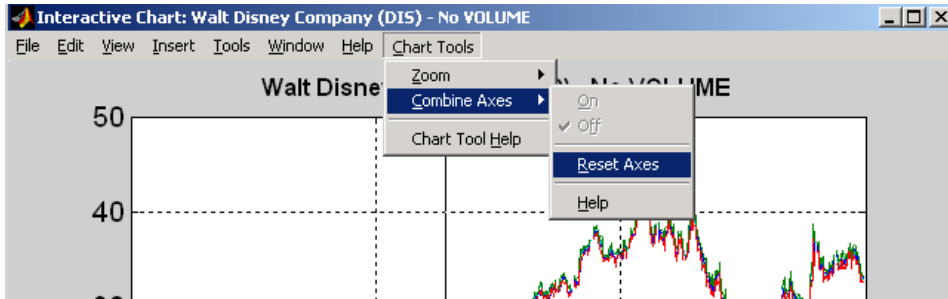
Click the **Combine Selected Graphs** button. The chart with the combined plots looks like the next figure.



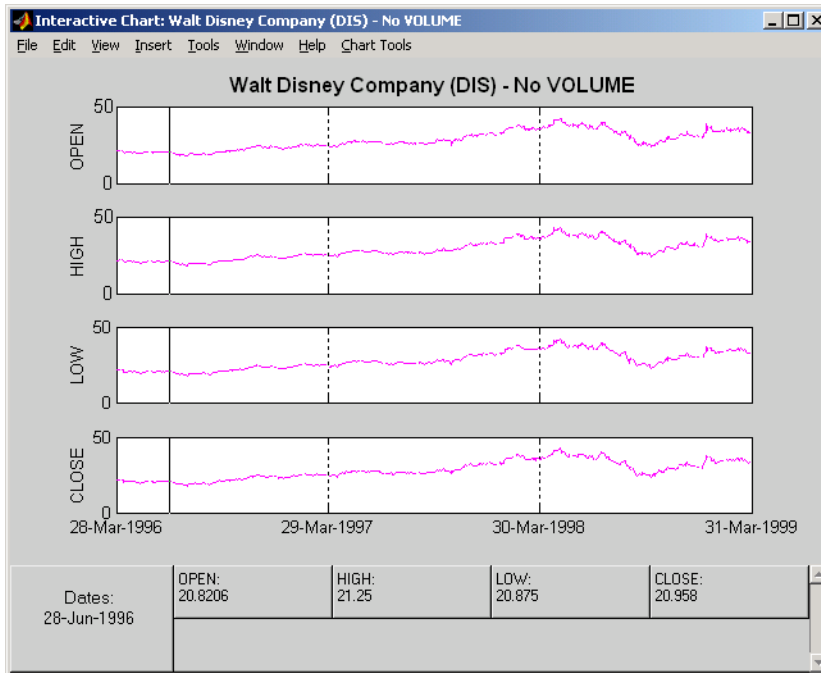
The plot with the combined axes is located at the top of the chart while the remaining plots follow it. The data boxes have also been changed. The boxes that correspond to the combined axes are relocated to the beginning, and the background colors are set to the color of the respective traces. The data boxes for the remaining axes retain their original formats.

Resetting Axes

If you have altered the chart by combining axes, you must reset the axes before you can visualize additional combinations. Reset the axes with the **Reset Axes** menu item under **Chart Tools > Combine Axes**. Note that now the **On** and **Off** features are turned off.



With axes reset, the interactive chart appears in its original format, and you can proceed with additional axes combinations.



Using Financial Time Series

- “Introduction” on page 12-2
- “Working with Financial Time Series Objects” on page 12-3
- “Using Time Series to Predict Equity Return” on page 12-22

Introduction

This section discusses how to manipulate and analyze financial time series data. The major topics discussed include

- “Financial Time Series Object Structure” on page 12-3
- “Data Extraction” on page 12-4
- “Object-to-Matrix Conversion” on page 12-6
- “Indexing a Financial Time Series Object” on page 12-7
- “Financial Time Series Operations” on page 12-14
- “Data Transformation and Frequency Conversion” on page 12-17

Much of this information is summarized in the “Using Time Series to Predict Equity Return” on page 12-22.

Working with Financial Time Series Objects

In this section...

“Introduction” on page 12-3

“Financial Time Series Object Structure” on page 12-3

“Data Extraction” on page 12-4

“Object-to-Matrix Conversion” on page 12-6

“Indexing a Financial Time Series Object” on page 12-7

“Financial Time Series Operations” on page 12-14

“Data Transformation and Frequency Conversion” on page 12-17

Introduction

A financial time series object is designed to be used as if it were a MATLAB structure. (See the MATLAB documentation for a description of MATLAB structures or how to use MATLAB in general.)

This part of the tutorial assumes that you know how to use MATLAB and are familiar with MATLAB structures. The terminology is similar to that of a MATLAB structure. The financial time series object term *component* is interchangeable with the MATLAB structure term *field*.

Financial Time Series Object Structure

A financial time series object always contains three component names: **desc** (description field), **freq** (frequency indicator field), and **dates** (date vector). If you build the object using the constructor **fints**, the default value for the description field is a blank string (' '). If you build the object from a text data file using **ascii2fts**, the default is the name of the text data file. The default for the frequency indicator field is **0** (Unknown frequency). Objects created from operations can default the setting to **0**. For example, if you decide to pick out values selectively from an object, the frequency of the new object might not be the same as that of the object from which it came.

The date vector **dates** does not have a default set of values. When you create an object, you have to supply the date vector. You can change the date vector afterward but, at object creation time, you must provide a set of dates.

The final component of a financial time series object is one or more data series vectors. If you do not supply a name for the data series, the default name is `series1`. If you have multiple data series in an object and do not supply the names, the default is the name series followed by a number, for example, `series1`, `series2`, and `series3`.

Data Extraction

Here is an exercise on how to extract data from a financial time series object. As mentioned before, you can think of the object as a MATLAB structure. Highlight each line in the exercise in the MATLAB Help browser, press the right mouse button, and select **Evaluate Selection** to execute it.

To begin, create a financial time series object called `myfts`:

```
dates = (datenum('05/11/99'):datenum('05/11/99')+100)';
data_series1 = exp(randn(1, 101))';
data_series2 = exp(randn(1, 101))';
data = [data_series1 data_series2];
myfts = fints(dates, data);
```

The `myfts` object looks like this:

```
myfts =

  desc: (none)
  freq: Unknown (0)

  'dates: (101)'      'series1: (101)'      'series2: (101)'
  '11-May-1999'      [      2.8108]         [      0.9323]
  '12-May-1999'      [      0.2454]         [      0.5608]
  '13-May-1999'      [      0.3568]         [      1.5989]
  '14-May-1999'      [      0.5255]         [      3.6682]
  '15-May-1999'      [      1.1862]         [      5.1284]
  '16-May-1999'      [      3.8376]         [      0.4952]
  '17-May-1999'      [      6.9329]         [      2.2417]
  '18-May-1999'      [      2.0987]         [      0.3579]
  '19-May-1999'      [      2.2524]         [      3.6492]
  '20-May-1999'      [      0.8669]         [      1.0150]
  '21-May-1999'      [      0.9050]         [      1.2445]
  '22-May-1999'      [      0.4493]         [      5.5466]
  '23-May-1999'      [      1.6376]         [      0.1251]
  '24-May-1999'      [      3.4472]         [      1.1195]
  '25-May-1999'      [      3.6545]         [      0.3374]...
```


There are more dates in the object; only the first few lines are shown here.

Note The actual data in your `series1` and `series2` will differ from the above because of the use of random numbers.

Now create another object with only the values for `series2`:

```
srs2 = myfts.series2
```

```
srs2 =
```

```

desc: (none)
freq: Unknown (0)

'dates: (101)'      'series2: (101)'
'11-May-1999'      [          0.9323]
'12-May-1999'      [          0.5608]
'13-May-1999'      [          1.5989]
'14-May-1999'      [          3.6682]
'15-May-1999'      [          5.1284]
'16-May-1999'      [          0.4952]
'17-May-1999'      [          2.2417]
'18-May-1999'      [          0.3579]
'19-May-1999'      [          3.6492]
'20-May-1999'      [          1.0150]
'21-May-1999'      [          1.2445]
'22-May-1999'      [          5.5466]
'23-May-1999'      [          0.1251]
'24-May-1999'      [          1.1195]
'25-May-1999'      [          0.3374]...
```

The new object `srs2` contains all the dates in `myfts`, but the only data series is `series2`. The name of the data series retains its name from the original object, `myfts`.

Note The output from referencing a data series field or indexing a financial time series object is always another financial time series object. The exceptions are referencing the description, frequency indicator, and dates fields, and indexing into the dates field.

Object-to-Matrix Conversion

The function `fts2mat` extracts the dates and/or the data series values from an object and places them into a vector or a matrix. The default behavior extracts just the values into a vector or a matrix. Look at the next example:

```
srs2_vec = fts2mat(myfts.series2)
```

```
srs2_vec =  
    0.9323  
    0.5608  
    1.5989  
    3.6682  
    5.1284  
    0.4952  
    2.2417  
    0.3579  
    3.6492  
    1.0150  
    1.2445  
    5.5466  
    0.1251  
    1.1195  
    0.3374...
```

If you want to include the dates in the output matrix, provide a second input argument and set it to 1. This results in a matrix whose first column is a vector of serial date numbers:

```
format long g
```

```
srs2_mtx = fts2mat(myfts.series2, 1)
```

```
srs2_mtx =  
  
    730251    0.932251754559576  
    730252    0.560845677519876  
    730253    1.59888712183914  
    730254    3.6681500883527  
    730255    5.12842215360269  
    730256    0.49519254119977  
    730257    2.24174134286213  
    730258    0.357918065917634
```

```

730259      3.64915665824198
730260      1.01504236943148
730261      1.24446420606078
730262      5.54661849025711
730263      0.12507959735904
730264      1.11953883096805
730265      0.337398214166607

```

The vector `srs2_vec` contains just `series2` values. The matrix `srs2_mtx` contains dates in the first column and the values of the `series2` data series in the second. Dates in the first column are in serial date format. Serial date format is a representation of the date string format (for example, serial date = 1 is equivalent to 01-Jan-0000). (The serial date vector can include time-of-day information.)

The long `g` display format displays the numbers without exponentiation. (To revert to the default display format, use `format short`. (See the `format` command in the MATLAB documentation for a description of MATLAB display formats.) Remember that both the vector and the matrix have 101 rows of data as in the original object `myfts` but are shown truncated here.

Indexing a Financial Time Series Object

You can also index into the object as with any other MATLAB variable or structure. A financial time series object lets you use a date string, a cell array of date strings, a date string range, or normal integer indexing. *You cannot, however, index into the object using serial dates.* If you have serial dates, you must first use the MATLAB `datestr` command to convert them into date strings.

When indexing by date string, note that

- Each date string must contain the day, month, and year. Valid formats are
 - 'ddmmyy hh:mm' or 'ddmmyyyy hh:mm'
 - 'mm/dd/yy hh:mm' or 'mm/dd/yyyy hh:mm'
 - 'dd-mmm-yy hh:mm' or 'dd-mmm-yyyy hh:mm'
 - 'mmm.dd,yy hh:mm' or 'mmm.dd,yyyy hh:mm'
- All data falls at the end of the indicated time period, that is, weekly data falls on Fridays, monthly data falls on the end of each month, and so on, whenever the data has gone through a frequency conversion.

Indexing with Date Strings

With date string indexing you get the values in a financial time series object for a specific date using a date string as the index into the object. Similarly, if you want values for multiple dates in the object, you can put those date strings into a cell array and use the cell array as the index to the object. Here are some examples.

This example extracts all values for May 11, 1999 from `myfts`:

```
format short
myfts('05/11/99')

ans =

    desc: (none)
    freq: Unknown (0)

    'dates: (1)'    'series1: (1)'    'series2: (1)'
    '11-May-1999'  [      2.8108]    [      0.9323]
```

The next example extracts only `series2` values for May 11, 1999 from `myfts`:

```
myfts.series2('05/11/99')

ans =

    desc: (none)
    freq: Unknown (0)

    'dates: (1)'    'series2: (1)'
    '11-May-1999'  [      0.9323]
```

The third example extracts all values for three different dates:

```
myfts({'05/11/99', '05/21/99', '05/31/99'})

ans =

    desc: (none)
    freq: Unknown (0)

    'dates: (3)'    'series1: (3)'    'series2: (3)'
    '11-May-1999'  [      2.8108]    [      0.9323]
    '21-May-1999'  [      0.9050]    [      1.2445]
    '31-May-1999'  [      1.4266]    [      0.6470]
```

The next example extracts only `series2` values for the same three dates:

```
myfts.series2({'05/11/99', '05/21/99', '05/31/99'})
```

```
ans =
```

```
desc: (none)
freq: Unknown (0)

'dates: (3)'      'series2: (3)'
```

'11-May-1999'	[0.9323]
'21-May-1999'	[1.2445]
'31-May-1999'	[0.6470]

Indexing with Date String Range

A financial time series is unique because it allows you to index into the object using a date string range. A date string range consists of two date strings separated by two colons (::). In MATLAB this separator is called the double-colon operator. An example of a MATLAB date string range is `'05/11/99::05/31/99'`. The operator gives you all data points available between those dates, including the start and end dates.

Here are some date string range examples:

```
myfts ('05/11/99::05/15/99')
```

```
ans =
```

```
desc: (none)
freq: Unknown (0)

'dates: (5)'      'series1: (5)'      'series2: (5)'
```

'11-May-1999'	[2.8108]	[0.9323]
'12-May-1999'	[0.2454]	[0.5608]
'13-May-1999'	[0.3568]	[1.5989]
'14-May-1999'	[0.5255]	[3.6682]
'15-May-1999'	[1.1862]	[5.1284]

```
myfts.series2('05/11/99::05/15/99')
```

```
ans =
```

```
desc: (none)
freq: Unknown (0)
```

```
'dates: (5)'      'series2: (5)'  
'11-May-1999' [      0.9323]  
'12-May-1999' [      0.5608]  
'13-May-1999' [      1.5989]  
'14-May-1999' [      3.6682]  
'15-May-1999' [      5.1284]
```

As with any other MATLAB variable or structure, you can assign the output to another object variable:

```
nfts = myfts.series2('05/11/99:05/20/99');
```

`nfts` is the same as `ans` in the second example.

If one of the dates does not exist in the object, an error message indicates that one or both date indexes are out of the range of the available dates in the object. You can either display the contents of the object or use the command `ftsbound` to determine the first and last dates in the object.

Indexing with Integers

Integer indexing is the normal form of indexing in MATLAB. Indexing starts at 1 (not 0); index = 1 corresponds to the first element, index = 2 to the second element, index = 3 to the third element, and so on. Here are some examples with and without data series reference.

Get the first item in `series2`:

```
myfts.series2(1)  
  
ans =  
  
desc: (none)  
freq: Unknown (0)  
  
'dates: (1)'      'series2: (1)'  
'11-May-1999' [      0.9323]
```

Get the first, third, and fifth items in `series2`:

```
myfts.series2([1, 3, 5])  
  
ans =
```

```

desc: (none)
freq: Unknown (0)

'dates: (3)'      'series2: (3)'
'11-May-1999'    [      0.9323]
'13-May-1999'    [      1.5989]
'15-May-1999'    [      5.1284]

```

Get items 16 through 20 in **series2**:

```
myfts.series2(16:20)
```

```
ans =
```

```

desc: (none)
freq: Unknown (0)

'dates: (5)'      'series2: (5)'
'26-May-1999'    [      0.2105]
'27-May-1999'    [      1.8916]
'28-May-1999'    [      0.6673]
'29-May-1999'    [      0.6681]
'30-May-1999'    [      1.0877]

```

Get items 16 through 20 in the financial time series object **myfts**:

```
myfts(16:20)
```

```
ans =
```

```

desc: (none)
freq: Unknown (0)

'dates: (5)'      'series1: (5)'      'series2: (5)'
'26-May-1999'    [      0.7571]      [      0.2105]
'27-May-1999'    [      1.2425]      [      1.8916]
'28-May-1999'    [      1.8790]      [      0.6673]
'29-May-1999'    [      0.5778]      [      0.6681]
'30-May-1999'    [      1.2581]      [      1.0877]

```

Get the last item in **myfts**:

```
myfts(end)
```

```
ans =
```

```
desc: (none)
freq: Unknown (0)

'dates: (1)'      'series1: (1)'      'series2: (1)'
'19-Aug-1999'    [      1.4692]      [      3.4238]
```

This example uses the MATLAB special variable `end`, which points to the last element of the object when used as an index. The example returns an object whose contents are the values in the object `myfts` on the last date entry.

Indexing When Time-of-Day Data Is Present

Both integer and date string indexing are permitted when time-of-day information is present in the financial time series object. You can index into the object with both date and time specifications, but not with time of day alone. To show how indexing works with time-of-day data present, create a financial time series object called `timeday` containing a time specification:

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
         '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates,1),1), ...
                       times]);
timeday = fints(dates_times, (1:6)', {'Data1'}, 1, 'My first FINTS')

timeday =

desc: My first FINTS
freq: Daily (1)

'dates: (6)'      'times: (6)'      'Data1: (6)'
'01-Jan-2001'    '11:00'          [      1]
'    "    '    '12:00'          [      2]
'02-Jan-2001'    '11:00'          [      3]
'    "    '    '12:00'          [      4]
'03-Jan-2001'    '11:00'          [      5]
'    "    '    '12:00'          [      6]
```

Use integer indexing to extract the second and third data items from `timeday`:

```
timeday(2:3)

ans =
```



```

desc: My first FINTS
freq: Daily (1)

'dates: (2)'   'times: (2)'   'Data1: (2)'
'01-Jan-2001' '12:00'   [          2]
'02-Jan-2001' '11:00'   [          3]

```

For date string indexing, enclose the date and time string in one pair of quotation marks. If there is one date with multiple times, indexing with only the date returns the data for all the times for that specific date. For example, the command `timeday('01-Jan-2001')` returns the data for all times on January 1, 2001:

```

ans =

desc: My first FINTS
freq: Daily (1)

'dates: (2)'   'times: (2)'   'Data1: (2)'
'01-Jan-2001' '11:00'   [          1]
'         '    '12:00'   [          2]

```

You can also indicate a specific date and time:

```

timeday('01-Jan-2001 12:00')

ans =

desc: My first FINTS
freq: Daily (1)

'dates: (1)'   'times: (1)'   'Data1: (1)'
'01-Jan-2001' '12:00'   [          2]

```

Use the double-colon operator `::` to specify a range of dates and times:

```

timeday('01-Jan-2001 12:00::03-Jan-2001 11:00')

ans =

desc: My first FINTS
freq: Daily (1)

'dates: (4)'   'times: (4)'   'Data1: (4)'
'01-Jan-2001' '12:00'   [          2]
'02-Jan-2001' '11:00'   [          3]

```

```
' ' " ' '12:00' [ 4]
'03-Jan-2001' '11:00' [ 5]
```

Treat `timeday` as a MATLAB structure if you want to obtain the contents of a specific field. For example, to find the times of day included in this object, enter

```
datestr(timeday.times)
```

```
ans =
```

```
11:00 AM
12:00 PM
11:00 AM
12:00 PM
11:00 AM
12:00 PM
```

Financial Time Series Operations

Several MATLAB functions have been overloaded to work with financial time series objects. The overloaded functions include basic arithmetic functions such as addition, subtraction, multiplication, and division and other functions such as arithmetic average, filter, and difference. Also, specific methods have been designed to work with the financial time series object. For a list of functions grouped by type, enter

```
help ftseries
```

at the MATLAB command prompt.

Basic Arithmetic

Financial time series objects permit you to do addition, subtraction, multiplication, and division, either on the entire object or on specific object fields. This is a feature that MATLAB structures do not allow. You cannot do arithmetic operations on entire MATLAB structures, only on specific fields of a structure.

You can perform arithmetic operations on two financial time series objects as long as they are compatible. (All contents are the same except for the description and the values associated with the data series.)

Note *Compatible* time series are not the same as *equal* time series. Two time series objects are equal when everything but the description fields is the same.

Here are some examples of arithmetic operations on financial time series objects.

Load a MAT-file that contains some sample financial time series objects:

```
load dji30short
```

One of the objects in `dji30short` is called `myfts1`:

```
myfts1 =
desc: DJI30MAR94.dat
freq: Daily (1)
'dates: (20)' 'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994' [ 3830.90] [ 3868.04] [ 3800.50] [ 3832.30]
'07-Mar-1994' [ 3851.72] [ 3882.40] [ 3824.71] [ 3856.22]
'08-Mar-1994' [ 3858.48] [ 3881.55] [ 3822.45] [ 3851.72]
'09-Mar-1994' [ 3853.97] [ 3874.52] [ 3817.95] [ 3853.41]
'10-Mar-1994' [ 3852.57] [ 3865.51] [ 3801.63] [ 3830.62]...
```

Create another financial time series object that is identical to `myfts1`:

```
newfts = fints(myfts1.dates, fts2mat(myfts1)/100,...
{'Open', 'High', 'Low', 'Close'}, 1, 'New FTS')
newfts =
desc: New FTS
freq: Daily (1)
'dates: (20)' 'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994' [ 38.31] [ 38.68] [ 38.01] [ 38.32]
'07-Mar-1994' [ 38.52] [ 38.82] [ 38.25] [ 38.56]
'08-Mar-1994' [ 38.58] [ 38.82] [ 38.22] [ 38.52]
'09-Mar-1994' [ 38.54] [ 38.75] [ 38.18] [ 38.53]
'10-Mar-1994' [ 38.53] [ 38.66] [ 38.02] [ 38.31]...
```

Perform an addition operation on both time series objects:

```
addup = myfts1 + newfts
addup =
desc: DJI30MAR94.dat
freq: Daily (1)
'dates: (20)' 'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994' [ 3869.21] [ 3906.72] [ 3838.51] [ 3870.62]
'07-Mar-1994' [ 3890.24] [ 3921.22] [ 3862.96] [ 3894.78]
```

```
'08-Mar-1994' [ 3897.06] [ 3920.37] [ 3860.67] [ 3890.24]
'09-Mar-1994' [ 3892.51] [ 3913.27] [ 3856.13] [ 3891.94]
'10-Mar-1994' [ 3891.10] [ 3904.17] [ 3839.65] [ 3868.93]...
```

Now, perform a subtraction operation on both time series objects:

```
subout = myfts1 - newfts
```

```
subout =
```

```
desc: DJI30MAR94.dat
```

```
freq: Daily (1)
```

```
'dates: (20)' 'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994' [ 3792.59] [ 3829.36] [ 3762.49] [ 3793.98]
'07-Mar-1994' [ 3813.20] [ 3843.58] [ 3786.46] [ 3817.66]
'08-Mar-1994' [ 3819.90] [ 3842.73] [ 3784.23] [ 3813.20]
'09-Mar-1994' [ 3815.43] [ 3835.77] [ 3779.77] [ 3814.88]
'10-Mar-1994' [ 3814.04] [ 3826.85] [ 3763.61] [ 3792.31]...
```

Operations with Objects and Matrices

You can also perform operations involving a financial time series object and a matrix or scalar:

```
addscalar = myfts1 + 10000
```

```
addscalar =
```

```
desc: DJI30MAR94.dat
```

```
freq: Daily (1)
```

```
'dates: (20)' 'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994' [ 13830.90] [ 13868.04] [ 13800.50] [ 13832.30]
'07-Mar-1994' [ 13851.72] [ 13882.40] [ 13824.71] [ 13856.22]
'08-Mar-1994' [ 13858.48] [ 13881.55] [ 13822.45] [ 13851.72]
'09-Mar-1994' [ 13853.97] [ 13874.52] [ 13817.95] [ 13853.41]
'10-Mar-1994' [ 13852.57] [ 13865.51] [ 13801.63] [ 13862.70]...
```

For operations with both an object and a matrix, the size of the matrix must match the size of the object. For example, a matrix to be subtracted from `myfts1` must be 20-by-4, since `myfts1` has 20 dates and four data series:

```
submtx = myfts1 - randn(20, 4)
```

```
submtx =
```

```
desc: DJI30MAR94.dat
freq: Daily (1)
```

```
'dates: (20)' 'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994' [ 3831.33] [ 3867.75] [ 3802.10] [ 3832.63]
'07-Mar-1994' [ 3853.39] [ 3883.74] [ 3824.45] [ 3857.06]
'08-Mar-1994' [ 3858.35] [ 3880.84] [ 3823.51] [ 3851.22]
'09-Mar-1994' [ 3853.68] [ 3872.90] [ 3816.53] [ 3851.92]
'10-Mar-1994' [ 3853.72] [ 3866.20] [ 3802.44] [ 3831.17]...
```

Arithmetic Operations with Differing Data Series Names

Arithmetic operations on two objects that have the same size but contain different data series names require the function `fts2mat`. This function extracts the values in an object and puts them into a matrix or vector, whichever is appropriate.

To see an example, create another financial time series object the same size as `myfts1` but with different values and data series names:

```
newfts2 = fints(myfts1.dates, fts2mat(myfts1/10000),...
{'Rat1', 'Rat2', 'Rat3', 'Rat4'}, 1, 'New FTS')
```

If you attempt to add (or subtract, and so on) this new object to `myfts1`, an error indicates that the objects are not identical. Although they contain the same dates, number of dates, number of data series, and frequency, the two time series objects do not have the same data series names. Use `fts2mat` to bypass this problem:

```
addother = myfts1 + fts2mat(newfts2);
```

This operation adds the matrix that contains the contents of the data series in the object `newfts2` to `myfts1`. You should carefully consider the effects on your data before deciding to combine financial time series objects in this manner.

Other Arithmetic Operations

In addition to the basic arithmetic operations, several other mathematical functions operate directly on financial time series objects. These functions include exponential (`exp`), natural logarithm (`log`), common logarithm (`log10`), and many more.

Data Transformation and Frequency Conversion

The data transformation and the frequency conversion functions convert a data series into a different format.

Data Transformation Functions

Function	Purpose
<code>boxcox</code>	Box-Cox transformation
<code>diff</code>	Differencing
<code>fillts</code>	Fill missing values
<code>filter</code>	Filter
<code>lagts</code>	Lag time series object
<code>leadts</code>	Lead time series object
<code>peravg</code>	Periodic average
<code>smoothts</code>	Smooth data
<code>tsmovavg</code>	Moving average

Frequency Conversion Functions

Function	New Frequency
<code>convertto</code>	As specified
<code>resamplets</code>	As specified
<code>toannual</code>	Annual
<code>todaily</code>	Daily
<code>tomonthly</code>	Monthly
<code>toquarterly</code>	Quarterly
<code>tosemi</code>	Semiannually
<code>toweekly</code>	Weekly

As an example look at `boxcox`, the Box-Cox transformation function. This function transforms the data series contained in a financial time series object into another set of data series with relatively normal distributions.

First create a financial time series object from the supplied `whirlpool.dat` data file.

```
whrl = ascii2fts('whirlpool.dat', 1, 2, []);
```

Fill any missing values denoted with NaNs in `whrl` with values calculated using the `linear` method:

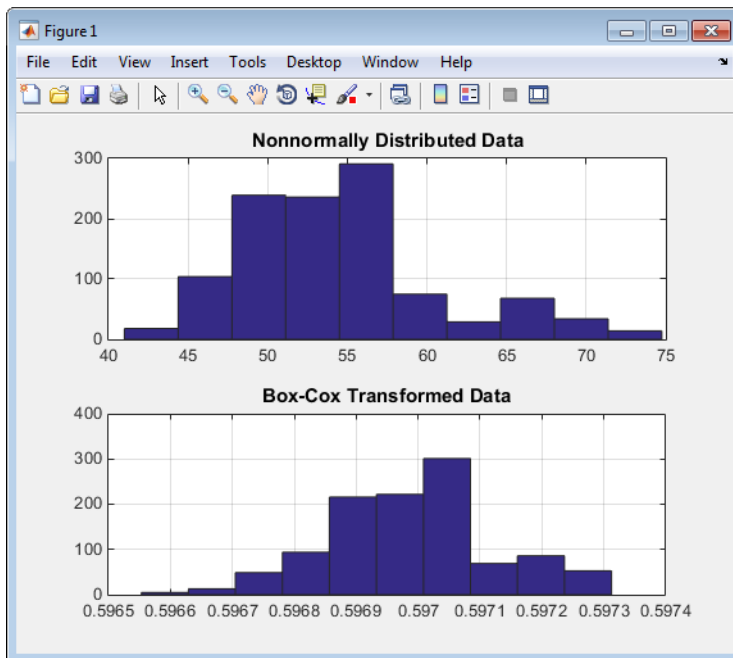
```
f_whr1 = fillts(whr1);
```

Transform the nonnormally distributed filled data series `f_whr1` into a normally distributed one using Box-Cox transformation:

```
bc_whr1 = boxcox(f_whr1);
```

Compare the result of the `Close` data series with a normal (Gaussian) probability distribution function and the nonnormally distributed `f_whr1`:

```
subplot(2, 1, 1);
hist(f_whr1.Close);
grid; title('Nonnormally Distributed Data');
subplot(2, 1, 2);
hist(bc_whr1.Close);
grid; title('Box-Cox Transformed Data');
```

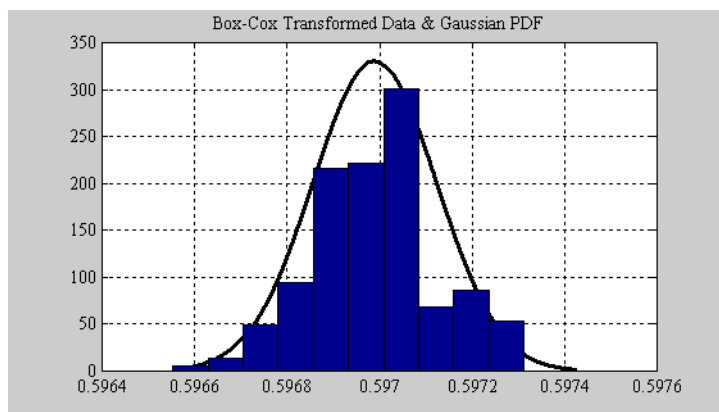


Box-Cox Transformation

The bar chart on the top represents the probability distribution function of the filled data series, `f_whr1`, which is the original data series `whr1` with the missing values

interpolated using the linear method. The distribution is skewed toward the left (not normally distributed). The bar chart on the bottom is less skewed to the left. If you plot a Gaussian probability distribution function (PDF) with similar mean and standard deviation, the distribution of the transformed data is very close to normal (Gaussian).

When you examine the contents of the resulting object `bc_whr1`, you find an identical object to the original object `whr1` but the contents are the transformed data series. If you have the Statistics Toolbox software, you can generate a Gaussian PDF with mean and standard deviation equal to those of the transformed data series and plot it as an overlay to the second bar chart. In the next figure, you can see that it is an approximately normal distribution.



Overlay of Gaussian PDF

The next example uses the `smoothts` function to smooth a time series.

To begin, transform `ibm9599.dat`, a supplied data file, into a financial time series object:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
```

Fill the missing data for holidays with data interpolated using the `fillts` function and the Spline fill method:

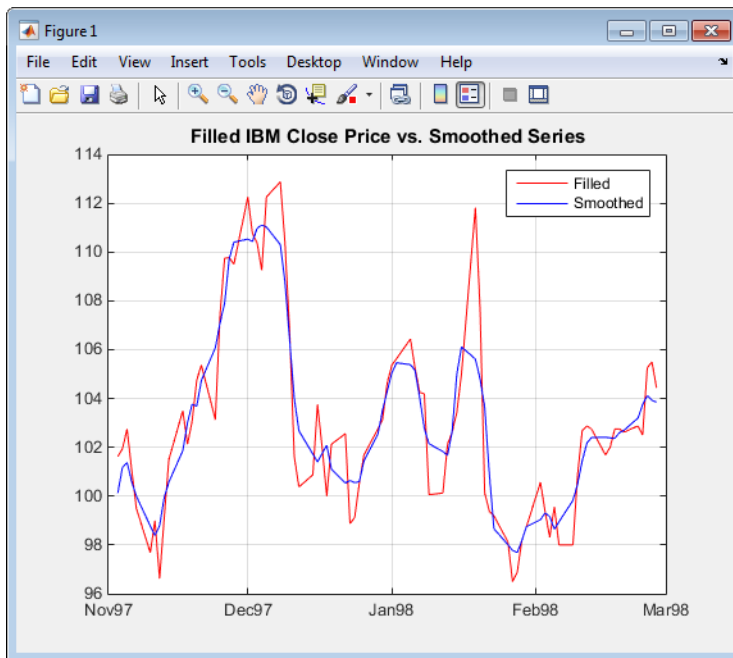
```
f_ibm = fillts(ibm, 'Spline');
```

Smooth the filled data series using the default Box (rectangular window) method:

```
sm_ibm = smoothts(f_ibm);
```


Now, plot the original and smoothed closing price series for IBM stock:

```
plot(f_ibm.CLOSE('11/01/97::02/28/98'), 'r')
datetick('x', 'mmyy')
hold on
plot(sm_ibm.CLOSE('11/01/97::02/28/98'), 'b')
hold off
datetick('x', 'mmyy')
legend('Filled', 'Smoothed')
title('Filled IBM Close Price vs. Smoothed Series')
```



Smoothed Data Series

These examples give you an idea of what you can do with a financial time series object. This toolbox provides some MATLAB functions that have been overloaded to work directly with these objects. The overloaded functions are those most commonly needed to work with time series data.

Using Time Series to Predict Equity Return

In this section...

“Overview” on page 12-22

“Load the Data” on page 12-23

“Create Financial Time Series Objects” on page 12-23

“Create Closing Prices Adjustment Series” on page 12-24

“Adjust Closing Prices and Make Them Spot Prices” on page 12-25

“Create Return Series” on page 12-25

“Regress Return Series Against Metric Data” on page 12-25

“Plot the Results” on page 12-26

“Calculate the Dividend Rate” on page 12-27

Overview

This example demonstrates a practical use of financial time series objects, predicting the return of a stock from a given set of data. The data is a series of closing stock prices, a series of dividend payments from the stock, and an explanatory series (in this case a market index). Additionally, the example calculates the dividend rate from the stock data provided.

Note You can find a file for this demonstration program in the directory *matlabroot/toolbox/finance/ftsdemos* on your MATLAB path. The file is named `predict_ret.m`.

To perform these computations:

- 1 Load the data.
- 2 Create financial time series objects from the loaded data.
- 3 Create the series from dividend payment for adjusting the closing prices.
- 4 Adjust the closing prices and make them the spot prices.
- 5 Create the return series.

- 6 Regress the return series against the metric data (for example, a market index) using the MATLAB \ operator.
- 7 Plot the results.
- 8 Calculate the dividend rate.

Load the Data

The data for this demonstration is found in the MAT-file `predict_ret_data.mat`:

```
load predict_ret_data.mat
```

The MAT-file contains six vectors:

- Dates corresponding to the closing stock prices, `sdates`
- Closing stock prices, `sdata`
- Dividend dates, `divdates`
- Dividend paid, `divdata`
- Dates corresponding to the metric data, `expdates`
- Metric data, `expdata`

Use the `whos` command to see the variables in your MATLAB workspace.

Create Financial Time Series Objects

It is useful to work with financial time series objects rather than with the vectors now in the workspace. By using objects, you can easily keep track of the dates. Also, you can easily manipulate the data series based on dates because the object keeps track of the administration of time series for you.

Use the object constructor `fints` to construct three financial time series objects.

```
t0 = fints(sdates, sdata, {'Close'}, 'd', 'Inc');
d0 = fints(divdates, divdata, {'Dividends'}, 'u', 'Inc');
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index');
```

The variables `t0`, `d0`, and `x0` are financial time series objects containing the stock closing prices, dividend payments, and the explanatory data, respectively. To see the contents

of an object, type its name at the MATLAB command prompt and press **Enter**. For example:

```
d0
d0 =
  'desc:'          'Inc'
  'freq:'          'Unknown (0)'
           ''      ''
  'dates: (4)'    'Dividends: (4)'
  '04/15/99'     '0.2000'
  '06/30/99'     '0.3500'
  '10/02/99'     '0.2000'
  '12/30/99'     '0.1500'
```

Create Closing Prices Adjustment Series

The price of a stock is affected by the dividend payment. On the day before the dividend payment date, the stock price reflects the amount of dividend to be paid the next day. On the dividend payment date, the stock price is decreased by the amount of dividend paid. Create a time series that reflects this adjustment factor:

```
dadj1      = d0;
dadj1.dates = dadj1.dates-1;
```

Now create the series that adjust the prices at the day of dividend payment; this is an adjustment of 0. You also need to add the previous dividend payment date since the stock price data reflect the period subsequent to that day; the previous dividend date was December 31, 1998:

```
dadj2      = d0;
dadj2.Dividends = 0;
dadj2      = fillts(dadj2, 'linear', '12/31/98');
dadj2('12/31/98') = 0;
```

Combining the two objects above gives the data needed to adjust the prices. However, since the stock price data is daily data and the effect of the dividend is linearly divided during the period, use the `fillts` function to make a daily time series from the adjustment data. Use the dates from the stock price data to make the dates of the adjustment the same:

```
dadj3 = [dadj1; dadj2];
```

```
dadj3 = fillts(dadj3, 'linear', t0.dates);
```

Adjust Closing Prices and Make Them Spot Prices

The stock price recorded already reflects the dividend effect. To obtain the “correct” price, subtract the dividend amount from the closing prices. Put the result inside the same object `t0` with the data series name `Spot`.

To make sure that adjustments correspond, index into the adjustment series using the dates from the stock price series `t0`. Use the `datestr` command because `t0.dates` returns the dates in serial date format. Also, since the data series name in the adjustment series `dadj3` does not match the one in `t0`, use the function `fts2mat`:

```
t0.Spot = t0.Close - fts2mat(dadj3(datestr(t0.dates)));
```

Create Return Series

Now calculate the return series from the stock price data. A stock return is calculated by dividing the difference between the current closing price and the previous closing price by the previous closing price.

```
tret = (t0.Spot - lagts(t0.Spot, 1)) ./ lagts(t0.Spot, 1);
tret = chfield(tret, 'Spot', 'Return');
```

Ignore any warnings you receive during this sequence. Since the operation on the first line above preserves the data series name `Spot`, it has to be changed with the `chfield` command to reflect the contents correctly.

Regress Return Series Against Metric Data

The explanatory (metric) data set is a weekly data set while the stock price data is a daily data set. The frequency needs to be the same. Use `todayly` to convert the weekly series into a daily series. The constant needs to be included here to get the constant factor from the regression:

```
x1 = todayly(x0);
x1.Const = 1;
```

Get all the dates common to the return series calculated above and the explanatory (metric) data. Then combine the contents of the two series that have dates in common into a new time series:

```
dcommon = intersect(tret.dates, x1.dates);  
regts0 = [tret(datestr(dcommon)), x1(datestr(dcommon))];
```

Remove the contents of the new time series that are not finite:

```
finite_regts0 = find(all(isfinite( fts2mat(regts0)), 2));  
regts1 = regts0( finite_regts0 );
```

Now, place the data to be regressed into a matrix using the function `fts2mat`. The first column of the matrix corresponds to the values of the first data series in the object, the second column to the second data series, and so on. In this case, the first column is regressed against the second and third column:

```
DataMatrix = fts2mat(regts1);  
XCoeff = DataMatrix(:, 2:3) \ DataMatrix(:, 1);
```

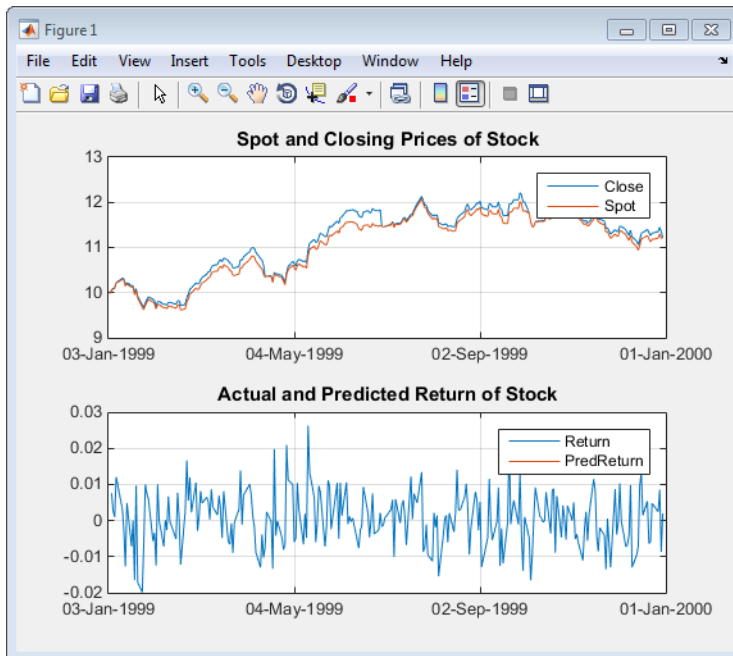
Using the regression coefficients, calculate the predicted return from the stock price data. Put the result into the return time series `tret` as the data series `PredReturn`:

```
RetPred = DataMatrix(:,2:3) * XCoeff;  
tret.PredReturn(datestr(regts1.dates)) = RetPred;
```

Plot the Results

Plot the results in a single figure window. The top plot in the window has the actual closing stock prices and the dividend-adjusted stock prices (spot prices). The bottom plot shows the actual return of the stock and the predicted stock return through regression:

```
subplot(2, 1, 1);  
plot(t0);  
title('Spot and Closing Prices of Stock');  
subplot(2, 1, 2);  
plot(tret);  
title('Actual and Predicted Return of Stock');
```



Closing Prices and Returns

Calculate the Dividend Rate

The last part of the task is to calculate the dividend rate from the stock price data. Calculate the dividend rate by dividing the dividend payments by the corresponding closing stock prices.

First check to see if you have the stock price data on all the dividend dates:

```
datestr(d0.dates, 2)
ans =
04/15/99
06/30/99
10/02/99
12/30/99
t0(datestr(d0.dates))
ans =
```

```

'desc:'          'Inc'          ''
'freq:'          'Daily (1)'      ''
      ''          ''          ''
'dates: (3)'     'Close: (3)'     'Spot: (3)'
'04/15/99'      '10.3369'      '10.3369'
'06/30/99'      '11.4707'      '11.4707'
'12/30/99'      '11.2244'      '11.2244'

```

Note that stock price data for October 2, 1999 does not exist. The `fillts` function can overcome this situation; `fillts` allows you to insert a date and interpolate a value for the date from the existing values in the series. There are a number of interpolation methods. For more information, see `fillts`.

Use `fillts` to create a new time series containing the missing date from the original data series. Then set the frequency indicator to daily:

```

t1 = fillts(t0, 'nearest', d0.dates);
t1.freq = 'd';

```

Calculate the dividend rate:

```

tdr = d0./fts2mat(t1.Close(datestr(d0.dates)))

```

tdr =

```

'desc:'          'Inc'          ''
'freq:'          'Unknown (0)'    ''
      ''          ''          ''
'dates: (4)'     'Dividends: (4)'
'04/15/99'      '0.0193'
'06/30/99'      '0.0305'
'10/02/99'      '0.0166'
'12/30/99'      '0.0134'

```


Financial Time Series App

- “What Is the Financial Time Series App?” on page 13-2
- “Getting Started with the Financial Time Series App ” on page 13-3
- “Loading Data with the Financial Time Series App” on page 13-5
- “Using the Financial Time Series App” on page 13-10
- “Using the Financial Time Series App with GUIs” on page 13-17

What Is the Financial Time Series App?

The Financial Time Series app enables you to create and manage financial time series (`fints`) objects. The Financial Time Series app interoperates with the Financial Time Series Graphical User Interface (`ftsgui`) and Interactive Chart (`chartfts`). In addition, you can use Datafeed Toolbox™ or Database Toolbox™ software to connect to external data sources.

A financial time series object minimally consists of:

- `Desc`, which is the description field.
- `Freq`, which is a frequency indicator field.
- `Dates`, which is a date vector field. If the date vector incorporates time-of-day information, the object contains an additional field named `times`.
- In addition, you can have at least one data series vector. You can specify names for any data series vectors. If you do not specify names, the object uses the default names `series1`, `series2`, `series3`, and so on.

In general, the workflow for using the Financial Time Series app is:

- 1 Acquire data.
- 2 Create a variable.
- 3 Convert the variable to `fints`.
- 4 Convert `fints` to a MATLAB double object.

To obtain the data for the Financial Time Series app, you need to use a MATLAB double object or a financial time series (`fints`) object. You can use previously stored internal data on your computer or you can connect to external data sources using Datafeed Toolbox or Database Toolbox software.

Note: You must obtain a license for these products from MathWorks before you can use either of these toolboxes.

After creating a financial time series object, you can use the Financial Time Series app to change the characteristics of the time series object, including merging with other financial time series objects, removing rows or columns, and changing the frequency. You can also use the Financial Time Series app to generate various forms of plotted output and you can reconvert a `fints` object to a MATLAB double-precision matrix.

Getting Started with the Financial Time Series App

To start the Financial Time Series app:

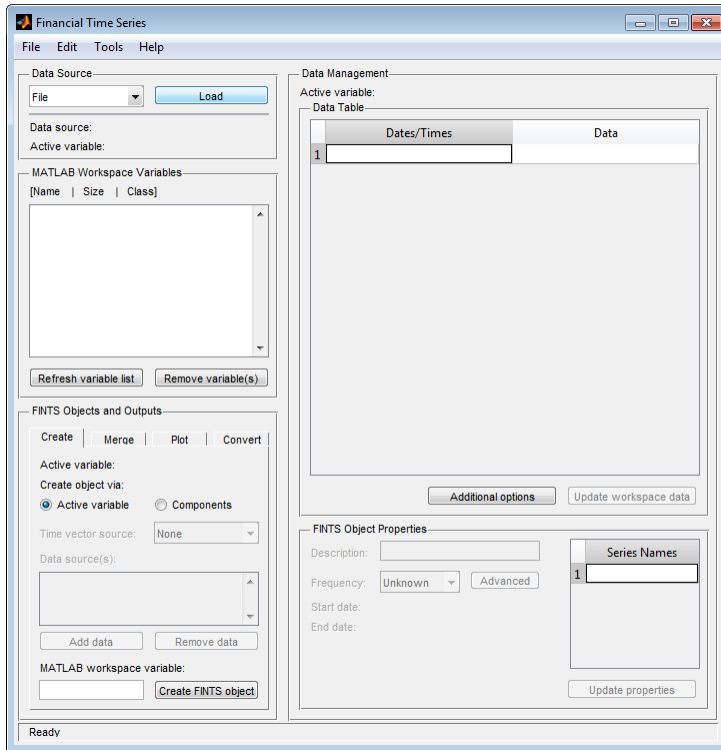
- 1** On the MATLAB desktop toolstrip, click the **Apps** tab and in the apps gallery, under **Computational Finance**, click **Financial Time Series**.

The Financial Time Series app opens. For an overview of the Financial Time Series app, see “What Is the Financial Time Series App?” on page 13-2.

- 2** To load data with the Financial Time Series app, see “Loading Data with the Financial Time Series App” on page 13-5.

If you plan to load data from Database Toolbox or Datafeed Toolbox software, ensure you have a license. For more information on using these toolboxes, see the Database Toolbox and Datafeed Toolbox documentation.

- 3** For more information on the tasks supported by the Financial Time Series app, see “Using the Financial Time Series App” on page 13-10 and “Using the Financial Time Series App with GUIs” on page 13-17.



Loading Data with the Financial Time Series App

In this section...

“Overview” on page 13-5

“Obtaining External Data” on page 13-5

“Obtaining Internal Data” on page 13-7

“Viewing the MATLAB Workspace” on page 13-8

Overview

The **Data source** pane in the Financial Time Series app lets you do the following:

- Obtain live data from various external data servers using either Datafeed Toolbox or Database Toolbox software.
- Load data you previously obtained and stored in a file.
- View data contained within the MATLAB workspace.

Obtaining External Data

You can obtain external data using Datafeed Toolbox or Database Toolbox software. Datafeed Toolbox software lets you obtain data from several financial data servers, including:

- Bloomberg[®]
- eSignal[®]
- FactSet[®]
- Federal Reserve Economic Data (FRED[®])
- Haver Analytics[®] financial data
- Interactive Data[™]
- IQFEED[®]
- Kx Systems[®], Inc. kdb+ database

- Thomson Reuters™
- SIX Financial Information
- Yahoo!®

Except for Federal Reserve Economic Data (FRED) and Yahoo!, these data servers require that you obtain a license from the vendor before you can access their data.

Tip If you open Datafeed Toolbox or Database Toolbox software before starting the Financial Time Series app, the Financial Time Series app is unable to recognize the toolboxes. When working with the Financial Time Series app, select **File > Load** to open these toolboxes.

Obtaining External Data with Datafeed Toolbox Software

- 1 From the Financial Time Series app, select **File > Load > Datafeed Toolbox** to open the toolbox.
- 2 Click the **Connection** tab in Datafeed Toolbox software to select the data source you want to load into the Financial Time Series app.
- 3 Click the **Data** tab in Datafeed Toolbox software to select the security and the associated data that you want to load into the Financial Time Series app.
- 4 After using Datafeed Toolbox software to define the connection, security, data, and **MATLAB variable** name, click **Get Data** and then, using the Financial Time Series app, click **Refresh variable list**. The **Data source** field in the Financial Time Series app displays the name of the security you selected from the **Data** tab in Datafeed Toolbox software. The Financial Time Series app **Active variable** field indicates the name of the MATLAB workspace variable you chose for this security.
- 5 Click **Close** to exit Datafeed Toolbox software. The Financial Time Series app clears the **Data source** and **Active variable** fields.

Obtaining External Data with Database Toolbox Software

- 1 From the Financial Time Series app, select **File > Load > Database Toolbox** to open the toolbox.
- 2 From the Visual Query Builder window, select the data you want to load into the Financial Time Series app.

- 3 After using Database Toolbox software to select data and name the **MATLAB workspace variable**, click **Execute** and then, using the Financial Time Series app, click **Refresh variable list**. The **Data source** field in the Financial Time Series app displays the name of the highlighted data source that you selected from the **Data** list box in the Visual Query Builder window. The Financial Time Series app **Active variable** field indicates the name of the MATLAB workspace variable you chose for the security in the Visual Query Builder window.
- 4 From the Database Toolbox software, select **Query > Close Visual Query Builder**, the Financial Time Series app clears the **Data source** and **Active variable** fields.

Obtaining Internal Data

You can use the Financial Time Series app to load data from files previously stored on your computer. The types of data files you can load are as follows:

- MATLAB **.mat** files, with or without **fints** objects
- ASCII text files (**.dat** or **.txt** suffixes)
- Excel **.xls** files

To obtain internal data:

- 1 From the Financial Time Series app, select **File > Load > File** to open the Load a MAT, ASCII, XLS File dialog box.
- 2 Select the data you want to load into the Financial Time Series app.
 - If you load a MATLAB MAT-file, the variables in the file are placed into the MATLAB workspace. The **MATLAB Workspace Variables** list box shows the variables that have been added to the workspace. For example, if you load the file **disney.mat**, which is distributed with the toolbox, the **MATLAB Workspace Variables** list box displays the variables in that MAT-file.

Note: The Financial Time Series app automatically generates a line plot for each workspace variables unless you disable this feature by resetting the default action under **File > Preferences > Generate line plot on load**.

- If you load a **.dat** or an ASCII **.txt** file, the ASCII File Parameters dialog box opens. Use this dialog box to transform a text data file into a MATLAB financial time series **fints** object. The format for the ascii data must be:

- Dates must be in a valid date string format:
 - 'ddmmyy' or 'ddmmyyyy'
 - 'mm/dd/yy' or 'mm/dd/yyyy'
 - 'dd-mmm-yy' or 'dd-mmm-yyyy'
 - 'mmm.dd,yy' or 'mmm.dd,yyyy'
- Time information must be in 'hh:mm' format.
- Each column must be separated either by spaces or a tab.

For more information on converting ascii data to a `fints` object, see `ascii2fts`.

- If you load an Excel `.xls` file, the Excel File Parameters dialog box opens. Use this dialog box to transform Excel worksheet data into a MATLAB financial time series (`fints`) object.
- 3** From the Financial Time Series app, select **File > Save** to save the data you loaded from an internal file.

Viewing the MATLAB Workspace

The **MATLAB Workspace Variables** list box displays all existing MATLAB workspace variables. Double-click any variable to display the data in the **Data Table**. You can only display financial time series (`fints`) objects, MATLAB doubles, and cell arrays of double data in the **Data Table**.

The screenshot displays the Financial Time Series app interface. The main window is titled "Financial Time Series" and contains several panels:

- Data Source:** Shows "Datafeed Toolbox" selected, with "GOOG Equity" as the data source and "cur1" as the active variable. A "Load" button is present.
- MATLAB Workspace Variables:** A list box showing variables "cur1" (29x7 double) and "myFts" (29x6 fints). Below the list are "Refresh variable list" and "Remove variable(s)" buttons.
- FINTS Objects and Outputs:** Includes tabs for "Create", "Merge", "Plot", and "Convert". The active variable is "myFts". It offers options to create an object via "Active variable" (selected) or "Components". The time vector source is "cur1". There are "Add data" and "Remove data" buttons, and a "Create FINTS object" button.
- Data Management:** Shows the active variable "myFts" and a "Data Table" with columns: Dates/Times, Close, Date, High, and Low. The table contains 16 rows of data from 09-Apr-2013 to 30-Apr-2013. Below the table are "Additional options" and "Update workspace data" buttons.
- FINTS Object Properties:** Shows "Description: GOOG Equity", "Frequency: Daily", "Start date: 09-Apr-2013", and "End date: 17-May-2013". There is an "Advanced" button. A "Series Names" list box contains "Close", "Date", "High", "Low", and "Open". An "Update properties" button is at the bottom.

The status bar at the bottom left indicates "Ready".

In addition, you can click **Refresh variable list** to refresh the **MATLAB Workspace Variables** list box. You need to refresh this list periodically because it is refreshed automatically only for operations performed with the Financial Time Series app, not for operations performed within MATLAB itself.

Click **Remove variable(s)** to remove variable from the **MATLAB Workspace Variables** list and from the MATLAB workspace.

Using the Financial Time Series App

In this section...

“Creating a Financial Time Series Object” on page 13-10

“Merge Financial Time Series Objects” on page 13-11

“Converting a Financial Time Series Object to a MATLAB Double-Precision Matrix” on page 13-11

“Plotting the Output in Several Formats” on page 13-12

“Viewing Data for a Financial Time Series Object in the Data Table” on page 13-13

“Modifying Data for a Financial Time Series Object in the Data Table” on page 13-14

“Viewing and Modifying the Properties for a FINTS Object” on page 13-16

Creating a Financial Time Series Object

Using the **Create** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can create a financial time series (`fints`) object from one or more selected variables.

Note: When you first start the Financial Time Series app, the **Create** tab appears on top, unless you reset the default using **File > Preferences > Show Create tab when ftstool starts**.

To create a financial time series (`fints`) object from one or more selected variables:

- 1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox or Database Toolbox software or an internal data source using **File > Load > File**.
- 2 Select one or more variables from the **MATLAB Workspace Variables** list.
- 3 Click the **Create** tab and then click **Active variable**.

When combining multiple variables, you can type a new variable name for the combined variables in the **MATLAB workspace variable** box. The new variable name is added to the **MATLAB Workspace Variables** list. (If you do not choose a name for the **MATLAB workspace variable**, the Financial Time Series app uses the default name `myFts`.)

- 4 Click **Create FINTS object** to display the result in the **Data Table**.

Merge Financial Time Series Objects

Using the **Create** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can create a new financial time series object by merging (joining) multiple existing financial time series objects.

Note: When you first start the Financial Time Series app, the **Create** tab appears on top, unless you reset the default using **File > Preferences**.

To create a financial time series (`fints`) object by merging multiple existing financial time series objects:

- 1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox or Database Toolbox software or an internal data source using **File > Load > File**.
- 2 To merge multiple existing financial time series objects, click the **Create** tab, click **Components**, and then select a value for the **Time vector source** and one or more items from the **Data sources** list.

Note: You can merge at once multiple financial time series objects. For more information on merging `fints` objects, see [merge](#).

- 3 Click **Create FINTS object** to display the result in the **Data Table**.

Converting a Financial Time Series Object to a MATLAB Double-Precision Matrix

Using the **Convert** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can convert a financial time series (`fints`) object to a MATLAB double-precision matrix.

To create a financial time series object from one or more selected variables:

- 1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox or Database Toolbox software or an internal data source using **File > Load > File**.

- 2 Select a variable from the **MATLAB Workspace Variables** list box.
- 3 Click the **Convert** tab and then determine whether to include or exclude dates in the conversion by clicking **Include dates** or **Exclude dates**.
- 4 Type a variable name in the **Output variable name** box. (If you do not choose a variable name, the Financial Time Series app uses the default name `myDb1`.)
- 5 Click **Convert FINTS to double matrix**. (This operation is equivalent to performing `fts2mat` on a financial time series object.)

Plotting the Output in Several Formats

Using the **Plot** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can create several forms of plotted output by using a selection list. You can create four types of bar charts, candle plots, high-low plots, line plots, and interactive charts (the latter is created by using the interoperation of the Financial Time Series app with the function `chartfts`).

The set of plots supported by the Financial Time Series app are identical to the set provided by the **Graphs** menu of the Financial Time Series GUI. (See “Graphs Menu” on page 14-14.) You can find more detailed information for the supported plots by consulting the reference page for each individual type of plot.

To create a plotted output:

- 1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox or Database Toolbox software or an internal data source using **File > Load > File**.
- 2 Select a variable from the **MATLAB Workspace Variables** list box or select data from the **Data Table**.
- 3 Click the **Plot** tab and indicate whether you are plotting based on a workspace variable or data from the **Data Table**.
- 4 From the **Type** drop-down list, select the type of plot.
- 5 Click **Plot**. The plot is displayed.

Note: If the selected workspace variable that you are plotting is not a `fints` object, a `fints` object is created when you click **Plot**. The new `fints` object uses the name designated by the **MATLAB workspace variable** box on the **Create** tab.

Viewing Data for a Financial Time Series Object in the Data Table

Once a financial time series (`fints`) object is created, the Financial Time Series app **Data Table** displays user-designated data, including financial time series objects, MATLAB double-precision variables, and cell arrays of doubles. (Cell arrays of doubles is often the resulting format when using Database Toolbox software.)

When displaying double variables (or a cell array of doubles) in the **Data Table**, the column headings for a double variable or cell array of doubles displayed in the **Data Table** are labeled **A**, **B**, **C**, and so on.

Overwriting Data in the Data Table Display

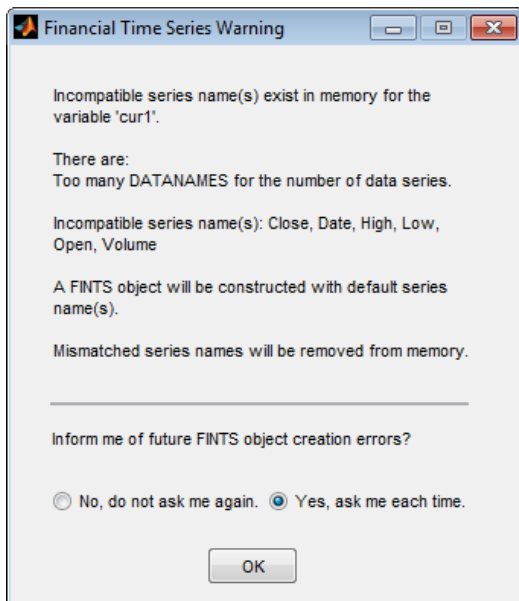
If you use the command line to overwrite data previously retrieved using Datafeed Toolbox or Database Toolbox software, two events could occur:

- If the new data contains the same number of columns as before, the headers remain unchanged when you attempt to create a financial time series (`fints`) object using the modified data.
- If the data contains a different number of columns, a warning dialog box appears.

For example, assume that you use Datafeed Toolbox software to obtain **Close**, **Date**, **High**, **Low**, **Open**, and **Volume** data for the equity GOOG. You store the data in the MATLAB workspace with the variable name `cur1`. From the command line, if you redefine the variable `cur1`, eliminating the second column (**Close**)

```
cur1(:,2) = [ ]
```

and then return to the Financial Time Series app and attempt to create a financial time series object, a warning dialog box appears.



Modifying Data for a Financial Time Series Object in the Data Table

The Financial Time Series app lets you update your data displayed in the **Data Table** by adding or removing rows or columns.

Note: Modifying data in the **Data Table** will not update the MATLAB workspace variable. To update the workspace variable after modifying the **Data Table**, click **Update workspace variable**.

Adding and Removing Rows

To add a row of data displayed in the **Data Table**:

- 1 Select a row from the **Data Table** display where you want to add a row. Click **Additional options** to open the Data Table Options dialog box.
- 2 Click **Add row**. The default is to add the row up. To add a row down, select **Insertion option** and then click **Add down**. In addition, you can select the **Insertion option** of **Date** to designate a specific date. (If a date is not specified,

the added row will contain a date that is chronologically in order with respect to the initial row.)

When you add rows, the **Data Table** display is immediately updated.

To remove a row of data from the **Data Table**:

- 1 Select one or more rows in the **Data Table** display that you want to remove. Click **Additional options** to open the Data Table Options dialog box.
- 2 Click **Remove row(s)**. The default is to remove the selected rows. In addition, to remove selected rows, select **Removal options** and then select other options for row removal from the **Remove rows** list box. You can specify a **Start** and **End** date or you can click the **Non-uniform range setting** option to designate a range.

When you remove rows, the **Data Table** display is updated immediately.

Adding and Removing Columns

To add a column of data displayed in the **Data Table**:

- 1 Select a column from the **Data Table** display where you want to add a column. Click **Additional options** to open the Data Table Options dialog box.
- 2 Click **Add column**. The default is to add the column to the left of the selected column.

Note: For time series objects, you cannot add a column to the left of the **Date/Times** column; there is no restriction for double data.

To add a column to the right, select **Insertion option** and then click **Add right**. In addition, you can use the **Insertion option** of **New Column Name** to designate a specific column name. (If a **New Column Name** is not specified, an added column will contain a column name of **series1**, **series2**, and so on.)

When you add columns, the **Data Table** display is updated immediately.

To remove a column of data displayed in the **Data Table**:

- 1 Select one or more columns in the **Data Table** display that you want to remove. Click **Additional options** to open the Data Table Options dialog box.

- 2 Click **Remove column(s)**. The default is to remove the selected rows. In addition, to remove selected columns, select **Removal options** and then select columns for removal from the **Remove columns** list box.

When you remove columns, the **Data Table** display is updated immediately.

Viewing and Modifying the Properties for a FINTS Object

The **FINTS Object Properties** pane in the Financial Time Series app lets you modify financial time series (`fints`) object properties. This area becomes active whenever the **Data Table** displays a financial time series object.

To modify the properties for a `fints` object:

- 1 After you create a `fints` object, double-click the object name in the **MATLAB Workspace Variables** list box to open the **Data Table** and display the `fints` object properties.
- 2 Click to modify the **Description**, **Frequency**, or **Series Names** fields.

The **Frequency** drop-down list supports the following conversion functions:

Function	New Frequency
<code>toannual</code>	Annual
<code>todayly</code>	Daily
<code>tomonthly</code>	Monthly
<code>toquarterly</code>	Quarterly
<code>tosemi</code>	Semiannually
<code>toweekly</code>	Weekly

- 3 Click **Update properties** to save the changes. This action also updates the associated workspace variable.

Using the Financial Time Series App with GUIs

The Financial Time Series app works with Datafeed Toolbox and Database Toolbox software to load data. In addition, the Financial Time Series app interoperates with `chartfts` to display an interactive plot and `ftsgui` to perform further time series data analysis.

The workflow for using the Financial Time Series app with `chartfts` is:

- 1 After loading data from either Datafeed Toolbox or Database Toolbox software or an internal file, select a variable from the **MATLAB Workspace Variables** list box.
- 2 Click the **Plot** tab, click **Type**, and then select **Interactive Chart**.
- 3 Click **Plot**. The interactive plot is displayed in `chartfts`. You can then use `chartfts` menu items for further display options.

For more information on `chartfts`, select **Help > Graphics Help**.

The workflow for using the Financial Time Series app with the Financial Time Series GUI (`ftsgui`) is:

- 1 After loading data from either Datafeed Toolbox or Database Toolbox software or an internal file, select a variable from the **MATLAB Workspace Variables** list box.
- 2 Select **Tools > FTSGUI** to open the Financial Time Series GUI window.
- 3 Select a variable from the **MATLAB Workspace Variables** list box. Click the **Plot** tab and then select one of the following from the **Type** drop-down list: **Line Plot**, **High-Low Plot**, or **Candlestick Plot**.
- 4 Click **Plot**. The plot is displayed in a MATLAB graphic window. In addition, the Financial Time Series GUI window displays an entry for the plotted `fints` object. You can then use the menu items in the Financial Time Series GUI window to perform further analysis.

For more information on `ftsgui`, select **Help > Help on Financial Time Series GUI**.

Note: If the selected workspace variable that you are plotting is not a `fints` object, a `fints` object is created when you click **Plot**. The new `fints` object uses the name designated by the **MATLAB workspace variable** box on the **Create** tab.

Financial Time Series Graphical User Interface

- “Financial Time Series Graphical User Interface” on page 14-2
- “Using the Financial Time Series GUI” on page 14-7

Financial Time Series Graphical User Interface

Use the Financial Time Series GUI to analyze your time series data and display the results graphically without resorting to the command line. The Financial Time Series GUI lets you visualize the data and the results at the same time.

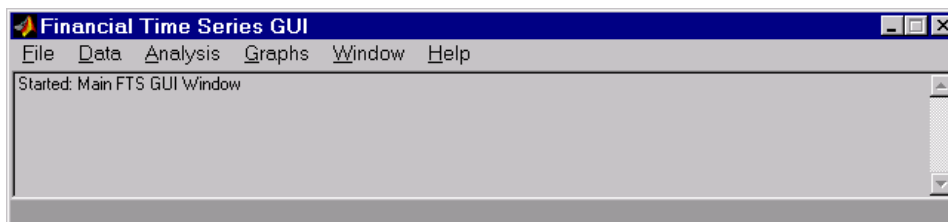
“Using the Financial Time Series GUI” on page 14-7 discusses how to use this GUI.

Main Window

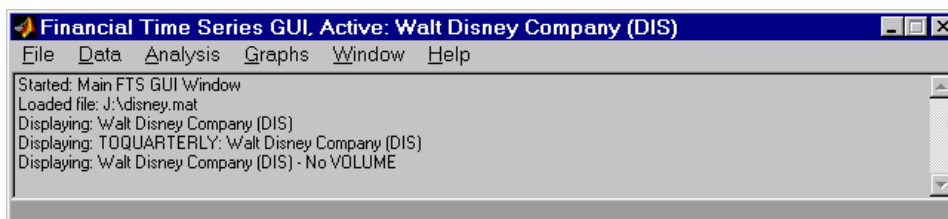
Start the financial time series GUI with the command

```
ftsgui
```

The Financial Time Series GUI window opens.

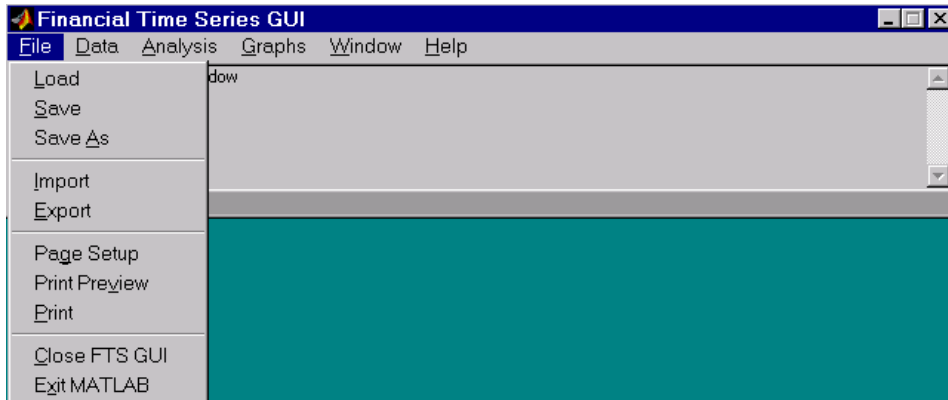


The title bar acts as an active time series object indicator (indicates the currently active financial time series object). For example, if you load the file `disney.mat` and want to use the time series data in the file `dis`, the title bar on the main GUI would read as shown.



The menu bar consists of six menu items: **File**, **Data**, **Analysis**, **Graphs**, **Window**, and **Help**. Under the menu bar is a status box that displays the steps you are doing.

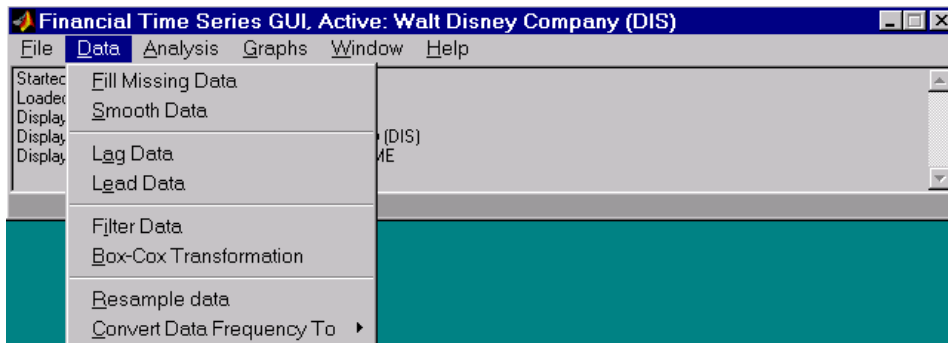
File Menu



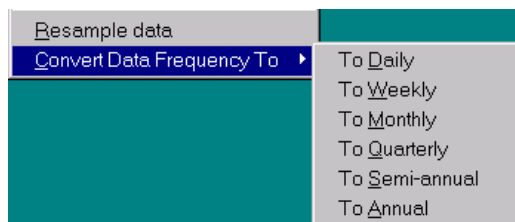
The **File** menu contains the commands for input and output. You can read and save (**Load**, **Save**, and **Save As**) MATLAB MAT-files, ASCII (text) data files. To load MATLAB MAT-files, the MAT-file must contain a `fints` object. You can also import (**Import**) Excel XLS files. MATLAB software does not support the export of Excel XLS files at this time.

The **File** menu also contains the printing suite (**Page Setup**, **Print Preview**, and **Print**). Lastly, from this menu you can close the GUI itself (**Close FTS GUI**) and quit MATLAB (**Exit MATLAB**).

Data Menu

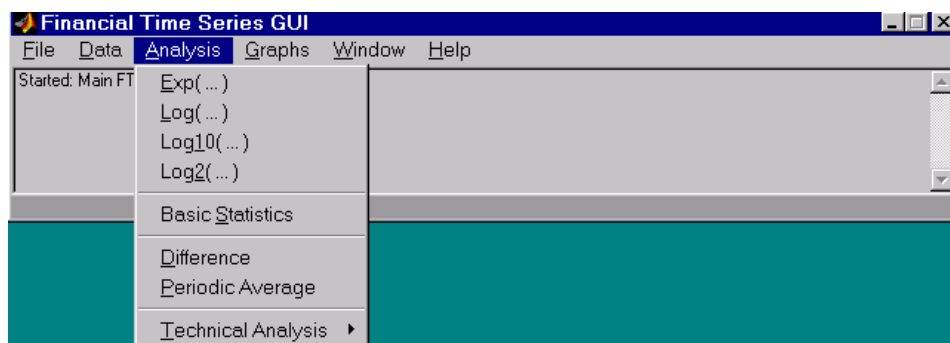


The **Data** menu provides a collection of data manipulation functions and data conversion functions.



To use any of the functions here, make sure that the correct financial time series object is displayed in the title bar of the main GUI window.

Analysis Menu

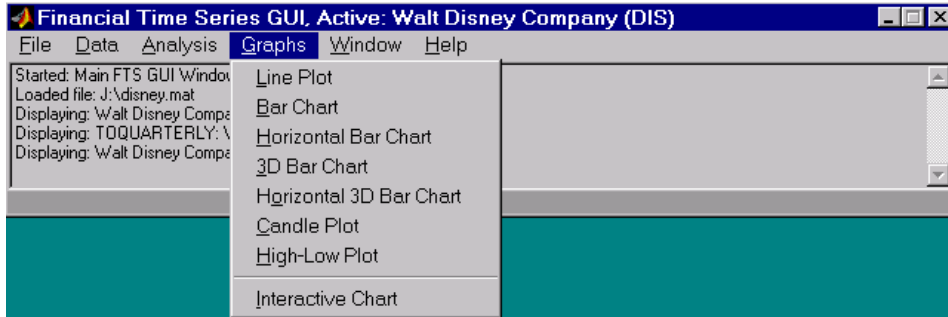


The **Analysis** menu provides

- A set of exponentiation and logarithmic functions.
- Statistical tools (**Basic Statistics**), which calculate and display the minimum, maximum, average (mean), standard deviation, and variance of the current (active) time series object; these basic statistics numbers are displayed in a dialog window.
- Data difference (**Difference**) and periodic average (**Periodic Average**) calculations. Data difference generates a vector of data that is the difference between the first data point and the second, the second and the third, and so on. The periodic average function calculates the average per defined length period, for example, averages of every five days.
- Technical analysis functions. See “Chart Technical Indicators” for a list of the technical analysis functions.

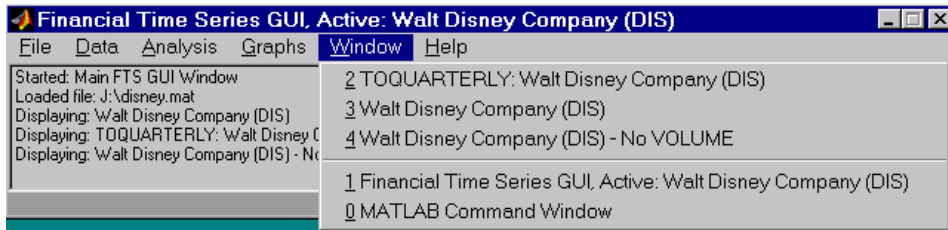
As with the **Data** menu, to use any of the **Analysis** menu functions, make sure that the correct financial time series object is displayed in the title bar of the main GUI window.

Graphs Menu



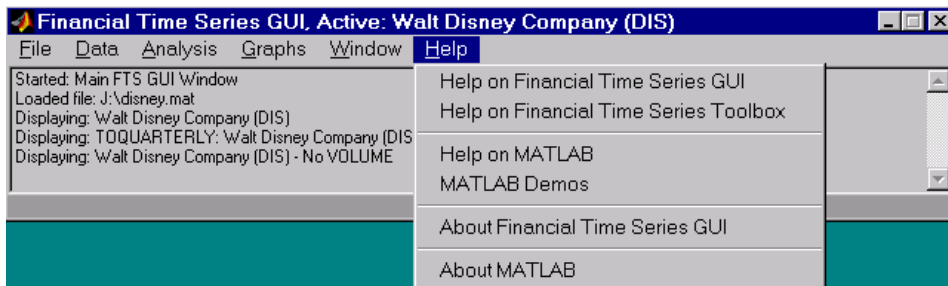
The **Graphs** menu contains functions that graphically display the current (active) financial time series object. You can also start up the interactive charting function (`chartfts`) from this menu.

Window Menu



The **Window** menu lists open windows under the current MATLAB session.

Help Menu



The **Help** menu provides a standard set of Help menu links.

Using the Financial Time Series GUI

In this section...

“Getting Started” on page 14-7

“Data Menu” on page 14-9

“Analysis Menu” on page 14-13

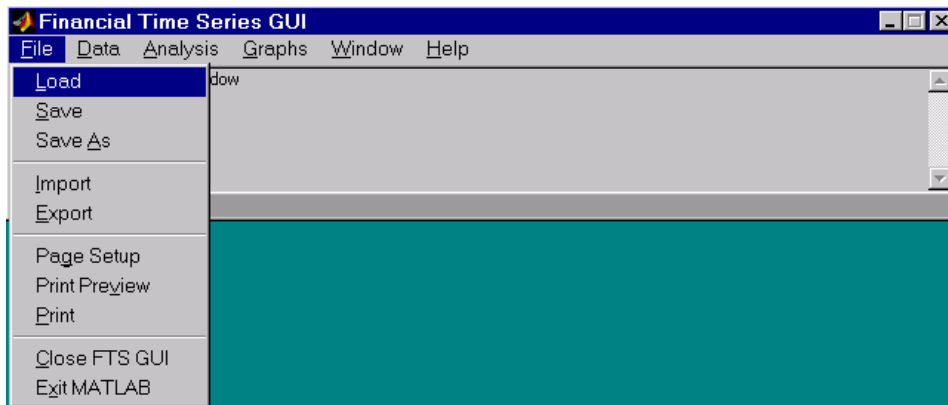
“Graphs Menu” on page 14-14

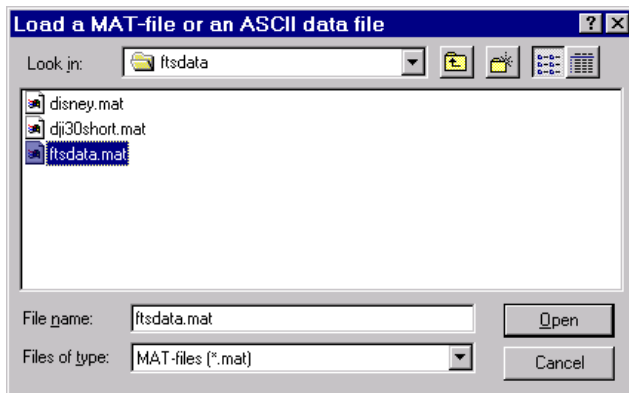
“Saving Time Series Data” on page 14-18

Getting Started

To use the Financial Time Series GUI, first start the financial time series GUI with the command `ftsgui`. Then load (or import) the time series data.

For example, if your data is in a MATLAB MAT-file, select **Load** from the **File** menu.



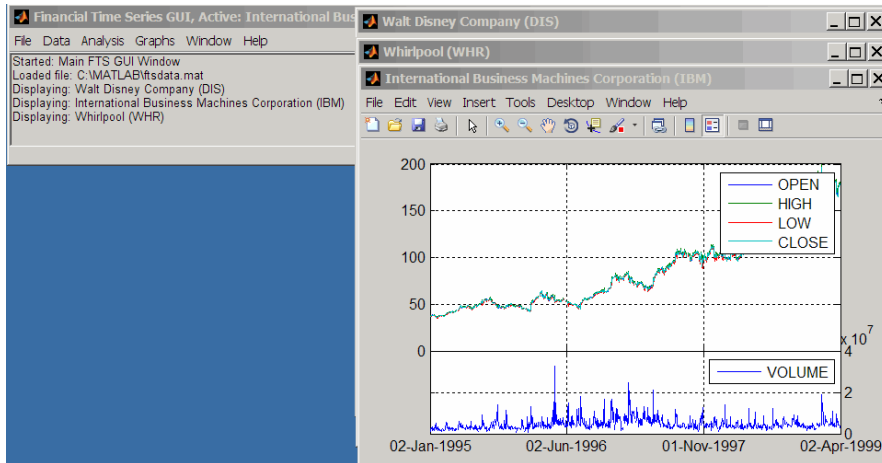


For illustration purposes, choose the file `ftsdata.mat` from the dialog presented.

If you don't see the MAT-file, look in the directory `matlabroot\toolbox\finance\findemos`, where `matlabroot` is the MATLAB root directory (the directory where MATLAB is installed).

Note Data loaded through the Financial Time Series GUI is not available in the MATLAB workspace. You can access this data only through the GUI itself, not with any MATLAB command-line functions.

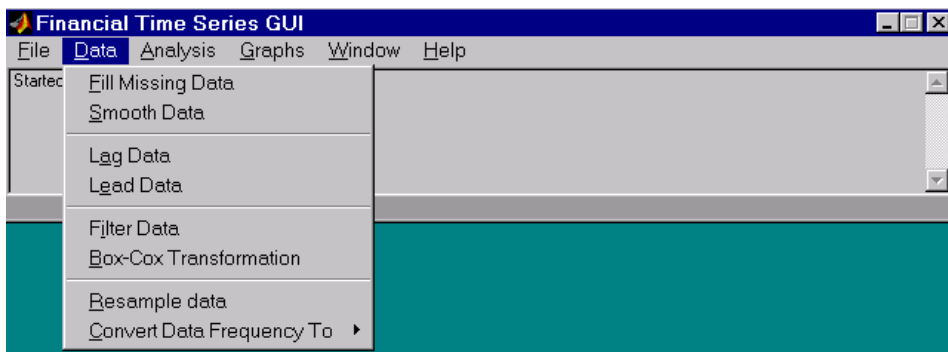
Each financial time series object inside the MAT-file is presented as a line plot in a separate window. The status window is updated accordingly.



Whirlpool (WHR) is the last plot displayed, as indicated on the title bar of the main window.

Data Menu

The **Data** menu provides functions that manipulate time series data.

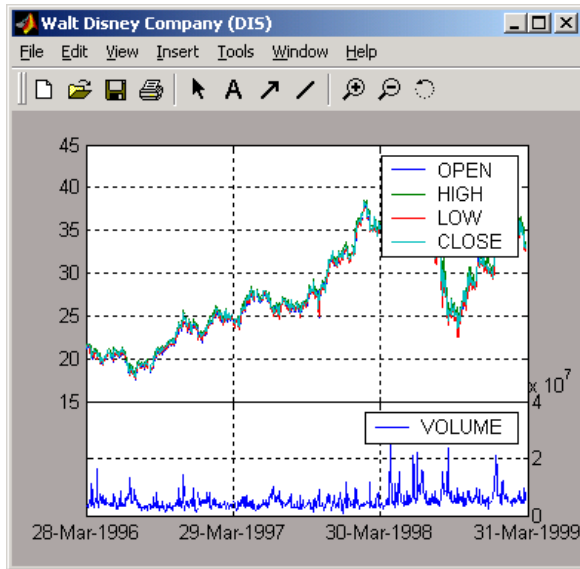


Here are some example tasks that illustrate the use of the functions on this menu.

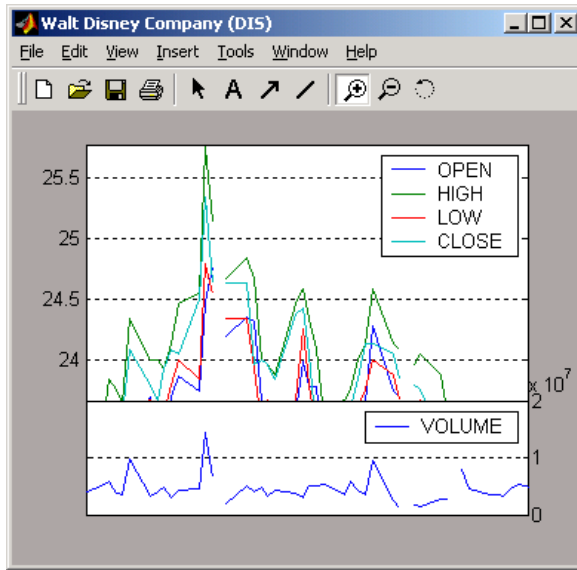
Fill Missing Data

First, look at filling missing data. The **Fill Missing Data** item uses the toolbox function `fillts`. With the data loaded from the file `ftsdata`, you have three time series: IBM

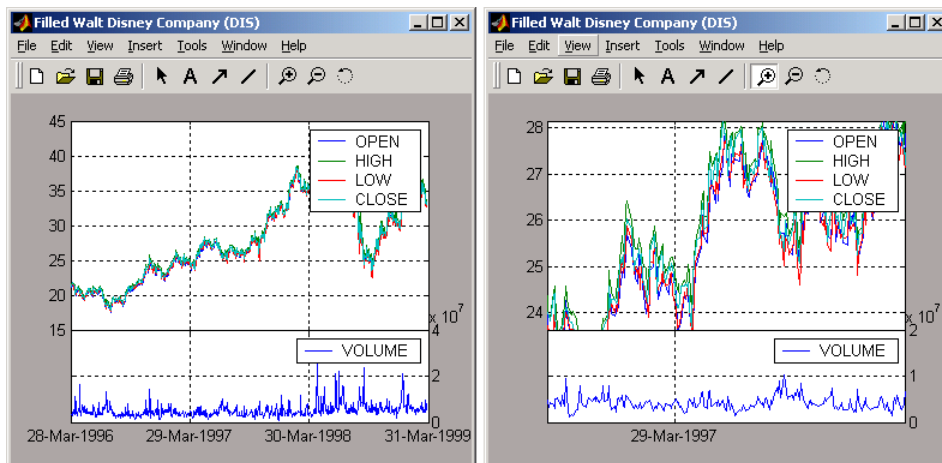
Corp. (IBM), Walt Disney Co. (DIS), and Whirlpool (WHR). Click on the window that shows the time series data for Walt Disney Co. (DIS).



To view any missing data in this time series data set, zoom into the plot using the Zoom tool (the magnifying glass icon with the plus sign) from the toolbar and select a region.



The gaps represent the missing data in the series. To fill these gaps, select **Data > Fill Missing Data**. This selection automatically fills the gaps and generates a new plot that displays the filled time series data.

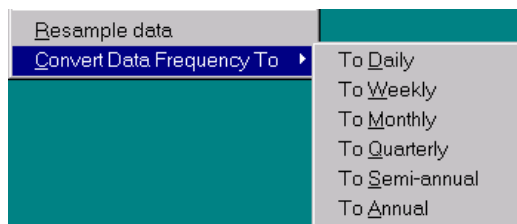


You cannot see the filled gaps when you display the entire data set. However, when you zoom into the plot, you see that the gaps have been eliminated. Note that the title bar

has changed; the title has been prefixed with the word **Filled** to reflect the filled time series data.

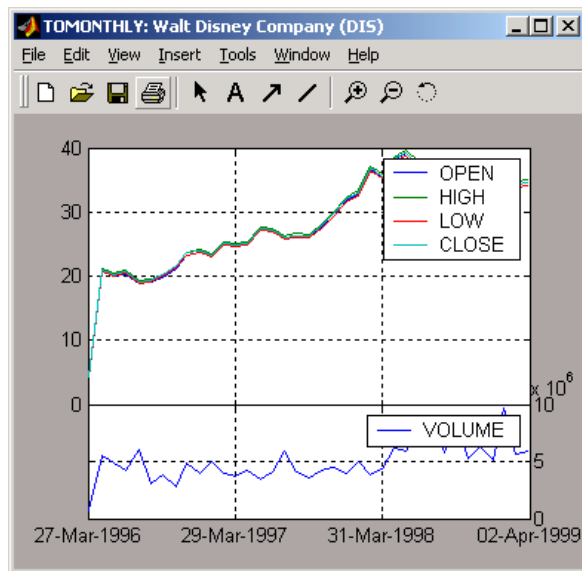
Frequency Conversion

The **Data** menu also provides access to frequency conversion functions.



This example changes the DIS time series data frequency from daily to monthly. Close the Filled Walt Disney Company (DIS) window, and click the Walt Disney Company (DIS) window to make it active (current) again. Then, from the **Data** menu, select **Convert Data Frequency To** and **To Monthly**.

A new figure window displays the result of this conversion.

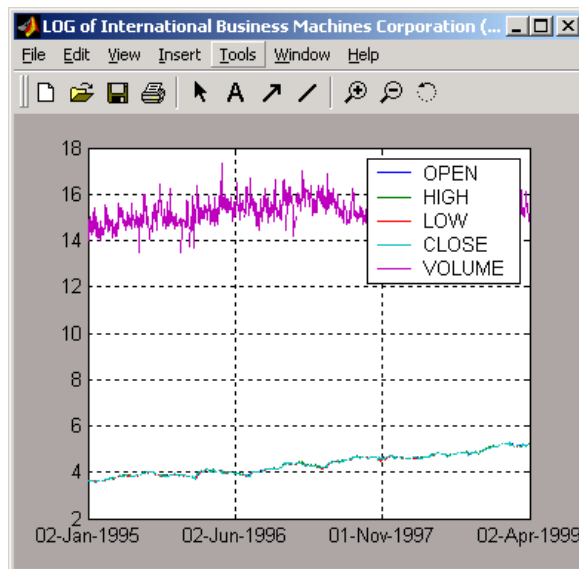


The title reflects that the data displayed had its frequency changed to monthly.

Analysis Menu

The **Analysis** menu provides functions that analyze time series data, including the technical analysis functions. (See “Chart Technical Indicators” for a complete list of the technical analysis functions and several usage examples.)

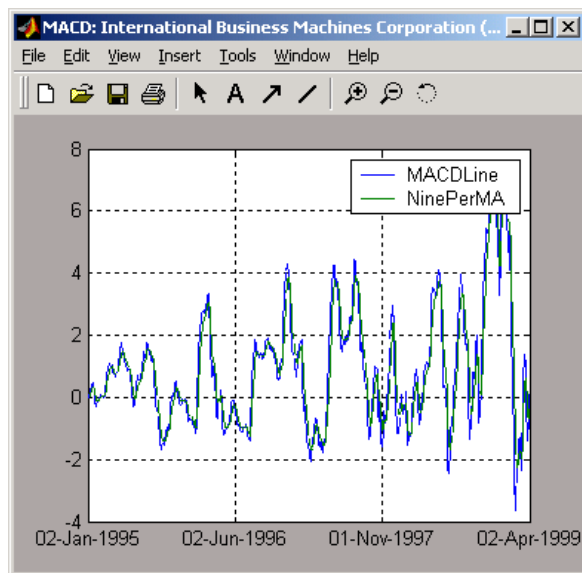
For example, you can use the **Analysis** menu to calculate the natural logarithm (\log) of the data contained within the data set `ftsdata.mat`. This data file provides time series data for IBM (IBM), Walt Disney (DIS), and Whirlpool (WHR). Click the window displaying the data for IBM Corporation (IBM) to make it active (current). Then select the **Analysis** menu, followed by **Log(...)**. The result appears in its own window.



Close the above window and click again on the IBM data window to make it active (current).

Note Before proceeding with any time series analysis, make certain that the title bar confirms that the active data series is the correct one.

From the **Analysis** menu on the main window, select **Technical Analysis** and **MACD**. The result, again, is displayed in its own window.



Other analysis functions work similarly.

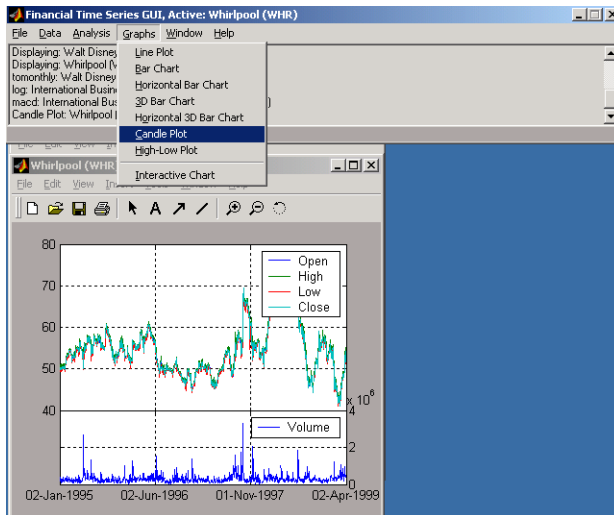
Graphs Menu

The **Graphs** menu displays time series data using the provided graphics functions. Included in the **Graphs** menu are several types of bar charts (`bar`, `barh` and `bar3`, `bar3h`), line plot (`plot`), candle plot (`candle`), and High-Low plot (`highlow`). The **Graphs** menu also provides access to the interactive charting function, `chartfts`.

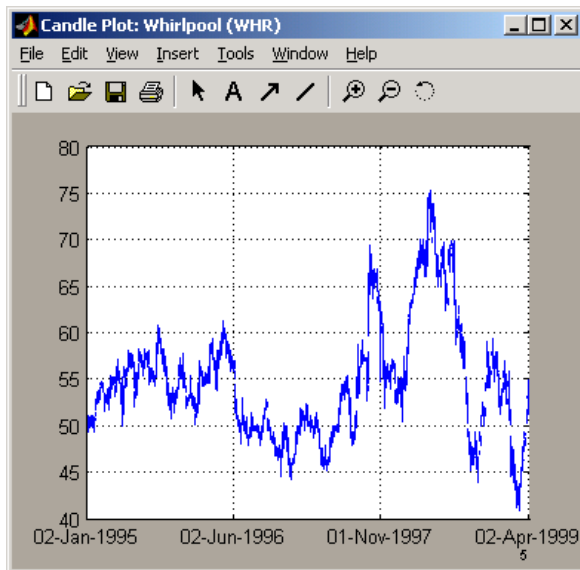
Candle Plot

For example, you can display the candle plot of a set of time series data and start up the interactive chart on the same data set.

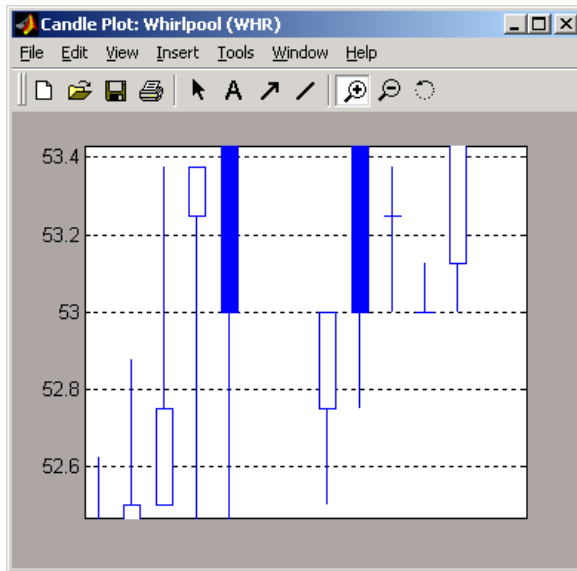
Load the `ftsdata.mat` data set, and click the window that displays the Whirlpool (WHR) time series data to make it active (current). From the main window, select the **Graphs** menu and then **Candle Plot**.



The result is shown below.

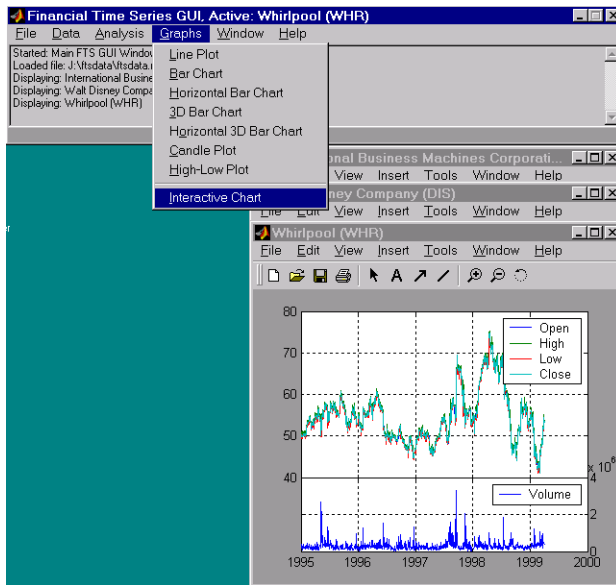


This does not look much like a candle plot because there are too many data points in the data set. All the candles are too compressed for effective viewing. However, when you zoom into a region of this plot, the candles become apparent.

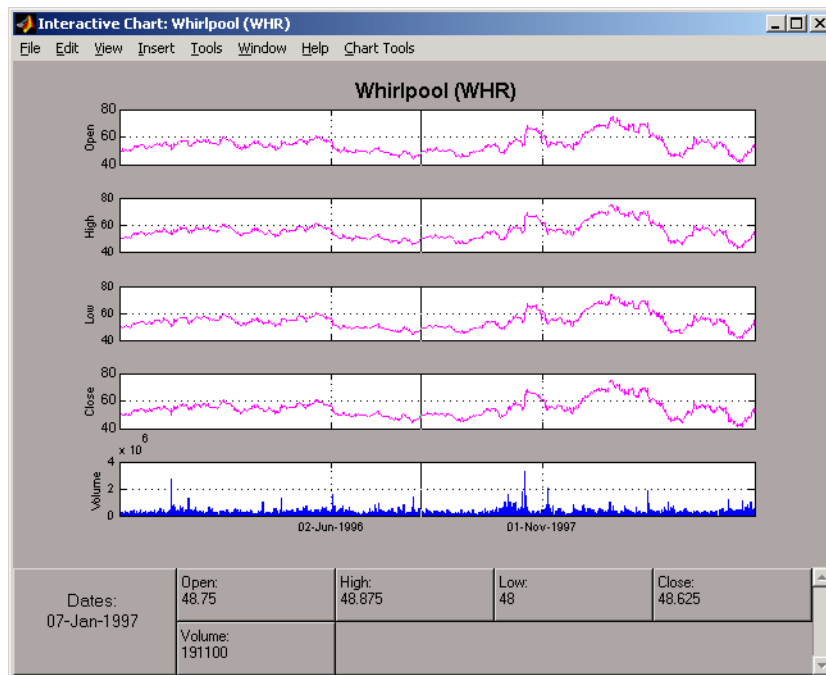


Interactive Chart

To create an interactive chart (`chartfts`) on the Whirlpool data, click the window that displays the Whirlpool (WHR) data to make it active (current). Then, go to the **Graphs** menu and select **Interactive Chart**.



The chart that results is shown below.



You can use this interactive chart as if you had invoked it with the `chartfts` command from the MATLAB command line. For a tutorial on the use of `chartfts`, see “Visualizing Financial Time Series Objects” on page 11-16.

Saving Time Series Data

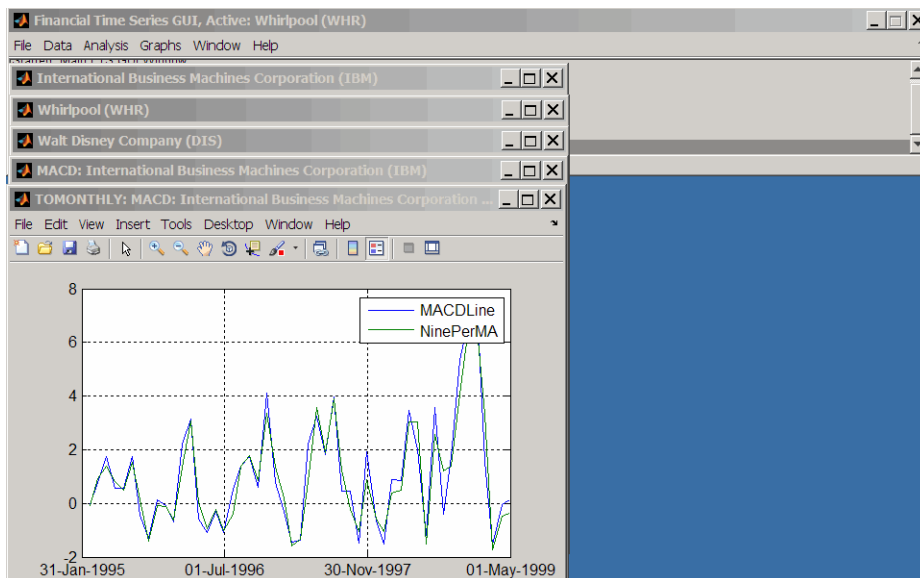
The **Save** and **Save As** items on the main window **File** menu let you save the time series data that results from your analyses and computations. These items save *all* time series data that has been loaded or processed during the current session, even if the window displaying the results of a computation has previously been dismissed.

Note The **Save** and **Save As** items on the **File** menu of the individual plot windows will not save the time series data, but will save the actual plot.

You can save your time series data in two ways:

- Into the latest MAT-file loaded (**Save**)
- Into a MAT-file chosen (or named) from the window (**Save As**)

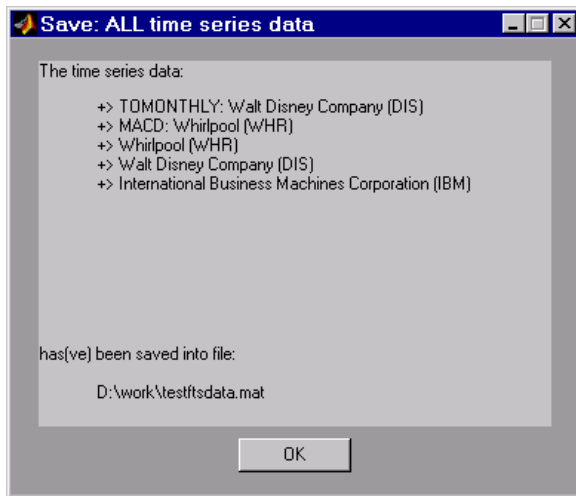
To illustrate this, start by loading the data file `testftstdata.mat` (located in `matlabroot/toolbox/finance/findemos`). Then, convert the Disney (DIS) data from daily (the original frequency) to monthly data. Next, run the MACD analysis on the Whirlpool (WHR) data. You now have a set of five open figure windows.



Saving into the Original File (Save)

To save the data back into the original file (`testftstdata.mat`), select **Save** from the **File** menu.

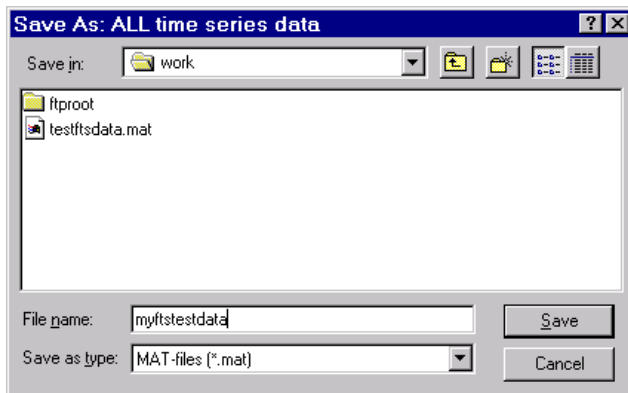
A confirmation window appears. It confirms that the data has been saved in the latest MAT-file loaded (`testftstdata.mat` in this example).



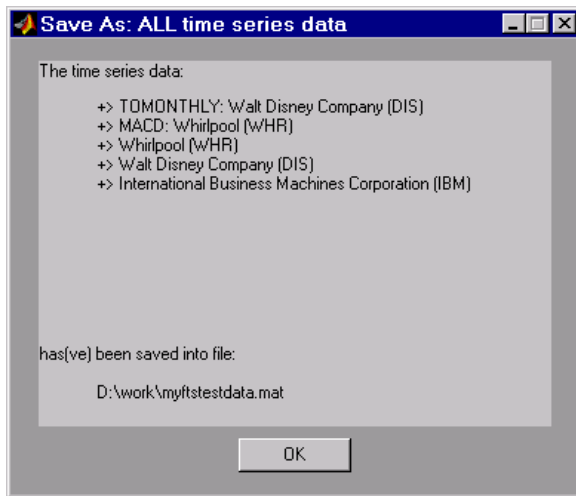
Saving into a New File (Save As)

To save the data in a different file, choose **Save As** from the **File** menu.

The dialog box that appears lets you choose an existing MAT-file from a list or type in the name of a new MAT-file you want to create.



After you click the **Save** button, another confirmation window appears.



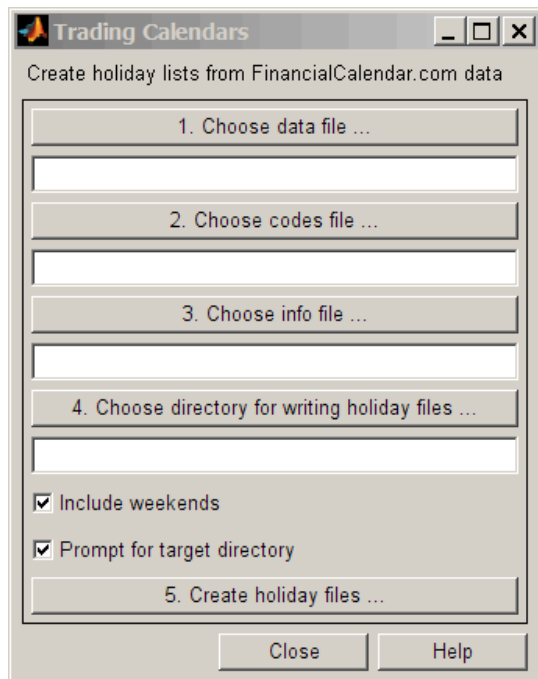
This confirmation window indicates that the data has been saved in a new file named `myftstestdata.mat`.

Trading Date Utilities

- “Trading Calendars Graphical User Interface” on page 15-2
- “UICalendar Graphical User Interface” on page 15-4

Trading Calendars Graphical User Interface

Use the `createholidays` function to open the Trading Calendars graphical user interface.



The `createholidays` function supports <http://www.FinancialCalendar.com> trading calendars. This function can be used from the command line or from the Trading Calendars graphical user interface. To use `createholidays` or the Trading Calendars graphical user interface, you must obtain data, codes, and info files from <http://www.FinancialCalendar.com> trading calendars. For more information on using the command line to programmatically generate the market specific `holidays.m` files without displaying the interface, see `createholidays`.

To use the Trading Calendars graphical user interface:

- 1 From the command line, type the following command to open the Trading Calendars graphical user interface.

createholidays

- 2** Click **Choose data file** to select the data file.
- 3** Click **Choose codes file** to select the codes file.
- 4** Click **Choose info file** to select the info file.
- 5** Click **Choose directory for writing holiday files** to select the output directory.
- 6** Select **Include weekends** to include weekends in the holiday list and click **Prompt for target directory** to be prompted for the file location for each `holidays.m` file that is created.
- 7** Click **Create holiday files** to convert `FinancialCalendar.com` financial center holiday data into market-specific `holidays.m` files.

The market-specific `holidays.m` files can be used in place of the standard `holidays.m` that ships with Financial Toolbox software.

UICalendar Graphical User Interface

In this section...

“Using UICalendar in Standalone Mode” on page 15-4

“Using UICalendar with an Application” on page 15-5

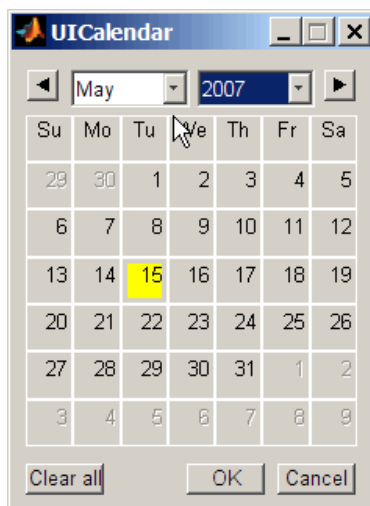
Using UICalendar in Standalone Mode

You can use the UICalendar graphical user interface in standalone mode to look up any date. To use the standalone mode:

- 1 Type the following command to open the UICalendar GUI:

```
uicalendar
```

The UICalendar interface is displayed:



- 2 Click the date and year controls to locate any date.

Using UICalendar with an Application

You can use the UICalendar graphical user interface with an application to look up any date. To use the UICalendar graphical interface with an application, use the following command:

```
uicalendar('PARAM1', VALUE1, 'PARAM2', VALUE2', ...)
```

For more information, see `uicalendar`.

Example of Using UICalendar with an Application

The UICalendar example creates a function that displays a graphical user interface that lets you select a date from the UICalendar graphical user interface and fill in a text field with that date.

- 1 Create a figure.

```
function uicalendarGUIExample
f = figure('Name', 'uicalendarGUIExample');
```

- 2 Add a text control field.

```
dateTextHandle = uicontrol(f, 'Style', 'Text', ...
    'String', 'Date:', ...
    'HorizontalAlignment', 'left', ...
    'Position', [100 200 50 20]);
```

- 3 Add a uicontrol editable text field to display the selected date.

```
dateEditBoxHandle = uicontrol(f, 'Style', 'Edit', ...
    'Position', [140 200 100 20], ...
    'BackgroundColor', 'w');
```

- 4 Create a push button that startups the UICalendar.

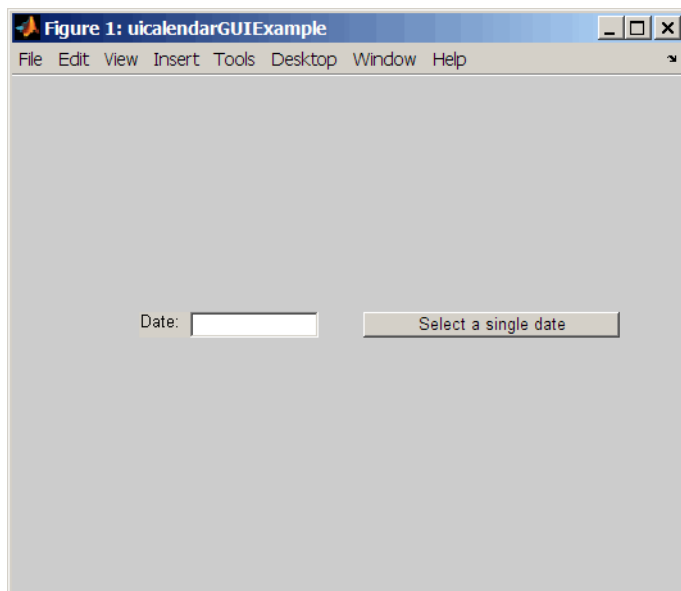
```
calendarButtonHandle = uicontrol(f, 'Style', 'PushButton', ...
    'String', 'Select a single date', ...
    'Position', [275 200 200 20], ...
    'callback', @pushbutton_cb);
```

- 5 To startup UICalendar, create a nested function (callback function) for the push button.

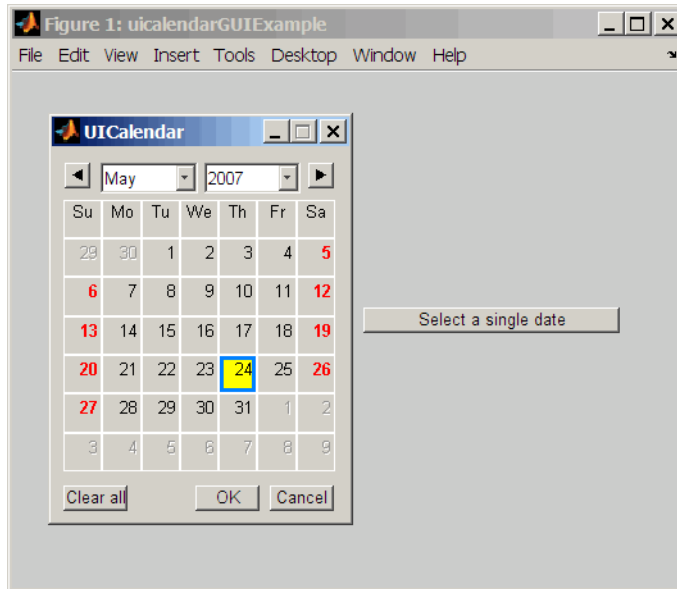
```
function pushbutton_cb(hcbo, eventStruct)
% Create a UICALENDAR with the following properties:
% 1) Highlight weekend dates.
% 2) Only allow a single date to be selected at a time.
% 3) Send the selected date to the edit box uicontrol.
```

```
uicalendar('Weekend', [1 0 0 0 0 0 1], ...  
'SelectionType', 1, ...  
'DestinationUI', dateEditBoxHandle);  
end  
end
```

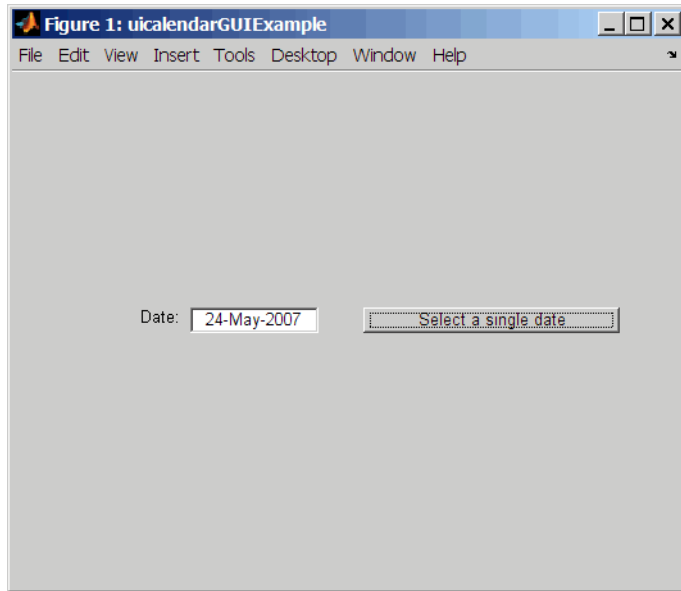
- 6 Run the function `uicalendarGUIExample` to display the application interface:



- 7 Click **Select a single date** to display the UICalendar graphical user interface:



- 8 Select a date and click **OK** to display the date in the text field:



Technical Analysis

- “Technical Indicators” on page 16-2
- “Technical Analysis Examples” on page 16-4

Technical Indicators

Technical analysis (or charting) is used by some investment managers to help manage portfolios. Technical analysis relies heavily on the availability of historical data. Investment managers calculate different indicators from available data and plot them as charts. Observations of price, direction, and volume on the charts assist managers in making decisions on their investment portfolios.

The technical analysis functions in this toolbox are tools to help analyze your investments. The functions in themselves will not make any suggestions or perform any qualitative analysis of your investment.

Technical Analysis: Oscillators

Function	Type
adosc	Accumulation/distribution oscillator
chaikosc	Chaikin oscillator
macd	Moving Average Convergence/Divergence
stochosc	Stochastic oscillator
tsaccel	Acceleration
tsmom	Momentum

Technical Analysis: Stochastics

Function	Type
chaikvolat	Chaikin volatility
fpctkd	Fast stochastics
spctkd	Slow stochastics
willpctr	Williams %R

Technical Analysis: Indexes

Function	Type
negvalidx	Negative volume index
posvalidx	Positive volume index
rsindex	Relative strength index

Technical Analysis: Indicators

Function	Type
adline	Accumulation/distribution line
bollinger	Bollinger band
hhigh	Highest high
llow	Lowest low
medprice	Median price
onbalvol	On balance volume
prcroc	Price rate of change
pvtrend	Price-volume trend
typprice	Typical price
volroc	Volume rate of change
wclose	Weighted close
willad	Williams accumulation/distribution

Technical Analysis Examples

In this section...

“Overview” on page 16-4

“Moving Average Convergence/Divergence (MACD)” on page 16-4

“Williams %R” on page 16-5

“Relative Strength Index (RSI)” on page 16-7

“On-Balance Volume (OBV)” on page 16-8

Overview

To illustrate some of the technical analysis functions, this section uses the IBM stock price data contained in the supplied file `ibm9599.dat`. First create a financial time series object from the data using `ascii2fts`:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
```

The time series data contains the open, close, high, and low prices, as well as the volume traded on each day. The time series dates start on January 3, 1995, and end on April 1, 1999, with some values missing for weekday holidays; weekend dates are not included.

Moving Average Convergence/Divergence (MACD)

Moving Average Convergence/Divergence (MACD) is an oscillator function used by technical analysts to spot overbought and oversold conditions. Look at the portion of the time series covering the 3-month period between October 1, 1995 and December 31, 1995. At the same time fill any missing values due to holidays within the time period specified:

```
part_ibm = fillts(ibm('10/01/95::12/31/95'));
```

Now calculate the MACD, which when plotted produces two lines; the first line is the MACD line itself and the second is the nine-period moving average line:

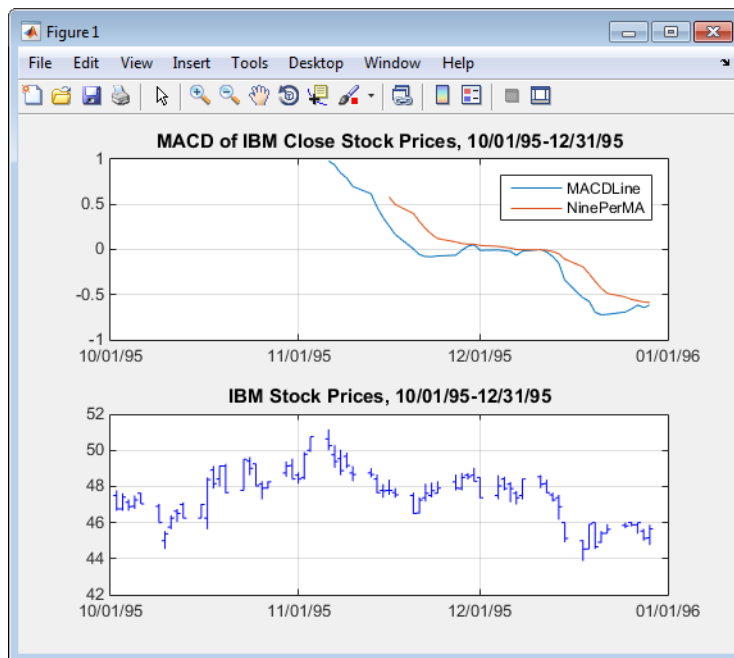
```
macd_ibm = macd(part_ibm);
```

Note When you call `macd` without giving it a second input argument to specify a particular data series name, it searches for a closing price series named `Close` (in all combinations of letter cases).

Plot the MACD lines and the High-Low plot of the IBM stock prices in two separate plots in one window.

```
subplot(2, 1, 1);
plot(macd_ibm);
title('MACD of IBM Close Stock Prices, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
subplot(2, 1, 2);
highlow(part_ibm);
title('IBM Stock Prices, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy')
```

The following figure shows the result.



Williams %R

Williams %R is an indicator that measures overbought and oversold levels. The function `willpctr` is from the `stochastics` category. All the technical analysis functions can

accept a different name for a required data series. If, for example, a function needs the high, low, and closing price series but your time series object does not have the data series names exactly as `High`, `Low`, and `Close`, you can specify the correct names as follows.

```
wpr = willpctr(tsobj, 14, 'HighName', 'Hi', 'LowName', 'Lo',...  
'CloseName', 'Closing')
```

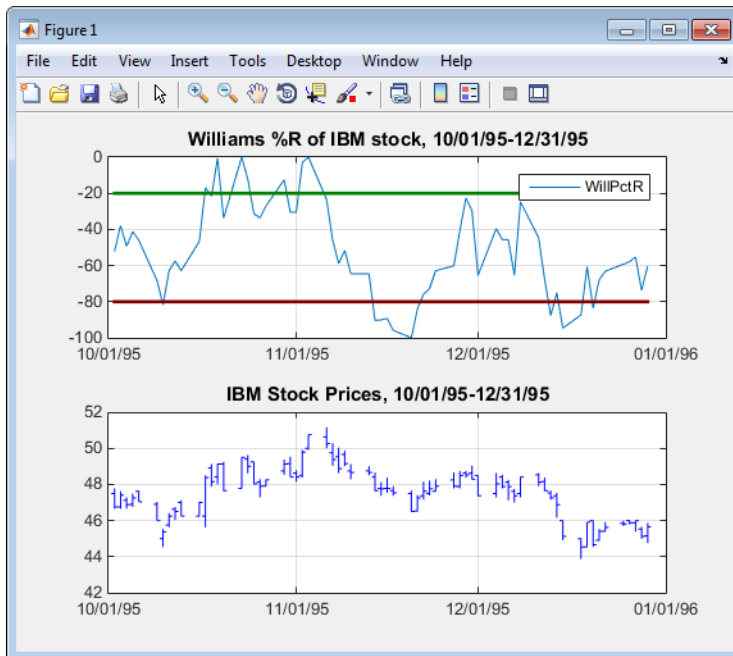
The function `willpctr` now assumes that your high price series is named `Hi`, low price series is named `Lo`, and closing price series is named `Closing`.

Since the time series object `part_ibm` has its data series names identical to the required names, name adjustments are not needed. The input argument to the function is only the name of the time series object itself.

Calculate and plot the Williams %R indicator for IBM stock along with the price range using these commands:

```
wpctr_ibm = willpctr(part_ibm);  
subplot(2, 1, 1);  
plot(wpctr_ibm);  
title('Williams %R of IBM stock, 10/01/95-12/31/95');  
datetick('x', 'mm/dd/yy');  
hold on;  
plot(wpctr_ibm.dates, -80*ones(1, length(wpctr_ibm)),...  
'color', [0.5 0 0], 'linewidth', 2)  
plot(wpctr_ibm.dates, -20*ones(1, length(wpctr_ibm)),...  
'color', [0 0.5 0], 'linewidth', 2)  
subplot(2, 1, 2);  
highlow(part_ibm);  
title('IBM Stock Prices, 10/01/95-12/31/95');  
datetick('x', 'mm/dd/yy');
```

The next figure shows the results. The top plot has the Williams %R line plus two lines at -20% and -80%. The bottom plot is the High-Low plot of the IBM stock price for the corresponding time period.



Relative Strength Index (RSI)

The Relative Strength Index (RSI) is a momentum indicator that measures an equity's price relative to itself and its past performance. The function name is `rsindex`.

The `rsindex` function needs a series that contains the closing price of a stock. The default period length for the RSI calculation is 14 periods. This length can be changed by providing a second input argument to the function. Similar to the previous commands, if your closing price series is not named `Close`, you can provide the correct name.

Calculate and plot the RSI for IBM stock along with the price range using these commands:

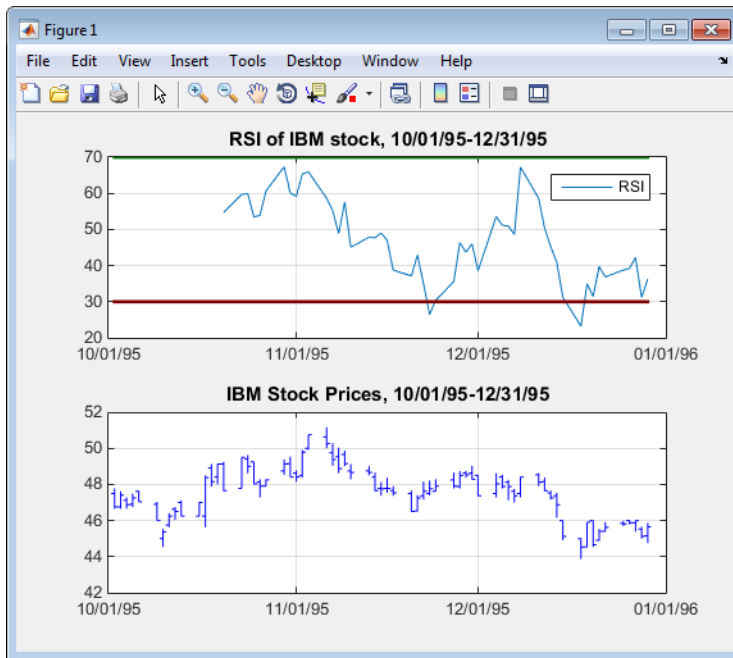
```
rsi_ibm = rsindex(part_ibm);
subplot(2, 1, 1);
plot(rsi_ibm);
title('RSI of IBM stock, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
hold on;
```

```

plot(rsi_ibm.dates, 30*ones(1, length(wpctr_ibm)),...
'color', [0.5 0 0], 'linewidth', 2)
plot(rsi_ibm.dates, 70*ones(1, length(wpctr_ibm)),...
'color', [0 0.5 0], 'linewidth', 2)
subplot(2, 1, 2);
highlow(part_ibm);
title('IBM Stock Prices, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');

```

The next figure shows the result.



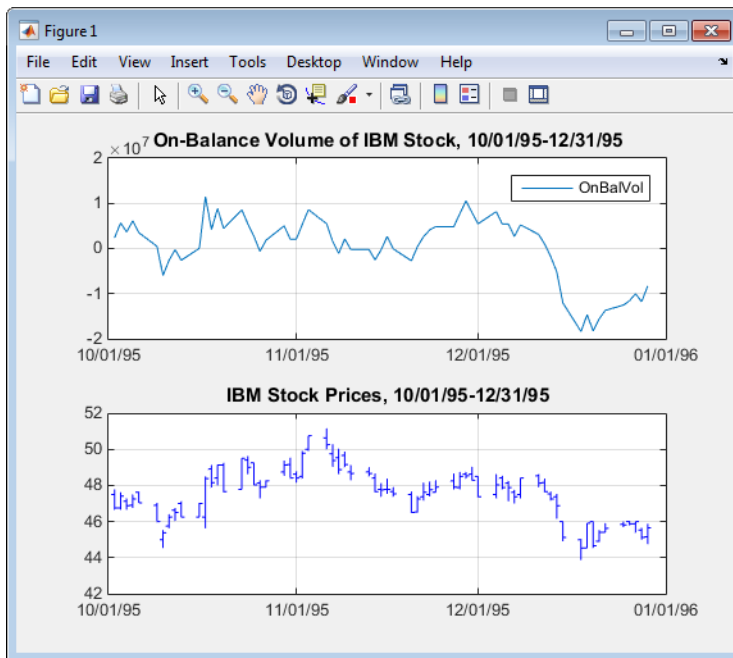
On-Balance Volume (OBV)

On-Balance Volume (OBV) relates volume to price change. The function `onbalvol` requires you to have the closing price (`Close`) series as well as the volume traded (`Volume`) series.

Calculate and plot the OBV for IBM stock along with the price range using these commands:


```
obv_ibm = onbalvol(part_ibm);  
subplot(2, 1, 1);  
plot(obv_ibm);  
title('On-Balance Volume of IBM Stock, 10/01/95-12/31/95');  
datetick('x', 'mm/dd/yy');  
subplot(2, 1, 2);  
highlow(part_ibm);  
title('IBM Stock Prices, 10/01/95-12/31/95');  
datetick('x', 'mm/dd/yy');
```

The next figure shows the result.



Stochastic Differential Equations

- “SDEs” on page 17-2
- “SDE Class Hierarchy” on page 17-5
- “SDE Models” on page 17-7
- “Base SDE Models” on page 17-15
- “Drift and Diffusion Models” on page 17-18
- “Linear Drift Models” on page 17-22
- “Parametric Models” on page 17-24
- “Simulating Equity Prices” on page 17-33
- “Simulating Interest Rates” on page 17-59
- “Stratified Sampling” on page 17-70
- “Performance Considerations” on page 17-75
- “Pricing American Basket Options by Monte Carlo Simulation” on page 17-82

SDEs

In this section...
“SDE Modeling” on page 17-2
“Trials vs. Paths” on page 17-3
“NTRIALS, NPERIODS, and NSTEPS” on page 17-3

SDE Modeling

Financial 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.

The following table lists tasks you can perform using the SDE functionality.

To perform this task ...	Use these types of models ...
“Simulating Equity Prices” on page 17-33	<ul style="list-style-type: none"> • Geometric Brownian Motion (GBM) • Constant Elasticity of Variance (CEV) • Stochastic Differential Equation (SDE) • Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) • Stochastic Differential Equations from Linear Drift (SDELD) • Heston Stochastic Volatility (Heston)
“Simulating Interest Rates” on page 17-59	<ul style="list-style-type: none"> • Hull-White-Vasicek (HWV) • Cox-Ingersoll-Ross (CIR) • Stochastic Differential Equation (SDE) • Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) • Stochastic Differential Equations from Mean-Reverting Drift (SDEM RD) Models

To perform this task ...	Use these types of models ...
“Pricing Equity Options” on page 17-56	Geometric Brownian Motion (GBM)
“Stratified Sampling” on page 17-70	All supported models
“Performance Considerations” on page 17-75	All supported models

Trials vs. Paths

Monte Carlo simulation literature often uses different terminology for the evolution of the simulated variables of interest, such as *trials* and *paths*. 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 pairwise dependence induced by *antithetic sampling*, and applies to more sophisticated variance reduction techniques, such as *stratified sampling*.

NTRIALS, NPERIODS, and NSTEPS

SDE methods in the Financial 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 three-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: About Solution Precision and Error” on page 17-77 and “Managing Memory” on page 17-75.

SDE Class Hierarchy

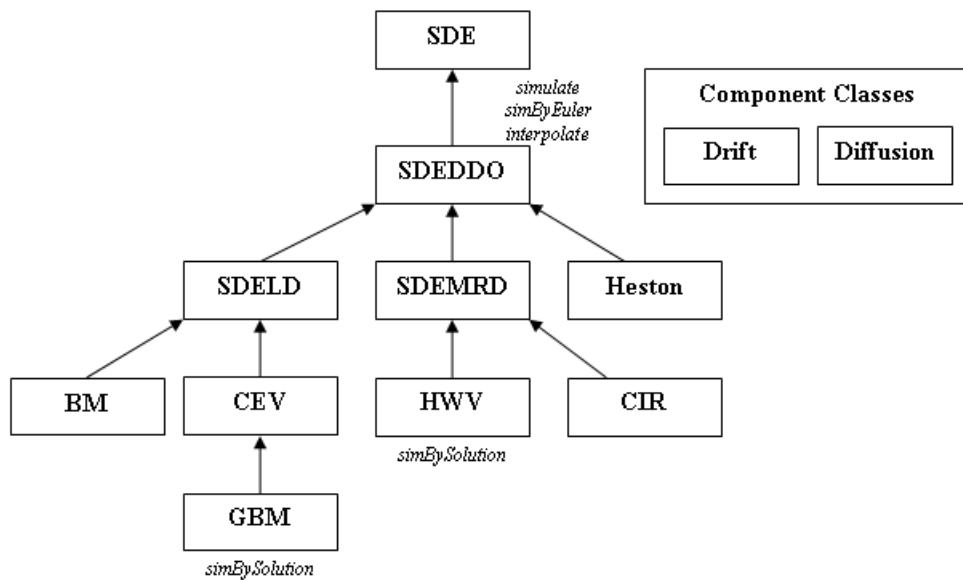
The Financial Toolbox SDE class structure represents a generalization and specialization hierarchy. The top-level class provides the most general model interface and offers the default Monte Carlo simulation and interpolation methods. In turn, derived classes offer restricted interfaces that simplify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

The following table lists the SDE classes. The introductory examples in “Available Models” on page 17-13 show how to use these classes to create objects associated with univariate models. Although the Financial 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	“Base SDE Models” on page 17-15
Drift, Diffusion	“Overview” on page 17-18
SDEDDO	“Drift and Diffusion Models” on page 17-18
SDELD	“Linear Drift Models” on page 17-22
CEV	“Creating Constant Elasticity of Variance (CEV) Models” on page 17-25
BM	“Creating Brownian Motion (BM) Models” on page 17-24
SDEM RD	“Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEM RD) Models” on page 17-27
GBM	“Creating Geometric Brownian Motion (GBM) Models” on page 17-26
HWV	“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 17-29
CIR	“Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models” on page 17-28
Heston	“Creating Heston Stochastic Volatility Models” on page 17-31

The following figure illustrates the inheritance relationships among SDE classes.



SDE Models

In this section...

“Introduction” on page 17-7
“Creating SDE Objects” on page 17-7
“Drift and Diffusion” on page 17-12
“Available Models” on page 17-13
“SDE Methods” on page 17-14

Introduction

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 “Working with Objects in Functions” in the MATLAB documentation.

Creating SDE Objects

- “Constructing Objects” on page 17-8
- “Displaying Objects” on page 17-8
- “Assigning and Referencing Object Parameters” on page 17-8
- “Constructing and Evaluating Models” on page 17-8
- “Specifying SDE Simulation Parameters” on page 17-9

Constructing Objects

You use *constructors* to create SDE objects.

For examples and more information, see:

- “Available Models” on page 17-13
- “Simulating Equity Prices” on page 17-33
- “Simulating Interest Rates” on page 17-59

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: Univariate GBM Models” on page 17-27.

Specifying SDE Simulation Parameters

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.

Consider the following general stochastic differential equation:

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

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.

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 17-1 is

useful in implementing derived classes that impose additional structure on the drift and diffusion-rate functions.

Specifying User-defined Functions as Model Parameters

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.

Model parameters that are specified as functions are evaluated in the same way as user-specified random number (noise) generation functions. (For more information, see “Evaluating Different Types of Functions” on page 17-11.) Model parameters that are specified as functions are inputs to remove object constructors. User-specified noise and processing functions are *optional* inputs to simulation methods.

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.

Evaluating Different Types of 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 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:

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

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.

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 “Pricing Equity Options” on page 17-56.

Therefore, all simulation methods evaluate noise generation functions as:

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

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

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

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

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

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.

Drift and Diffusion

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

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

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:

$$F(t, X_t) = S(t)[L(t) - X_t]$$

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:

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

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 terminology 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.

Available Models

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

SDE 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)	$dX_t = B(t)X_t dt + V(t)X_t^{\alpha(t)} dW_t$
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)	$dX_t = S(t)(L(t) - X_t)dt + V(t)dW_t$
Heston	$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$ $dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$

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` property
- `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.

Base SDE Models

In this section...

“Overview” on page 17-15

“Example: Base SDE Models” on page 17-15

Overview

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 the following inputs:

- A drift-rate function **F**. This function 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**. This function 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: Base SDE Models

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

$$dX_t = 0.1X_t dt + 0.3X_t dW_t$$

- 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.

Drift and Diffusion Models

In this section...

“Overview” on page 17-18

“Example: Drift and Diffusion Rates” on page 17-19

“Example: SDEDDO Models” on page 17-20

Overview

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 input parameters.

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: Drift and Diffusion Rates

In this example, you create `drift` and `diffusion` rate objects to create the same model as in “Example: Base SDE Models” on page 17-15.

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.

Example: 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.

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.

Linear Drift Models

In this section...

“Overview” on page 17-22

“Example: SDELD Models” on page 17-22

Overview

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:

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

SDELD objects provide a parametric alternative to the mean-reverting drift form, as discussed in “Example: SDEMRD Models” on page 17-28. 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: SDELD Models

Create the same model as in “Example: Base SDE Models” on page 17-15:

```
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

Parametric Models

In this section...

“Creating Brownian Motion (BM) Models” on page 17-24

“Example: BM Models” on page 17-24

“Creating Constant Elasticity of Variance (CEV) Models” on page 17-25

“Creating Geometric Brownian Motion (GBM) Models” on page 17-26

“Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEM RD) Models” on page 17-27

“Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models” on page 17-28

“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 17-29

“Creating Heston Stochastic Volatility Models” on page 17-31

Creating Brownian Motion (BM) Models

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

$$dX_t = \mu(t)dt + V(t)dW_t$$

Example: 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. This specialized method is invoked automatically 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 three-dimensional array.
- If **Z** is unspecified, the assumed Gaussian correlation structure is a double matrix.

Creating Constant Elasticity of Variance (CEV) Models

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

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

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

Example: 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

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

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$$

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. This specialized method is invoked automatically 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 three-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 piecewise constant parameters. If the model parameters are piecewise constant over each observation period, the state vector X_t is lognormally distributed and the simulated process is exact for the observation times at which X_t is sampled.

Example: 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) 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:

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

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: 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

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

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

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

Example: CIR Models

Create a CIR object to represent the same model as in “Example: SDEM RD Models” on page 17-28:

```
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

The Hull-White/Vasicek(HWV) short rate class derives directly from SDE with mean-reverting drift (that is, SDEM RD):

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$$

Example: 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 = hww(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 piecewise 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 piecewise constant parameters. If $S(t, X_t)$, $L(t, X_t)$, and $V(t, X_t)$ are piecewise 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.

Creating Heston Stochastic Volatility Models

The Heston (`heston`) class derives directly from SDE from Drift and Diffusion (SDEDDO). Each Heston model is a bivariate composite model, consisting of two coupled univariate models:

$$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

$$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

Equation 17-5 is typically associated with a price process. Equation 17-6 represents the evolution of the price process' variance. Models of type `heston` are typically used to price equity options.

Example: Heston Models

Create a `heston` object to represent the model:

$$dX_{1t} = 0.1X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

$$dX_{2t} = 0.2[0.1 - X_{2t}]dt + 0.05\sqrt{X_{2t}}dW_{2t}$$

```
obj = heston (0.1, 0.2, 0.1, 0.05)
```

```
obj =
```

```
Class HESTON: Heston Bivariate Stochastic Volatility
-----
Dimensions: State = 2, Brownian = 2
-----
StartTime: 0
StartState: 1 (2x1 double array)
Correlation: 2x2 diagonal double array
```

Drift: drift rate function $F(t, X(t))$
Diffusion: diffusion rate function $G(t, X(t))$
Simulation: simulation method/function `simByEuler`
Return: 0.1
Speed: 0.2
Level: 0.1
Volatility: 0.05

Simulating Equity Prices

In this section...

“Simulating Multidimensional Market Models” on page 17-33

“Inducing Dependence and Correlation” on page 17-48

“Dynamic Behavior of Market Parameters” on page 17-50

“Pricing Equity Options” on page 17-56

Simulating Multidimensional 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:

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

where:

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

Representing Market Models Using SDE Objects

Create an SDE object to represent the equity market model.

- 1 Load the `Data_GlobalIdx2` data set:

```
load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
Dataset.NIK Dataset.FTSE Dataset.SP];
```

- 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);
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 17-7:

```
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.

Representing Market Models Using SDEDDO Objects

Create an SDEDDO object to represent the market model in Equation 17-7:

- 1 Load the `Data_GlobalIdx2` data set:

```
load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
Dataset.NIK Dataset.FTSE Dataset.SP];
```

- 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);
```

- 4 Create drift and diffusion objects:

```
F = drift(zeros(nVariables,1), diag(expReturn))
G = diffusion(ones(nVariables,1), diag(sigma))
```

```
F =
```

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

```
G =
```

```
Class DIFFUSION: Diffusion Rate Specification
-----
Rate: diffusion rate function G(t,X(t))
Alpha: 6x1 double array
Sigma: 6x6 diagonal double array
```

- 5 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 “Representing Market Models Using SDE Objects” on page 17-33. 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.

Representing Market Models Using SDELD, CEV, and GBM Objects

Create `SDELD`, `CEV`, and `GBM` objects to represent the market model in Equation 17-7.

- 1 Load the `Data_GlobalIdx2` data set:

```

load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
         Dataset.NIK Dataset.FTSE Dataset.SP];

```

- 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);
t          = 0;
X          = 100;
X          = X(ones(nVariables,1));
```

4 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
```

5 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
```

6 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.

Simulating Equity Markets Using the Default Simulate Method

- 1 Load the `Data_GlobalIdx2` data set and specify the SDE model as in “Representing Market Models Using SDE Objects” on page 17-33.

```
load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
         Dataset.NIK Dataset.FTSE Dataset.SP];

returns = price2ret(prices);

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

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

SDE = sde(F, G, 'Correlation', ...
         correlation, 'StartState', X);
```

- 2 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
rng(142857, 'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt);
```

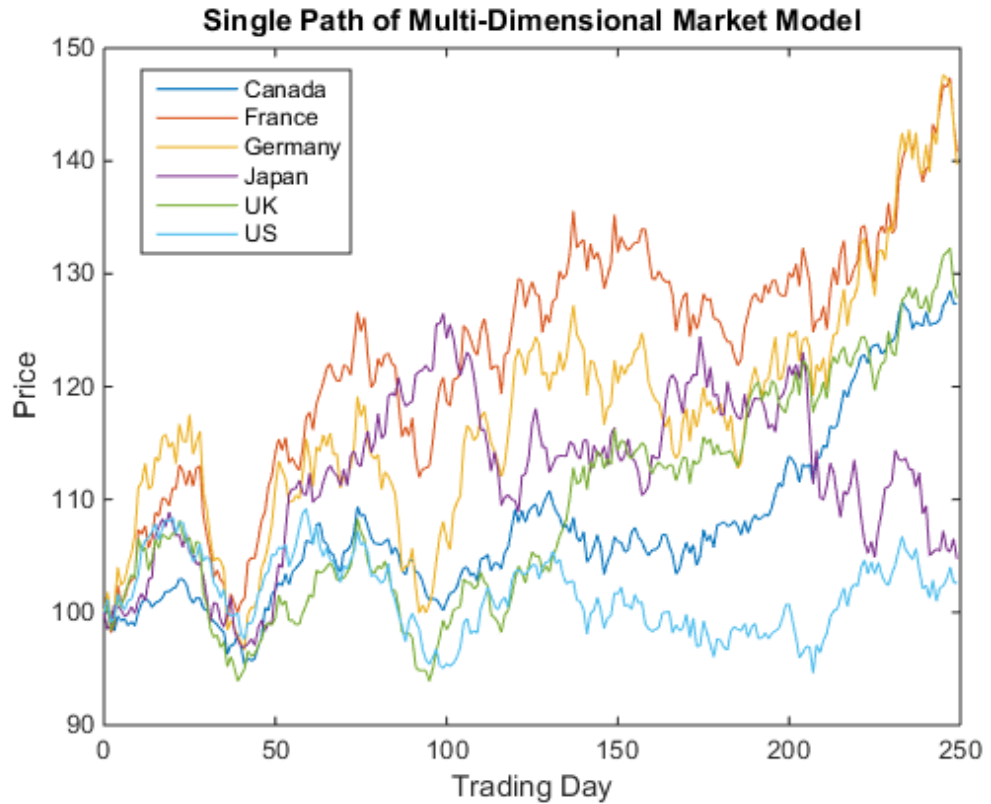
```
whos S
```

Name	Size	Bytes	Class	Attributes
S	250x6	12000	double	

The output array S is a $250\text{-by-}6 = (\text{NPERIODS} + 1)\text{-by-}n\text{Variables-by-}1$ 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 .

- 3 Plot the simulated paths.

```
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')
```



Simulating Equity Markets Using the SimByEuler Method

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 “Simulating Equity Markets Using the Default Simulate Method” on page 17-39:

- 1 Load the `Data_GlobalIdx2` data set and specify the SDE model as in “Representing Market Models Using SDE Objects” on page 17-33.

```
load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
         Dataset.NIK Dataset.FTSE Dataset.SP];

returns = price2ret(prices);

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

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

SDE = sde(F, G, 'Correlation', ...
          correlation, 'StartState', X);
```

- 2 Simulate a single path using `simByEuler`.

```
nPeriods = 249;      % # of simulated observations
dt        = 1;       % time increment = 1 day
rng(142857,'twister')
[S,T] = simByEuler(SDE, nPeriods, 'DeltaTime', dt);
```

- 3 Simulate 10 trials with the same initial conditions, and examine `S`:

```
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);
```

whos `S`

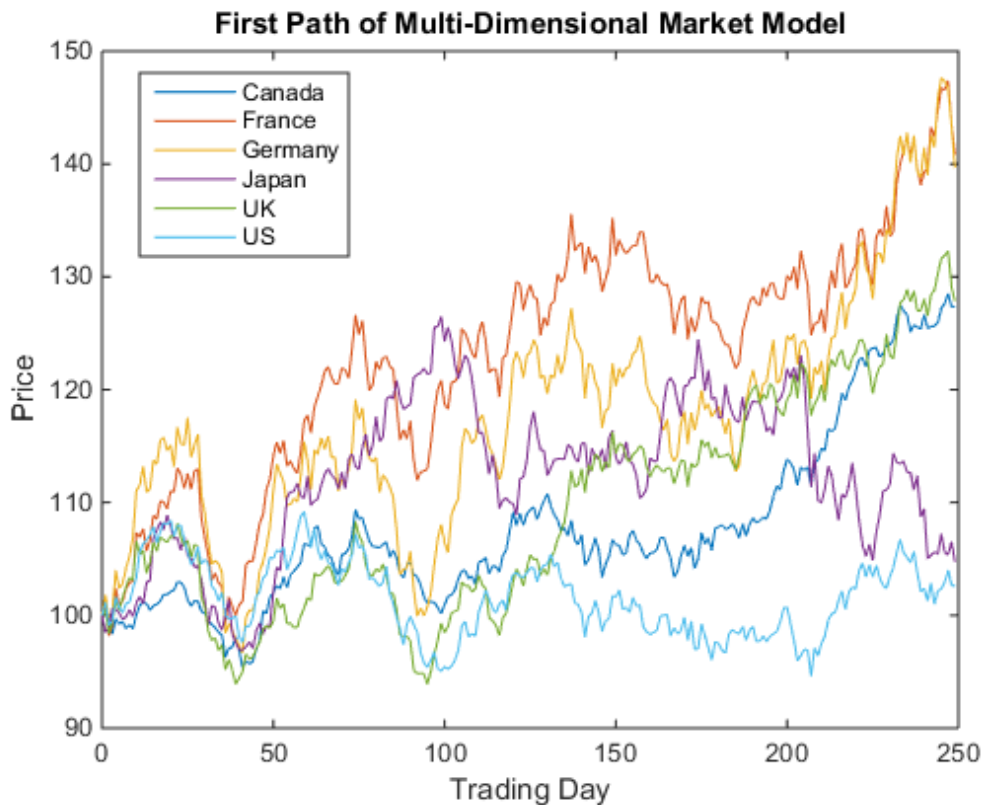
Name	Size	Bytes	Class	Attributes
------	------	-------	-------	------------

```
S          250x6x10          120000 double
```

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

4 Plot the first paths.

```
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')
```



The first realization of S is identical to the paths in the plot.

Simulating Equity Markets 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 Load the `Data_GlobalIdx2` data set and specify the SDE model as in “Representing Market Models Using SDE Objects” on page 17-33, and the GBM model as in “Representing Market Models Using SDELD, CEV, and GBM Objects” on page 17-36.

```
load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
         Dataset.NIK Dataset.FTSE Dataset.SP];

returns = price2ret(prices);

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

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

SDE = sde(F, G, 'Correlation', ...
          correlation, 'StartState', X);

GBM = gbm(diag(expReturn),diag(sigma), 'Correlation', ...
          correlation, 'StartState', X);
```

- 2 To illustrate the performance benefit of the overloaded Euler approximation method, increase the number of trials to 10000.

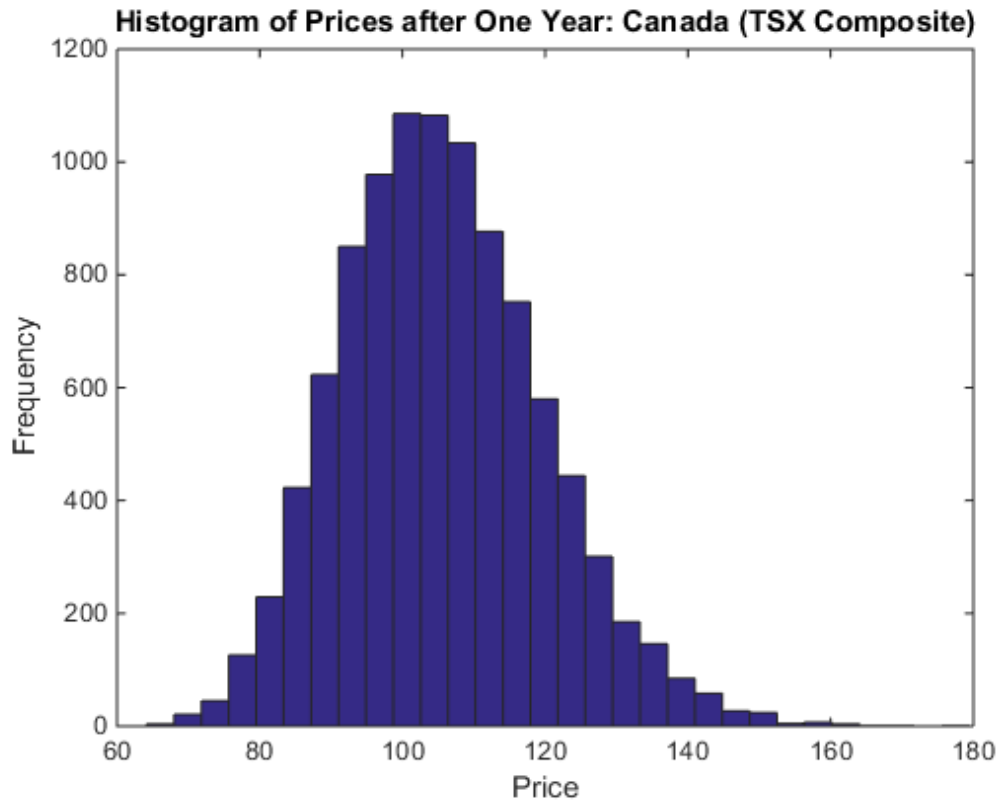
```
nPeriods = 249;      % # of simulated observations
dt        = 1;       % time increment = 1 day
rng(142857,'twister')
[X,T] = simulate(GBM, nPeriods, 'DeltaTime', dt, ...
```

```
        'nTrials', 10000);  
  
whos X  
  
Name          Size          Bytes  Class  Attributes  
X             250x6x10000    120000000  double
```

The output X is a much larger time series array.

- 3 Using this sample size, examine the terminal distribution of Canada's TSX Composite to verify qualitatively the lognormal character of the data.

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

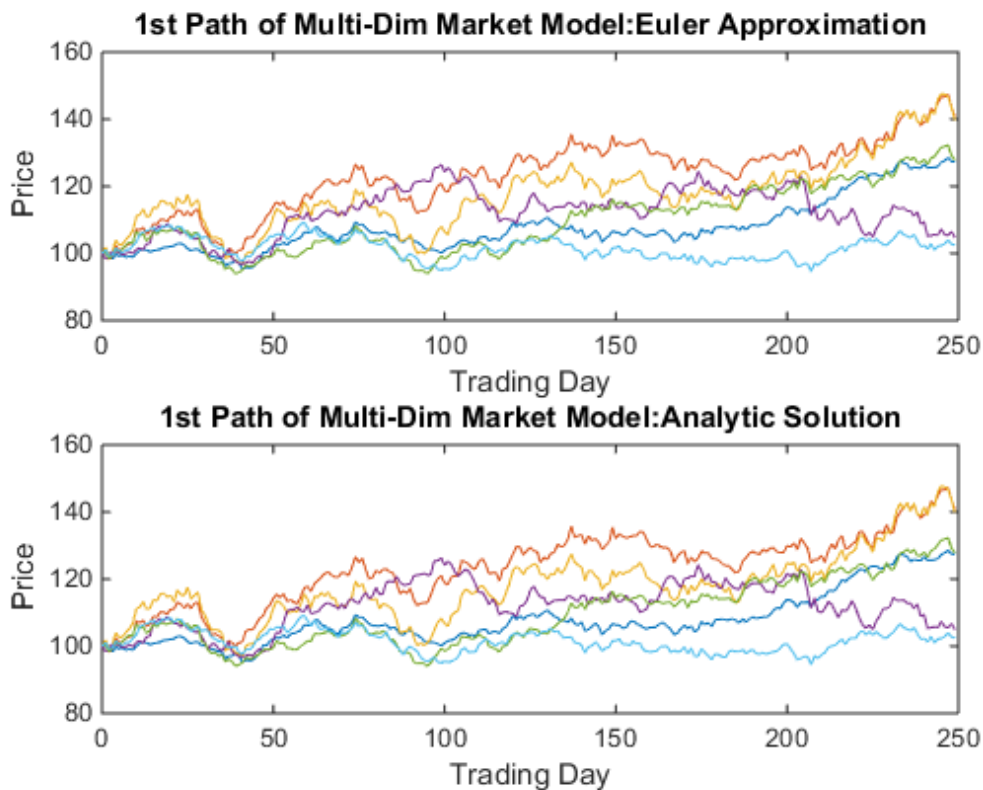


- 4 Simulate 10 trials of the solution and plot the first trial:

```

rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);
rng(142857,'twister')
[X,T] = simBySolution(GBM, nPeriods,...
    'DeltaTime', dt, 'nTrials', 10);
subplot(2,1,1)
plot(T, S(:,:,1)), xlabel('Trading Day'),ylabel('Price')
title('1st Path of Multi-Dim Market Model:Euler Approximation')
subplot(2,1,2)
plot(T, X(:,:,1)), xlabel('Trading Day'),ylabel('Price')
title('1st Path of Multi-Dim Market Model:Analytic Solution')

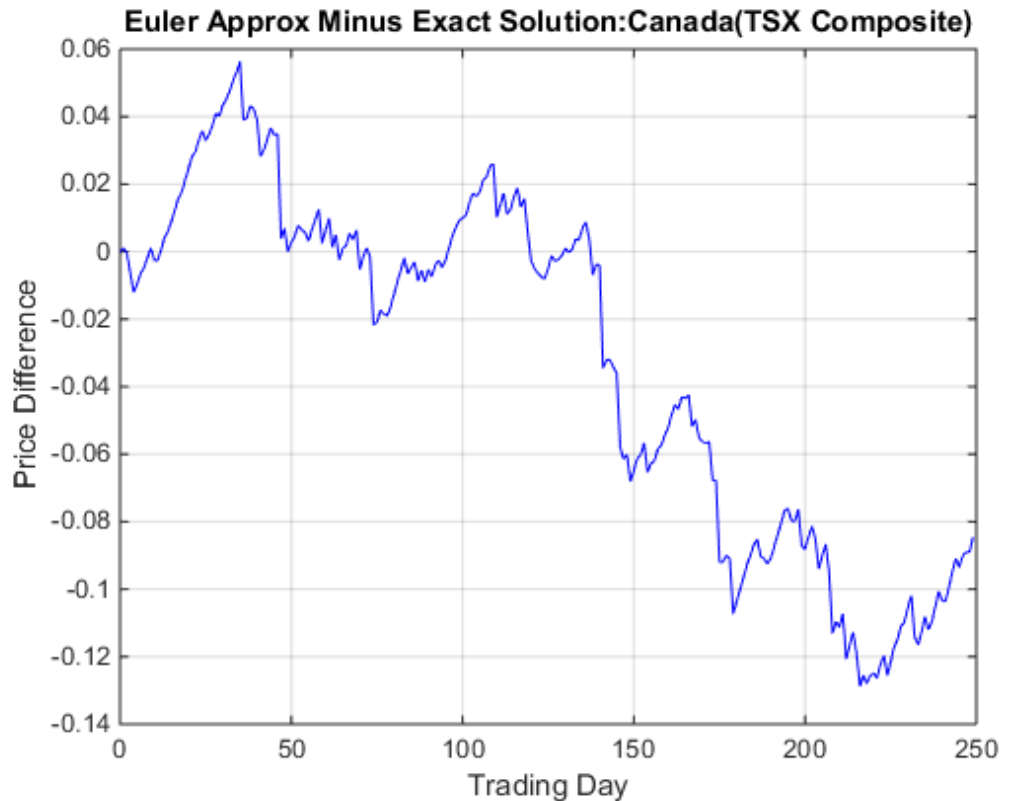
```



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.

- 5 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 Approx 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 piecewise constant parameters.

When all model parameters are piecewise 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: About Solution Precision and Error” on page 17-77.

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 *Simulating Equity Prices Using GBM Simulation Methods*). 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 data set:

```
load Data_GlobalIdx2
prices = [Dataset.TSX Dataset.CAC Dataset.DAX ...
         Dataset.NIK Dataset.FTSE Dataset.SP];
```

- 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.

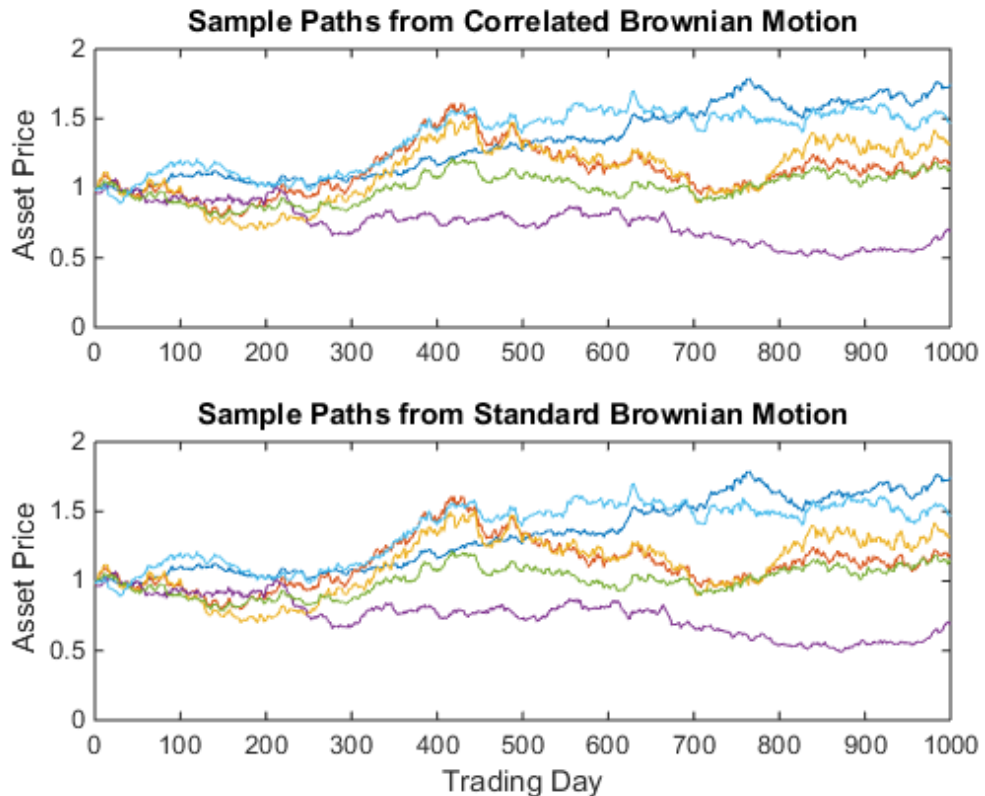
- 5 Simulate a single trial of 1000 observations (roughly four years of daily data) using both techniques. By default, all state variables start at 1:

```
rng(22814,'twister')
[X1,T] = simByEuler(GBM1,1000); % correlated Brownian motion
rng(22814,'twister')
[X2,T] = simByEuler(GBM2,1000); % standard Brownian motion
```

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

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

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



Dynamic Behavior of Market Parameters

As discussed in “Creating SDE Objects” on page 17-7, 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 the data.** Load a daily historical data set containing 3-month Euribor rates and closing index levels of France's CAC 40 spanning the time interval February 7, 2001 to April 24, 2006:

```
load Data_GlobalIdx2
```

- 2 Simulate risk-neutral sample paths.** 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(Dataset.CAC);
sigma   = std(returns)*sqrt(250);
yields  = Dataset.EB3M;
yields  = 360*log(1 + yields);
```

- 3 Compare the sample paths from two risk-neutral historical simulation approaches.** 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
rng(5713, 'twister')
obj       = gbm(mean(yields),diag(sigma),'StartState',100)
[X1,T] = simulate(obj,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, piecewise 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) piecewise constant interpolation. The notion of piecewise constant parameters is pervasive throughout the SDE architecture, and is discussed in more detail in “Optimizing Accuracy: About Solution Precision and Error” on page 17-77.

- 4 Perform a second simulation using the same initial random number state.** Complete the comparison by performing the second simulation using the same initial random number state:

```
rng(5713, 'twister')
obj = gbm(r, diag(sigma), 'StartState', 100)
X2 = simulate(obj, 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 Compare the two simulation trials.** Plot the series of risk-free reference rates to compare the two simulation trials:

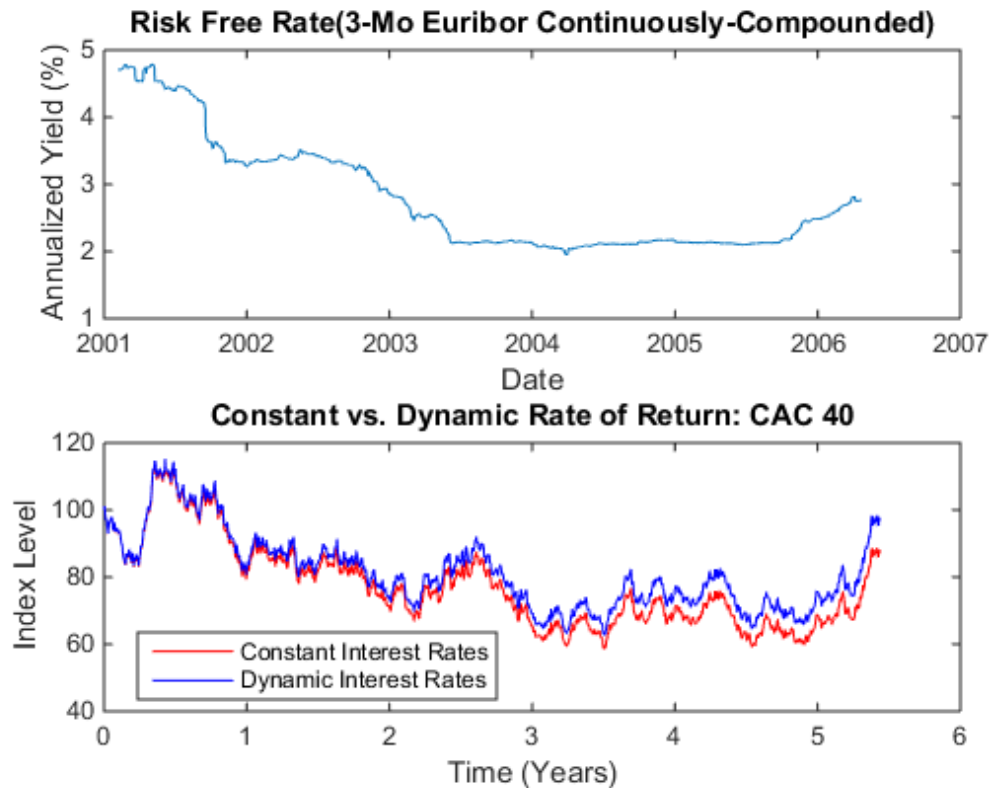
```
subplot(2,1,1)
plot(dates, 100*yields)
datetick('x')
xlabel('Date')
```



```

ylabel('Annualized Yield (%)')
title('Risk Free Rate(3-Mo 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.

Pricing Equity Options

As discussed in “Ensuring Positive Interest Rates” on page 17-66, 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 related example illustrates a simple, common end-of-period processing function to ensure nonnegative 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 `Example_BlackScholes.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 = Example_BlackScholes(nPeriods,nTrials)
```

```
f =
```

```
    BlackScholes: @Example_BlackScholes/saveTerminalStockPrice
    CallPrice: @Example_BlackScholes/getCallPrice
    PutPrice: @Example_BlackScholes/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:

```
rng(88161, 'twister')
X = simBySolution(obj,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 =
```

```
    13.9342
```

```
put =  
    6.4166
```

- 4** Price the options indirectly by invoking the nested functions:

```
f.CallPrice(strike,rate)  
f.PutPrice(strike,rate)
```

```
ans =  
    13.9342
```

```
ans =  
    6.4166
```

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.

Simulating Interest Rates

In this section...

“Simulating Interest Rates” on page 17-59

“Ensuring Positive Interest Rates” on page 17-66

Simulating Interest Rates

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).

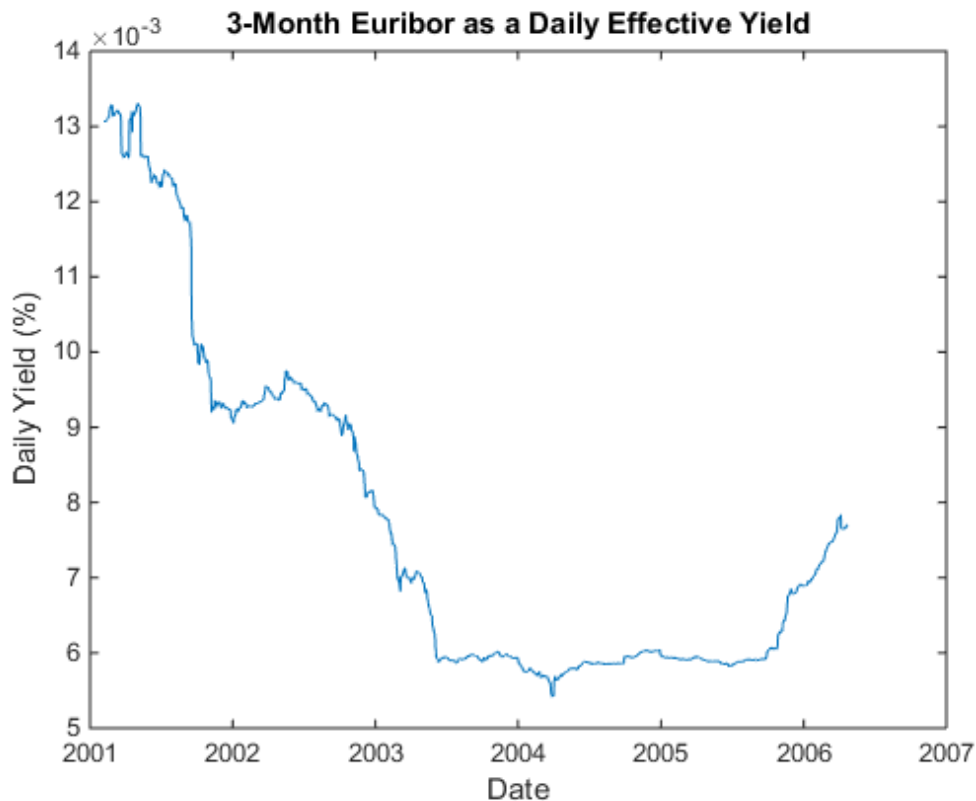
In contrast, interpolation methods allow you to traverse the time grid in any order, allowing both forward and backward movements in time. They allow you to specify a vector of interpolation times whose elements do not have to be unique.

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 the data.** Load a historical data set of three-month Euribor rates, observed daily, and corresponding trading dates spanning the time interval from February 7, 2001 through April 24, 2006:

```
load Data_GlobalIdx2
plot(dates, 100 * Dataset.EB3M)
datetick('x'), xlabel('Date'), ylabel('Daily Yield (%)')
```

```
title('3-Month Euribor as a Daily Effective Yield')
```



- 2 Fit a model to the data.** 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:

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

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:

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

```

yields      = Dataset.EB3M;
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 object and set its initial StartState.** Create an HWV object with an initial StartState set to the most recently observed short rate:

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

obj =

```

Class HWV: Hull-White/Vasicek
-----
    Dimensions: State = 1, Brownian = 1
-----
    StartTime: 0
    StartState: 7.70408e-05
    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-07
Level: 6.00424e-05
Speed: 0.00228854
```

- 4 Simulate the fitted model.** 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.

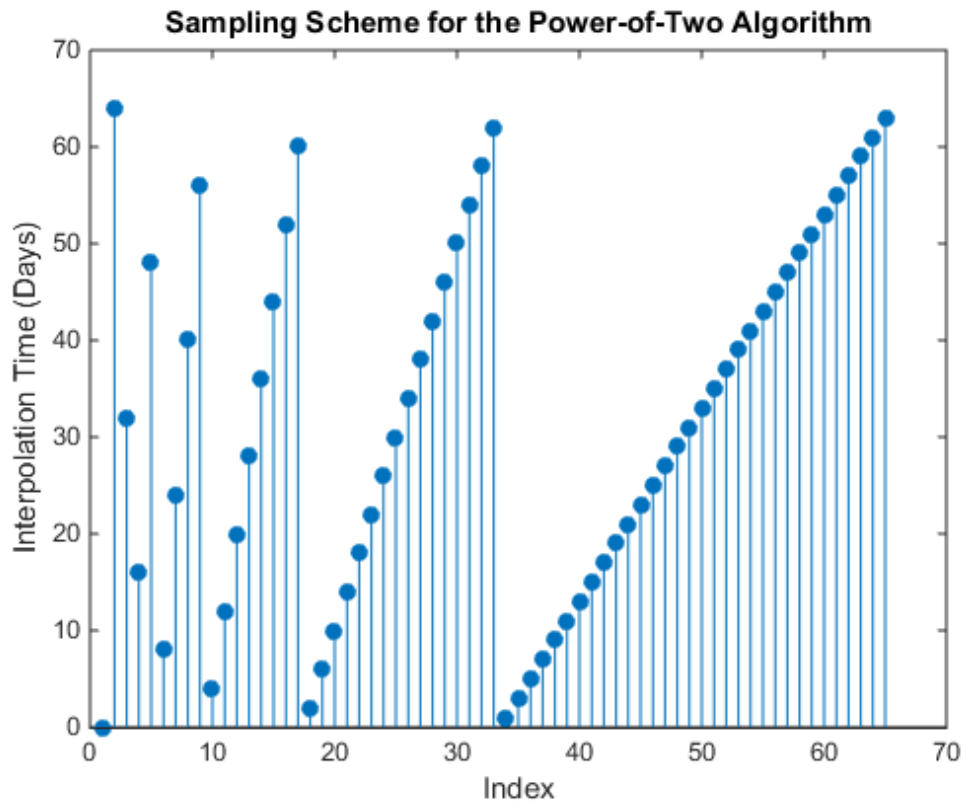
Create a vector of interpolation times to traverse the time grid by moving both forward and backward in time. Specifically, the first interpolation time is set to the most recent short rate observation time, the second interpolation time is set to the terminal time, and subsequent interpolation times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

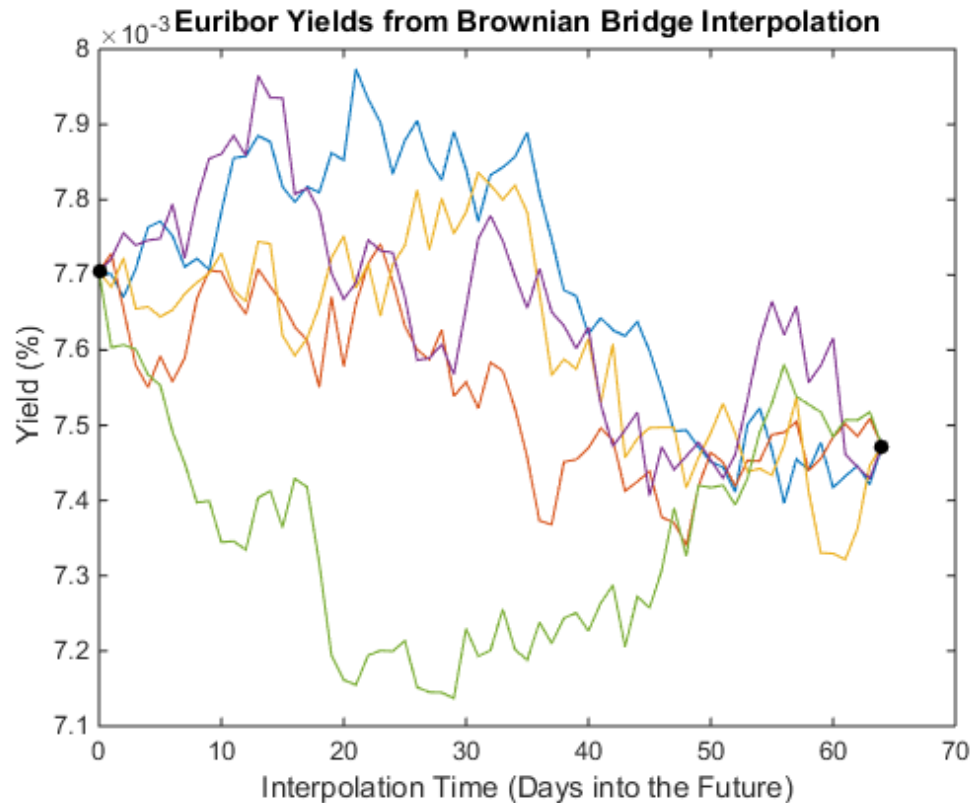
```
average = obj.StartState * exp(-speed * T) + level * ...  
(1 - exp(-speed * T));  
X       = [obj.StartState ; average];
```

- 7 Generate five sample paths.** 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 three-dimensional time series represents an independent trial), the interpolated output series `Y` also has five pages:

```
nTrials = 5;  
rng(63349, 'twister')  
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 from 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 “Stratified Sampling” on page 17-70.

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

Tip Brownian-bridge methods also apply more general variance-reduction techniques. For more information, see “Stratified Sampling” on page 17-70.

Ensuring Positive Interest Rates

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: About Solution Precision and Error” on page 17-77. 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 nonnegative 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 Simulate daily short rates of the CIR model.** To illustrate the latter statement, simulate daily short rates of the CIR model over one calendar year (approximately 250 trading days):

```
rng(14151617, 'twister')
obj = cir(0.25,@(t,X)0.1,0.2, 'StartState',0.02);
[X,T] = simByEuler(obj,250, 'DeltaTime',1/250, 'nTrials',5);
sprintf( '%0.4f\t%0.4f+i%0.4f\n', [T(195:205)';...
    real(X(195:205,1,4))'; imag(X(195:205,1,4))'] )
```

ans =

```
0.7760 0.0003+i0.0000
0.7800 0.0004+i0.0000
0.7840 0.0002+i0.0000
0.7880 -0.0000+i0.0000
0.7920 0.0001+i0.0000
0.7960 0.0002+i0.0000
0.8000 0.0002+i0.0000
0.8040 0.0008+i0.0001
0.8080 0.0004+i0.0001
0.8120 0.0008+i0.0001
0.8160 0.0008+i0.0001
```

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.

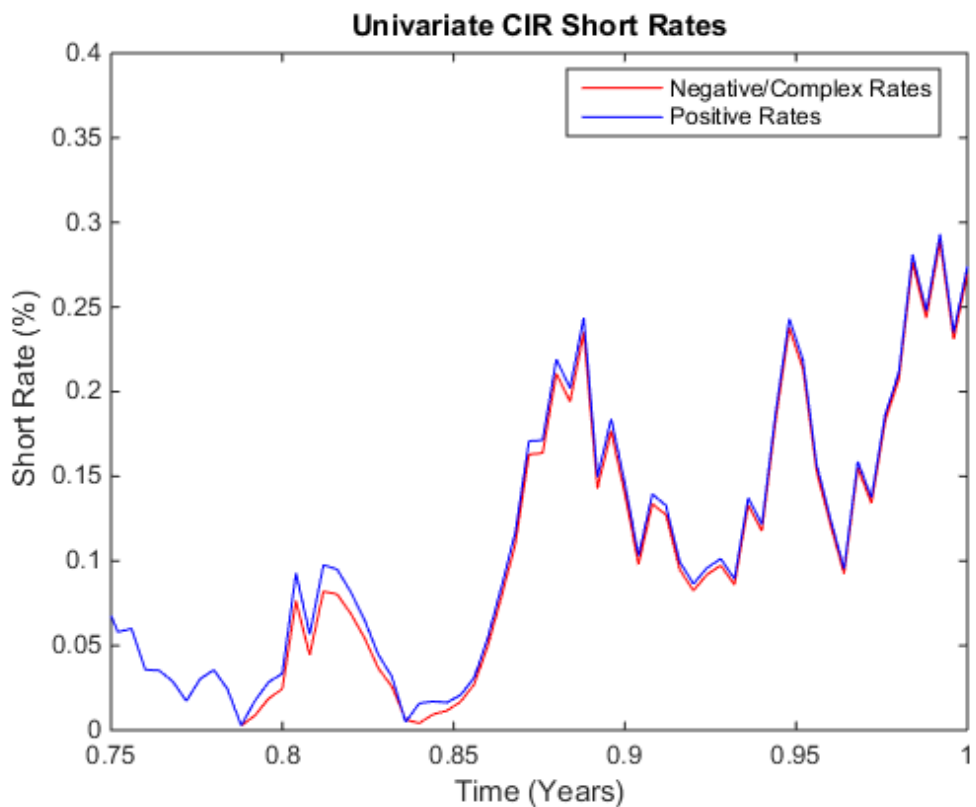
- 2 Repeat the simulation with a processing function.** Repeat the simulation, this time specifying a processing function that takes the absolute magnitude of the short rate at the end of each period. You can access the processing function by time and state (t, X_t) , but it only uses the state vector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

```
rng(14151617, 'twister')
```

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

- 3 Compare the adjusted and non-adjusted paths.** 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(:,1,4)), 'red',T,100*Y(:,1,4), 'blue')
axis([0.75 1 0 0.4])
xlabel('Time (Years)'), ylabel('Short Rate (%)')
title('Univariate CIR Short Rates')
legend({'Negative/Complex Rates' 'Positive Rates'}, ...
    'Location', 'Best')
```



Tip You can use this method to obtain more accurate SDE solutions. For more information, see “Performance Considerations” on page 17-75.

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 “Simulating Interest Rates” on page 17-59. 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 example `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:

```
rng(10203, 'twister')
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 = Example_StratifiedRNG(nPaths, sampleTimes);
```

- 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 three-dimensional output to a 2-dimensional equivalent array:

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

- 4 Verify the stratification:

- a Recreate the uniform draws with proportional sampling:

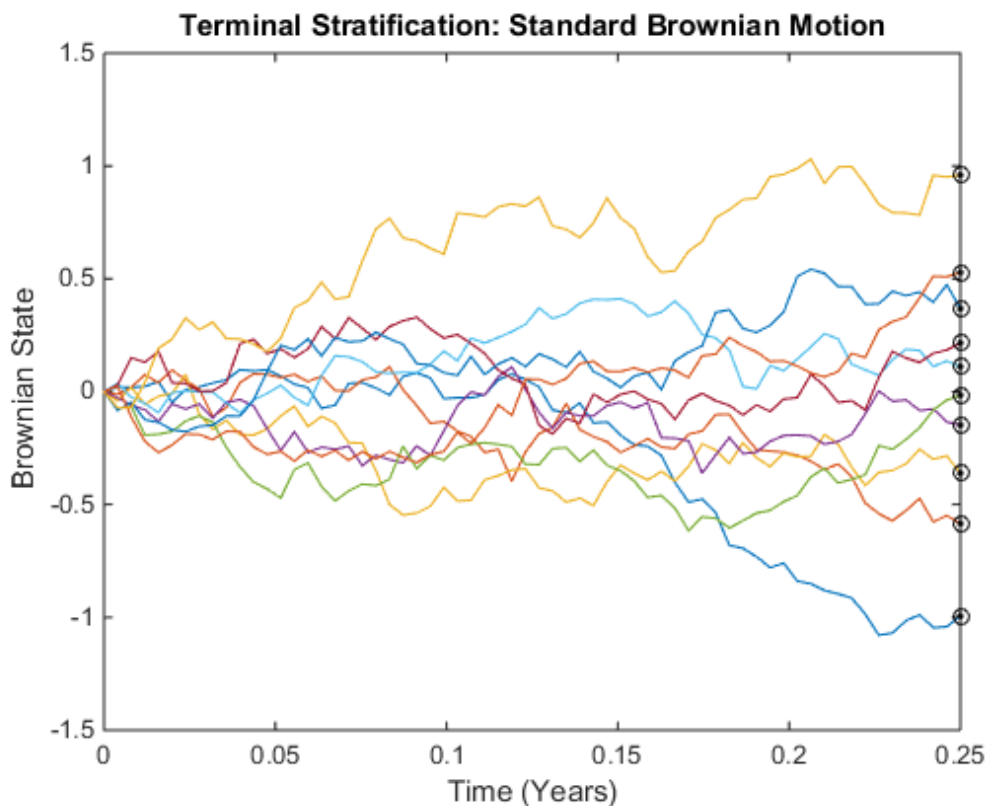
```
rng(10203, 'twister')
U = ((1:nPaths)' - 1 + rand(nPaths,1))/nPaths;
```

- 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 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:

```
load Data_GlobalIdx2
returns = price2ret(Dataset.FTSE);
sigma   = std(returns) * sqrt(252);
rate    = Dataset.EB3M;
rate    = mean(360 * log(1 + rate));
```

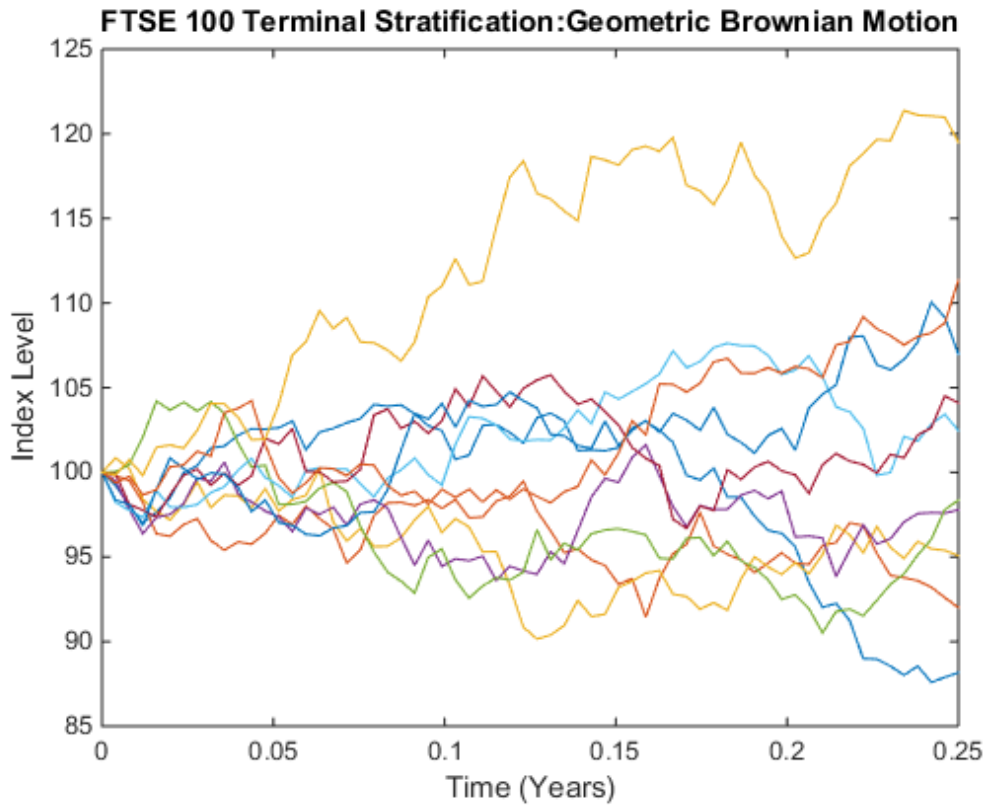
- 2 Create the GBM model, assuming the FTSE 100 starts at 100:

```
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           = Example_StratifiedRNG(nPaths, sampleTimes);
rng(10203, 'twister')
[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.

Tip For another example of variance reduction techniques, see “Simulating Interest Rates” on page 17-59.

Performance Considerations

In this section...

“Managing Memory” on page 17-75

“Enhancing Performance” on page 17-76

“Optimizing Accuracy: About Solution Precision and Error” on page 17-77

Managing Memory

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

- “Managing Memory with Outputs” on page 17-75
- “Managing Memory Using End-of-Period Processing Functions” on page 17-76

Managing Memory with Outputs

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: About Solution Precision and Error” on page 17-77).

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, three-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, then you can avoid storing `Paths` two ways:

- It is best practice to use the `~` output argument placeholder. For example, use the following output argument list to store `Z` and `Times`, but not `Paths`:

```
[~,Times,Z]
```

- Use 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.

Managing Memory Using End-of-Period Processing Functions

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 three-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. See “Pricing Equity Options” on page 17-56.

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.
- **Use models that have overloaded Euler simulation methods, 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 specifying 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.** Treat 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 applicable only to constant-parameter univariate models without specifying 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 (that is, an option that becomes worthless as soon as the price of the underlying asset crosses some prescribed barrier). Your end-of-period function could detect a barrier crossing and return a NaN to signal early termination of the current trial.

Optimizing Accuracy: About Solution 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 piecewise 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 piecewise constant approach provides exact solutions:

- “Creating Brownian Motion (BM) Models” on page 17-24 with constant parameters, simulated by Euler approximation (`simByEuler`).
- “Creating Geometric Brownian Motion (GBM) Models” on page 17-26 with constant parameters, simulated by closed-form solution (`simBySolution`).
- “Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 17-29 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 piecewise 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 three-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 Solution Accuracy

Consider a univariate geometric Brownian motion (GBM) model with constant parameters:

$$dX_t = 0.1X_t dt + 0.4X_t dW_t.$$

Assume that the expected rate of return and volatility parameters are annualized, and that a calendar year comprises 250 trading days.

- 1 Simulate approximately 4 years of univariate prices for both the exact solution and the Euler approximation for various values of `NSTEPS`:

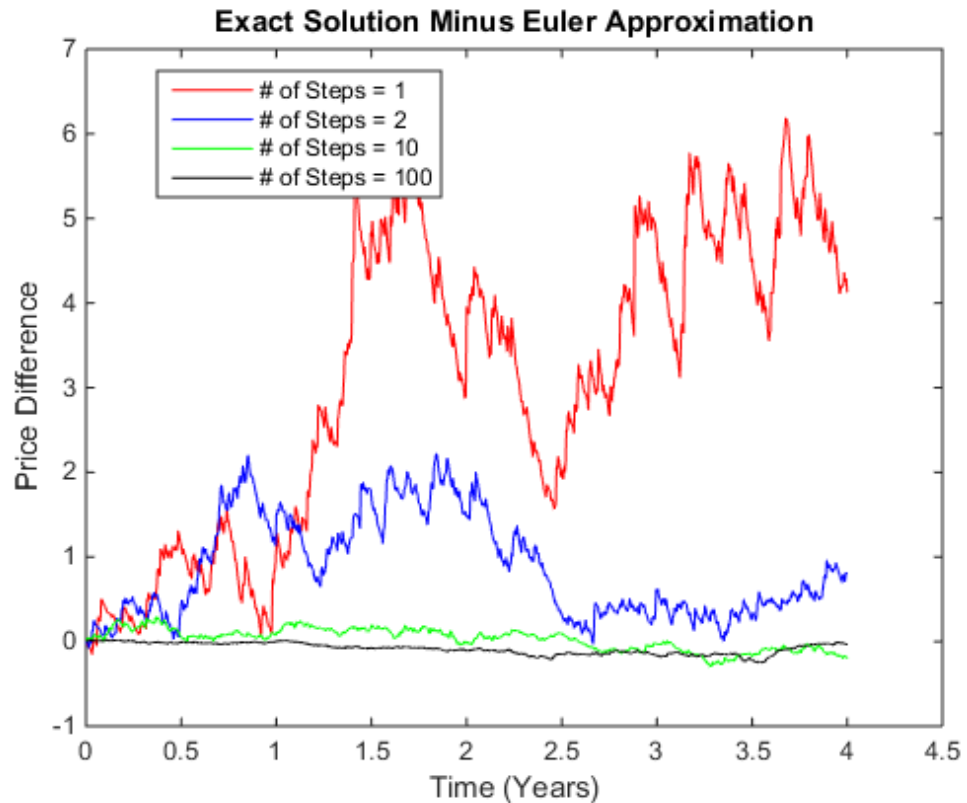
```
nPeriods = 1000;
dt       = 1/250;
obj      = gbm(0.1,0.4, 'StartState', 100);
rng(575, 'twister')
[X1,T1]  = simBySolution(obj,nPeriods,'DeltaTime',dt);
```

```
rng(575, 'twister')
[Y1,T1] = simByEuler(obj,nPeriods,'DeltaTime',dt);
rng(575, 'twister')
[X2,T2] = simBySolution(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',2);
rng(575, 'twister')
[Y2,T2] = simByEuler(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',2);
rng(575, 'twister')
[X3,T3] = simBySolution(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',10);
rng(575, 'twister')
[Y3,T3] = simByEuler(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',10);
rng(575, 'twister')
[X4,T4] = simBySolution(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',100);
rng(575, 'twister')
[Y4,T4] = simByEuler(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',100);
```

- 2 Compare the error (the difference between the exact solution and the Euler approximation) graphically:

```
clf;
plot(T1,X1 - Y1,'red')
hold on;
plot(T2,X2 - Y2,'blue')
plot(T3,X3 - Y3,'green')
plot(T4,X4 - Y4,'black')
hold off
xlabel('Time (Years)')
ylabel('Price Difference')
title('Exact Solution Minus Euler Approximation')
legend({'# of Steps = 1' '# of Steps = 2' ...
    '# of Steps = 10' '# of Steps = 100'},...
    'Location','Best')
hold off

whos T X Y
```



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.

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.

Pricing American Basket Options by Monte Carlo Simulation

This example shows how to model the fat-tailed behavior of asset returns and assess the impact of alternative joint distributions on basket option prices. Using various implementations of a separable multivariate Geometric Brownian Motion (GBM) process, often referred to as a *multi-dimensional market model*, the example simulates risk-neutral sample paths of an equity index portfolio and prices basket put options using the technique of Longstaff & Schwartz.

In addition, this example also illustrates salient features of the Stochastic Differential Equation (SDE) architecture, including

- Customized random number generation functions that compare Brownian motion and Brownian copulas
- End-of-period processing functions that form an equity index basket and price American options on the underlying basket based on the least squares method of Longstaff & Schwartz
- Piecewise probability distributions and Extreme Value Theory (EVT)

This example also highlights important issues of volatility and interest rate scaling. It illustrates how equivalent results can be achieved by working with daily or annualized data. For more information about EVT and copulas, see *Using Extreme Value Theory and Copulas to Evaluate Market Risk*.

Overview of the Modeling Framework

The ultimate objective of this example is to compare basket option prices derived from different noise processes. The first noise process is a traditional Brownian motion model whose index portfolio price process is driven by correlated Gaussian random draws. As alternatives, the Brownian motion benchmark is compared to noise processes driven by Gaussian and Student's t copulas, referred to collectively as a *Brownian copula*.

A copula is a multivariate cumulative distribution function (CDF) with uniformly-distributed margins. Although the theoretical foundations were established decades ago, copulas have experienced a tremendous surge in popularity over the last few years, primarily as a technique for modeling non-Gaussian portfolio risks.

Although numerous families exist, all copulas represent a statistical device for modeling the dependence structure between 2 or more random variables. In addition, important

statistics, such as *rank correlation* and *tail dependence*, are properties of a given copula and are unchanged by monotonic transforms of its margins.

These copula draws produce dependent random variables, which are subsequently transformed to individual variables (margins). This transformation is achieved with a semi-parametric probability distribution with generalized Pareto tails.

The risk-neutral market model to simulate is

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

where the risk-free rate, r , is assumed constant over the life of the option. Because this is a separable multivariate model, the risk-free return is a diagonal matrix in which the same riskless return is applied to all indices. Dividend yields are ignored to simplify the model its associated data collection.

In contrast, the specification of the exposure matrix, σ , depends on how the driving source of uncertainty is modeled. You can model it directly as a Brownian motion (correlated Gaussian random numbers implicitly mapped to Gaussian margins) or model it as a Brownian copula (correlated Gaussian or t random numbers explicitly mapped to semi-parametric margins).

Because the CDF and inverse CDF (quantile function) of univariate distributions are both monotonic transforms, a copula provides a convenient way to simulate dependent random variables whose margins are dissimilar and arbitrarily distributed. Moreover, because a copula defines a given dependence structure regardless of its margins, copula parameter calibration is typically easier than estimation of the joint distribution function.

Once you have simulated sample paths, options are priced by the least squares regression method of Longstaff & Schwartz (see *Valuing American Options by Simulation: A Simple Least-Squares Approach*, The Review of Financial Studies, Spring 2001). This approach uses least squares to estimate the expected payoff of an option if it is not immediately exercised. It does so by regressing the discounted option cash flows received in the future on the current price of the underlier associated with all in-the-money sample paths. The continuation function is estimated by a simple third-order polynomial, in which all cash flows and prices in the regression are normalized by the option strike price, improving numerical stability.

Import the Supporting Historical Dataset

Load a daily historical dataset of 3-month Euribor, the trading dates spanning the interval 07-Feb-2001 to 24-Apr-2006, and the closing index levels of the following representative large-cap equity indices:

- TSX Composite (Canada)
- CAC 40 (France)
- DAX (Germany)
- Nikkei 225 (Japan)
- FTSE 100 (UK)
- S&P 500 (US)

```
clear
load Data_GlobalIdx2
```

The following plots illustrate this data. Specifically, the plots show the relative price movements of each index and the Euribor risk-free rate proxy. The initial level of each index has been normalized to unity to facilitate the comparison of relative performance over the historical record.

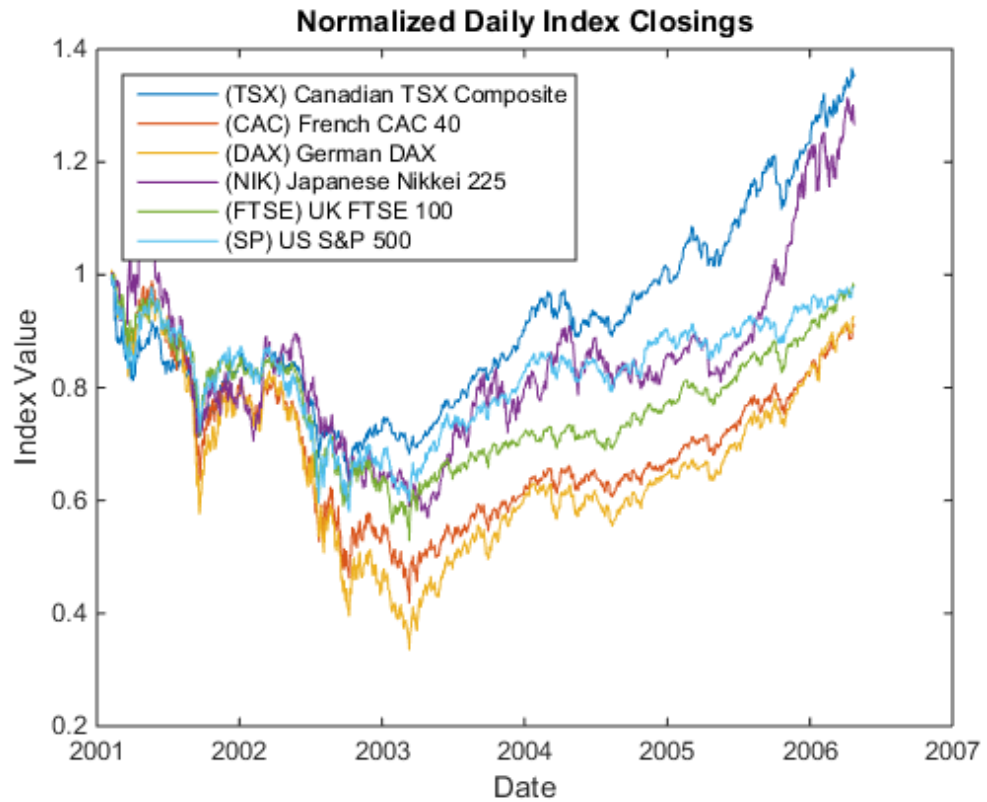
```
nIndices = size(Data,2)-1;    % # of indices

prices = Data(:,1:end-1);

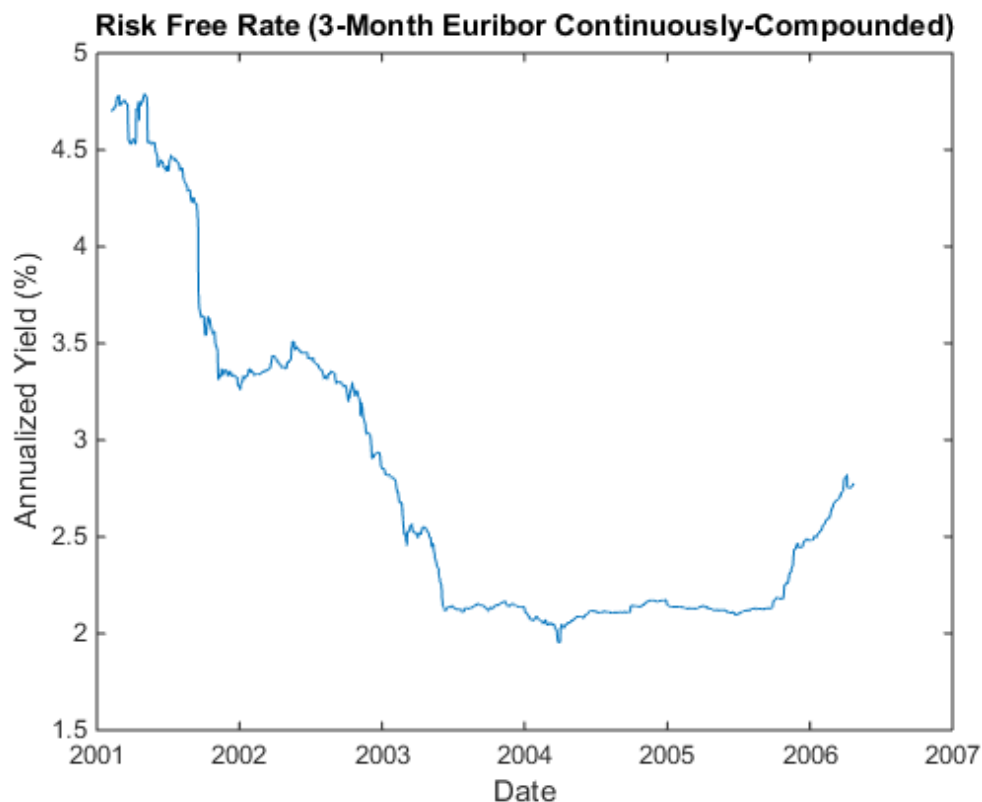
yields = Data(:,end);        % daily effective yields
yields = 360 * log(1 + yields); % continuously-compounded, annualized yield

plot(dates, ret2tick(tick2ret(prices,[], 'continuous'),[],[],[], 'continuous'))
datetick('x')

xlabel('Date')
ylabel('Index Value')
title('Normalized Daily Index Closings')
legend(series{1:end-1}, 'Location', 'NorthWest')
```



```
plot(dates, 100 * yields)
datetick('x')
xlabel('Date')
ylabel('Annualized Yield (%)')
title('Risk Free Rate (3-Month Euribor Continuously-Compounded)')
```



Extreme Value Theory & Piecewise Probability Distributions

To prepare for copula modeling, characterize individually the distribution of returns of each index. Although the distribution of each return series may be characterized parametrically, it is useful to fit a semi-parametric model using a piecewise distribution with generalized Pareto tails. This uses Extreme Value Theory to better characterize the behavior in each tail.

The Statistics Toolbox™ software currently supports two univariate probability distributions related to EVT, a statistical tool for modeling the fat-tailed behavior of financial data such as asset returns and insurance losses:

- Generalized Extreme Value (GEV) distribution, which uses a modeling technique known as the *block maxima or minima* method. This approach, divides a historical

dataset into a set of sub-intervals, or blocks, and the largest or smallest observation in each block is recorded and fitted to a GEV distribution.

- Generalized Pareto (GP) distribution, uses a modeling technique known as the *distribution of exceedances* or *peaks over threshold* method. This approach sorts a historical dataset, and fits the amount by which those observations that exceed a specified threshold to a GP distribution.

The following analysis highlights the Pareto distribution, which is more widely used in risk management applications.

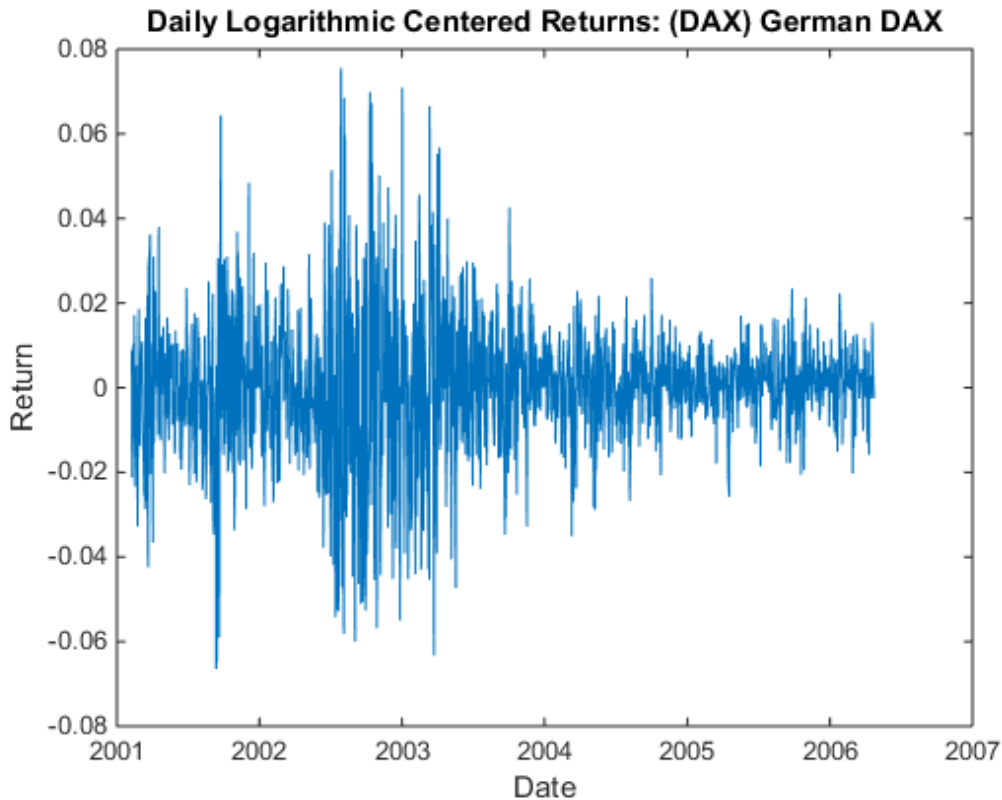
Suppose we wish to create a complete statistical description of the probability distribution of daily asset returns of any one of the equity indices. Assume that this description is provided by a piecewise semi-parametric distribution, where the asymptotic behavior in each tail is characterized by a generalized Pareto distribution.

Ultimately, a copula will be used to generate random numbers to drive the simulations. The CDF and inverse CDF transforms will capture the volatility of simulated returns as part of the diffusion term of the SDE. The mean return of each index is governed by the riskless rate and incorporated in the drift term of the SDE. The following code segment centers the returns (that is, extracts the mean) of each index.

Because the following analysis uses extreme value theory to characterize the distribution of each equity index return series, it is helpful to examine details for a particular country:

```
returns = tick2ret(prices,[],'continuous');           % convert prices to returns
returns = bsxfun(@minus, returns, mean(returns));    % center the returns
index   = 3;                                       % Germany stored in column 3

plot(dates(2:end), returns(:,index))
datetick('x')
xlabel('Date')
ylabel('Return')
title(['Daily Logarithmic Centered Returns: ' series{index}])
```



Note that this code segment can be changed to examine details for any country.

Using these centered returns, estimate the empirical, or non-parametric, CDF of each index with a Gaussian kernel. This smoothes the CDF estimates, eliminating the staircase pattern of unsmoothed sample CDFs. Although non-parametric kernel CDF estimates are well-suited for the interior of the distribution, where most of the data is found, they tend to perform poorly when applied to the upper and lower tails. To better estimate the tails of the distribution, apply EVT to the returns that fall in each tail.

Specifically, find upper and lower thresholds such that 10% of the returns is reserved for each tail. Then fit the amount by which the extreme returns in each tail fall beyond the associated threshold to a Pareto distribution by maximum likelihood.

The following code segment creates one object of type `paretotails` for each index return series. These Pareto tail objects encapsulate the estimates of the parametric Pareto lower tail, the non-parametric kernel-smoothed interior, and the parametric Pareto upper tail to construct a composite semi-parametric CDF for each index.

```
tailFraction = 0.1; % decimal fraction allocated to each tail
tails = cell(nIndices,1); % cell array of Pareto tail objects

for i = 1:nIndices
    tails{i} = paretotails(returns(:,i), tailFraction, 1 - tailFraction, 'kernel');
end
```

The resulting piecewise distribution object allows interpolation within the interior of the CDF and extrapolation (function evaluation) in each tail. Extrapolation allows estimation of quantiles outside the historical record, which is invaluable for risk management applications.

Pareto tail objects also provide methods to evaluate the CDF and inverse CDF (quantile function), and to query the cumulative probabilities and quantiles of the boundaries between each segment of the piecewise distribution.

Now that three distinct regions of the piecewise distribution have been estimated, graphically concatenate and display the result.

The following code calls the CDF and inverse CDF methods of the Pareto tails object of interest with data other than that upon which the fit is based. The referenced methods have access to the fitted state. They are now invoked to select and analyze specific regions of the probability curve, acting as a powerful data filtering mechanism.

For reference, the plot also includes a zero-mean Gaussian CDF of the same standard deviation. To a degree, the variation in options prices reflect the extent to which the distribution of each asset differs from this normal curve.

```
minProbability = cdf(tails{index}, (min(returns(:,index))));
maxProbability = cdf(tails{index}, (max(returns(:,index))));

pLowerTail = linspace(minProbability , tailFraction , 200); % lower tail
pUpperTail = linspace(1 - tailFraction, maxProbability , 200); % upper tail
pInterior = linspace(tailFraction , 1 - tailFraction, 200); % interior

plot(icdf(tails{index}, pLowerTail), pLowerTail, 'red' , 'LineWidth', 2)
hold on
grid on
plot(icdf(tails{index}, pInterior) , pInterior , 'black', 'LineWidth', 2)
```

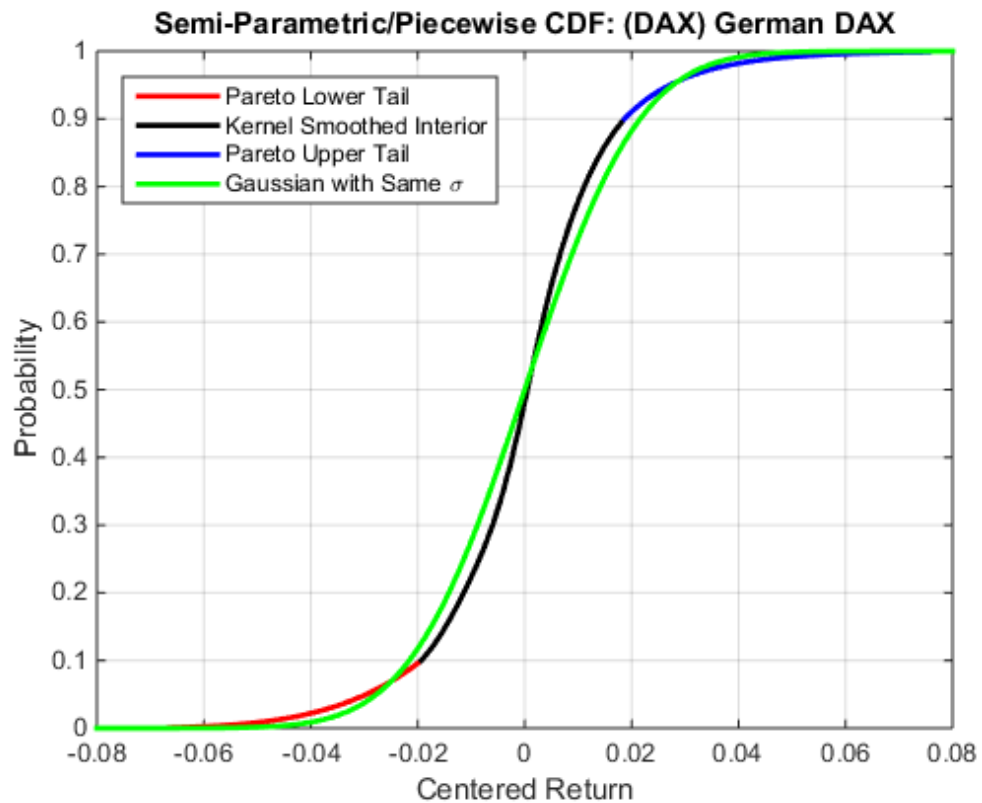
```

plot(icdf(tails{index}, pUpperTail), pUpperTail, 'blue' , 'LineWidth', 2)

limits = axis;
x = linspace(limits(1), limits(2));
plot(x, normcdf(x, 0, std(returns(:,index))), 'green', 'LineWidth', 2)

fig = gcf;
fig.Color = [1 1 1];
hold off
xlabel('Centered Return')
ylabel('Probability')
title(['Semi-Parametric/Piecewise CDF: ' series{index}])
legend({'Pareto Lower Tail' 'Kernel Smoothed Interior' ...
       'Pareto Upper Tail' 'Gaussian with Same \sigma'}, 'Location', 'NorthWest')

```



The lower and upper tail regions, displayed in red and blue, respectively, are suitable for extrapolation, while the kernel-smoothed interior, in black, is suitable for interpolation.

Copula Calibration

The Statistics Toolbox software includes functionality that calibrates and simulates Gaussian and t copulas.

Using the daily index returns, estimate the parameters of the Gaussian and t copulas using the function `copulafit`. Since a t copula becomes a Gaussian copula as the scalar degrees of freedom parameter (DoF) becomes infinitely large, the two copulas are really of the same family, and therefore share a linear correlation matrix as a fundamental parameter.

Although calibration of the linear correlation matrix of a Gaussian copula is straightforward, the calibration of a t copula is not. For this reason, the Statistics Toolbox software offers two techniques to calibrate a t copula:

- The first technique performs maximum likelihood estimation (MLE) in a two-step process. The inner step maximizes the log-likelihood with respect to the linear correlation matrix, given a fixed value for the degrees of freedom. This conditional maximization is placed within a 1-D maximization with respect to the degrees of freedom, thus maximizing the log-likelihood over all parameters. The function being maximized in this outer step is known as the profile log-likelihood for the degrees of freedom.
- The second technique is derived by differentiating the log-likelihood function with respect to the linear correlation matrix, assuming the degrees of freedom is a fixed constant. The resulting expression is a non-linear equation that can be solved iteratively for the correlation matrix. This technique approximates the profile log-likelihood for the degrees of freedom parameter for large sample sizes. This technique is usually significantly faster than the true maximum likelihood technique outlined above; however, you should not use it with small or moderate sample sizes as the estimates and confidence limits may not be accurate.

When the uniform variates are transformed by the empirical CDF of each margin, the calibration method is often known as canonical maximum likelihood (CML). The following code segment first transforms the daily centered returns to uniform variates by the piecewise, semi-parametric CDFs derived above. It then fits the Gaussian and t copulas to the transformed data:

```
U = zeros(size(returns));
```

```

for i = 1:nIndices
    U(:,i) = cdf(tails{i}, returns(:,i));    % transform each margin to uniform
end

options      = statset('Display', 'off', 'TolX', 1e-4);
[rhoT, DoF] = copulafit('t', U, 'Method', 'ApproximateML', 'Options', options);
rhoG        = copulafit('Gaussian', U);

```

The estimated correlation matrices are quite similar but not identical.

```
corrcoef(returns) % linear correlation matrix of daily returns
```

```
ans =
```

1.0000	0.4813	0.5058	0.1854	0.4573	0.6526
0.4813	1.0000	0.8485	0.2261	0.8575	0.5102
0.5058	0.8485	1.0000	0.2001	0.7650	0.6136
0.1854	0.2261	0.2001	1.0000	0.2295	0.1439
0.4573	0.8575	0.7650	0.2295	1.0000	0.4617
0.6526	0.5102	0.6136	0.1439	0.4617	1.0000

```
rhoG          % linear correlation matrix of the optimized Gaussian copula
```

```
rhoG =
```

1.0000	0.4745	0.5018	0.1857	0.4721	0.6622
0.4745	1.0000	0.8606	0.2393	0.8459	0.4912
0.5018	0.8606	1.0000	0.2126	0.7608	0.5811
0.1857	0.2393	0.2126	1.0000	0.2396	0.1494
0.4721	0.8459	0.7608	0.2396	1.0000	0.4518
0.6622	0.4912	0.5811	0.1494	0.4518	1.0000

```
rhoT          % linear correlation matrix of the optimized t copula
```

```
rhoT =
```

1.0000	0.4671	0.4858	0.1907	0.4734	0.6521
0.4671	1.0000	0.8871	0.2567	0.8500	0.5122
0.4858	0.8871	1.0000	0.2326	0.7723	0.5877

0.1907	0.2567	0.2326	1.0000	0.2503	0.1539
0.4734	0.8500	0.7723	0.2503	1.0000	0.4769
0.6521	0.5122	0.5877	0.1539	0.4769	1.0000

Note the relatively low degrees of freedom parameter obtained from the t copula calibration, indicating a significant departure from a Gaussian situation.

```
DoF          % scalar degrees of freedom parameter of the optimized t copula
```

```
DoF =
```

```
4.8613
```

Copula Simulation

Now that the copula parameters have been estimated, simulate jointly-dependent uniform variates using the function `copularnd`.

Then, by extrapolating the Pareto tails and interpolating the smoothed interior, transform the uniform variates derived from `copularnd` to daily centered returns via the inverse CDF of each index. These simulated centered returns are consistent with those obtained from the historical dataset. The returns are assumed to be independent in time, but at any point in time possess the dependence and rank correlation induced by the given copula.

The following code segment illustrates the dependence structure by simulating centered returns using the t copula. It then plots a 2-D scatter plot with marginal histograms for the French CAC 40 and German DAX using the Statistics Toolbox `scatterhist` function. The French and German indices were chosen simply because they have the highest correlation of the available data.

```
nPoints = 10000;          % # of simulated observations

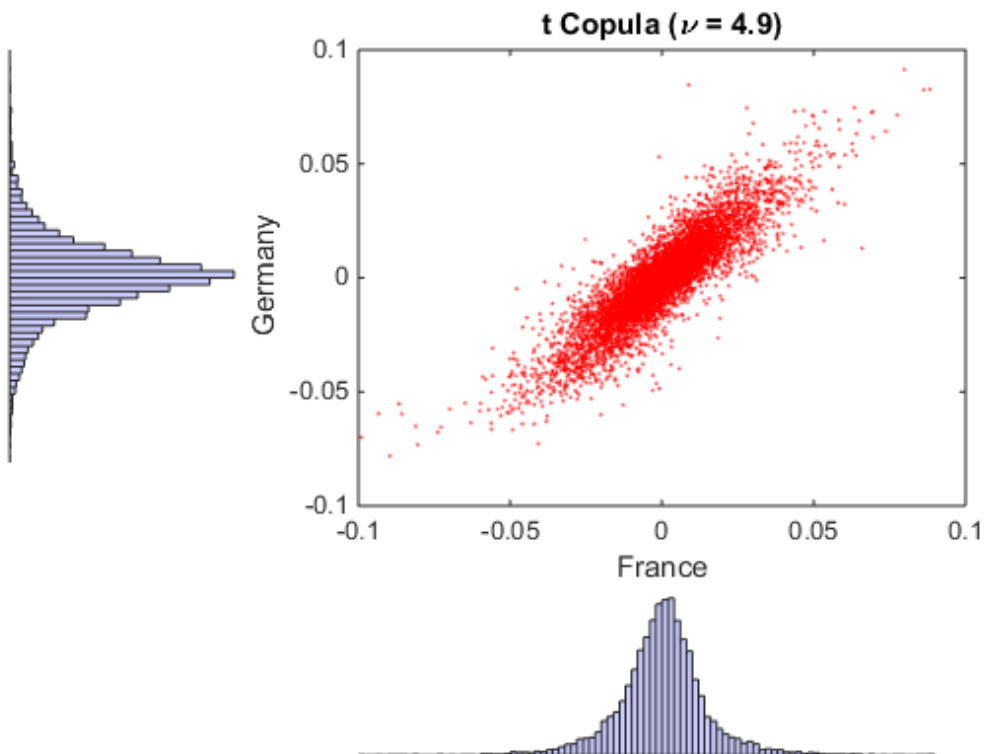
s = RandStream.getGlobalStream();
reset(s)

R = zeros(nPoints, nIndices);          % pre-allocate simulated returns array
U = copularnd('t', rhoT, DoF, nPoints); % simulate U(0,1) from t copula

for j = 1:nIndices
    R(:,j) = icdf(tails{j}, U(:,j));
```

```
end
```

```
h = scatterhist(R(:,2), R(:,3), 'Color', 'r', 'Marker', '.', 'MarkerSize', 1);
fig = gcf;
fig.Color = [1 1 1];
y1 = ylim(h(1));
y3 = ylim(h(3));
xlim(h(1), [-.1 .1])
ylim(h(1), [-.1 .1])
xlim(h(2), [-.1 .1])
ylim(h(3), [(y3(1) + (-0.1 - y1(1))) (y3(2) + (0.1 - y1(2)))]);
xlabel('France')
ylabel('Germany')
title(['t Copula (\nu = ' num2str(DoF,2) ')'])
```

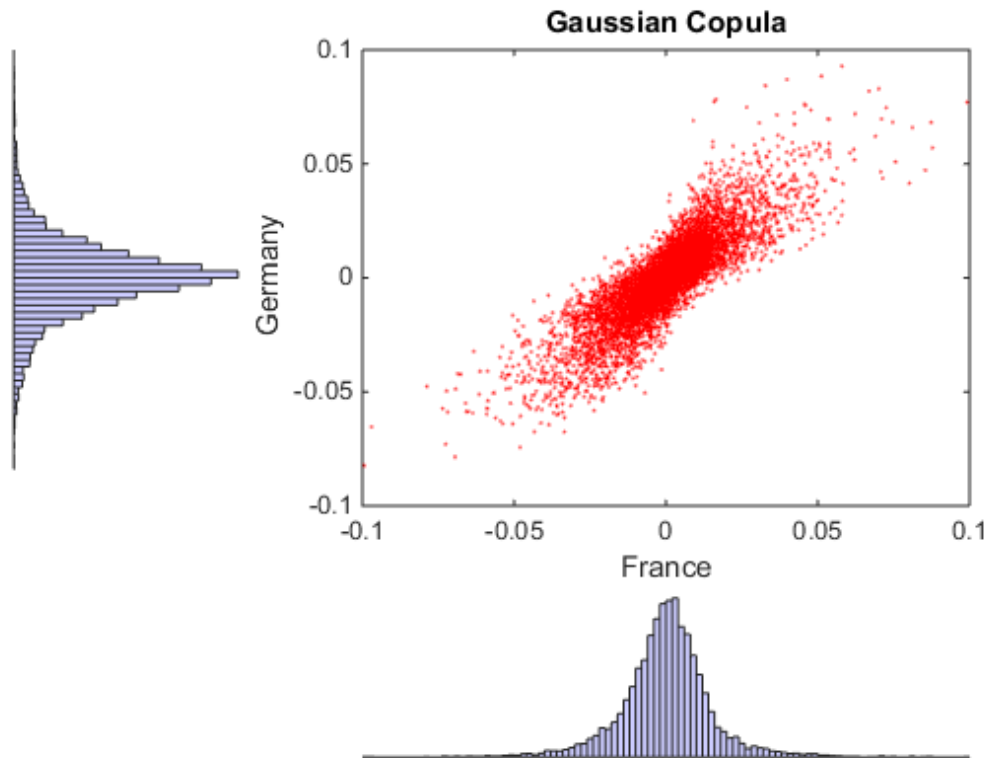


Now simulate and plot centered returns using the Gaussian copula.

```
reset(s)
R = zeros(nPoints, nIndices); % pre-allocate simulated returns array
U = copularnd('Gaussian', rhoG, nPoints); % simulate U(0,1) from Gaussian copula

for j = 1:nIndices
    R(:,j) = icdf(tails{j}, U(:,j));
end

h = scatterhist(R(:,2), R(:,3), 'Color', 'r', 'Marker', '.', 'MarkerSize', 1);
fig = gcf;
fig.Color = [1 1 1];
y1 = ylim(h(1));
y3 = ylim(h(3));
xlim(h(1), [-.1 .1])
ylim(h(1), [-.1 .1])
xlim(h(2), [-.1 .1])
ylim(h(3), [(y3(1) + (-0.1 - y1(1))) (y3(2) + (0.1 - y1(2))]))
xlabel('France')
ylabel('Germany')
title('Gaussian Copula')
```



Examine these two figures. There is a strong similarity between the miniature histograms on the corresponding axes of each figure. This similarity is not coincidental.

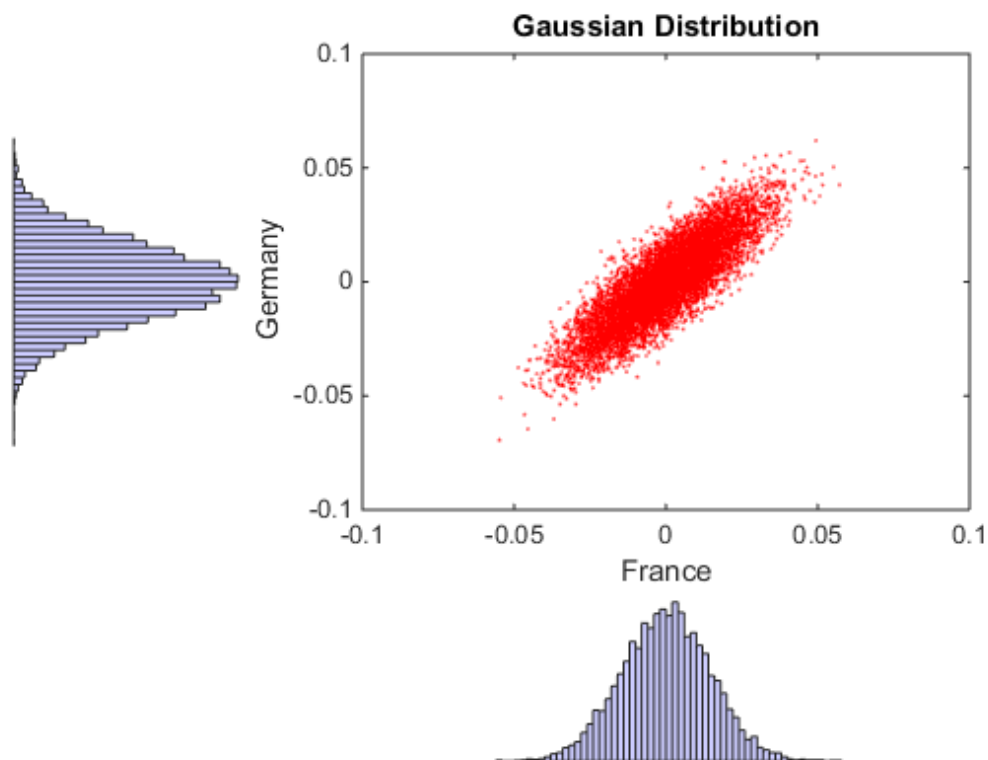
Both copulas simulate uniform random variables, which are then transformed to daily centered returns by the inverse CDF of the piecewise distribution of each index. Therefore, the simulated returns of any given index are identically distributed regardless of the copula.

However, the scatter graph of each figure indicates the dependence structure associated with the given copula, and in contrast to the univariate margins shown in the histograms, the scatter graphs are distinct.

Once again, the copula defines a dependence structure regardless of its margins, and therefore offers many features not limited to calibration alone.

For reference, simulate and plot centered returns using the Gaussian distribution, which underlies the traditional Brownian motion model.

```
reset(s)
R = mvnrnd(zeros(1,nIndices), cov(returns), nPoints);
h = scatterhist(R(:,2), R(:,3), 'Color', 'r', 'Marker', '.', 'MarkerSize', 1);
fig = gcf;
fig.Color = [1 1 1];
y1 = ylim(h(1));
y3 = ylim(h(3));
xlim(h(1), [-.1 .1])
ylim(h(1), [-.1 .1])
xlim(h(2), [-.1 .1])
ylim(h(3), [(y3(1) + (-0.1 - y1(1))) (y3(2) + (0.1 - y1(2))]))
xlabel('France')
ylabel('Germany')
title('Gaussian Distribution')
```



American Option Pricing Using the Longstaff & Schwartz Approach

Now that the copulas have been calibrated, compare the prices of at-the-money American basket options derived from various approaches. To simplify the analysis, assume that:

- All indices begin at 100.
- The portfolio holds a single unit, or share, of each index such that the value of the portfolio at any time is the sum of the values of the individual indices.
- The option expires in 3 months.
- The information derived from the daily data is annualized.
- Each calendar year is composed of 252 trading days.
- Index levels are simulated daily.

- The option may be exercised at the end of every trading day and approximates the American option as a Bermudan option.

Now compute the parameters common to all simulation methods:

```
dt          = 1 / 252;                % time increment = 1 day = 1/252 years
yields      = Data(:,end);           % daily effective yields
yields      = 360 * log(1 + yields); % continuously-compounded, annualized yields
r           = mean(yields);          % historical 3M Euribor average
X           = repmat(100, nIndices, 1); % initial state vector
strike      = sum(X);                % initialize an at-the-money basket

nTrials     = 100;                    % # of independent trials
nPeriods    = 63;                    % # of simulation periods: 63/252 = 0.25 years = 3 months
```

Now create two separable multi-dimensional market models in which the riskless return and volatility exposure matrices are both diagonal.

While both are diagonal GBM models with identical risk-neutral returns, the first is driven by a correlated Brownian motion and explicitly specifies the sample linear correlation matrix of centered returns. This correlated Brownian motion process is then weighted by a diagonal matrix of annualized index volatilities or standard deviations.

As an alternative, the same model could be driven by an uncorrelated Brownian motion (*standard Brownian motion*) by specifying `correlation` as an identity matrix, or by simply accepting the default value. In this case, the exposure matrix `sigma` is specified as the lower Cholesky factor of the index return covariance matrix. Because the copula-based approaches simulate dependent random numbers, the diagonal exposure form is chosen for consistency. For further details, see *Alternatives for Inducing Dependence & Correlation*.

```
sigma       = std(returns) * sqrt(252); % annualized volatility
correlation = corrcoef(returns);        % correlated Gaussian disturbances
GBM1        = gbm(diag(r(ones(1,nIndices))), diag(sigma), 'StartState', X, ...
                 'Correlation', correlation);
```

Now create the second model driven by the Brownian copula with an identity matrix `sigma`.

```
GBM2 = gbm(diag(r(ones(1,nIndices))), eye(nIndices), 'StartState', X);
```

The newly created model may seem unusual, but it highlights the flexibility of the SDE architecture.

When working with copulas, it is often convenient to allow the random number generator function $Z(t,X)$ to induce dependence (of which the traditional notion of linear correlation is a special case) with the copula, and to induce magnitude or scale of variation (similar to volatility or standard deviation) with the semi-parametric CDF and inverse CDF transforms. Since the CDF and inverse CDF transforms of each index inherit the characteristics of historical returns, this also explains why the returns are now centered.

In the following sections, statements like:

```
z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 'Gaussian');
```

or

```
z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 't');
```

fit the Gaussian and t copula dependence structures, respectively, and the semi-parametric margins to the centered returns scaled by the square root of the number of trading days per year (252). This scaling does not annualize the daily centered returns. Instead, it scales them such that the volatility remains consistent with the diagonal annualized exposure matrix σ of the traditional Brownian motion model (GBM1) created previously.

In this example, you also specify an end-of-period processing function that accepts time followed by state (t,X) , and records the sample times and value of the portfolio as the single-unit weighted average of all indices. This function also shares this information with other functions designed to price American options with a constant riskless rate using the least squares regression approach of Longstaff & Schwartz.

```
f = Example_LongstaffSchwartz(nPeriods, nTrials)
```

```
f =
```

```
    LongstaffSchwartz: @Example_LongstaffSchwartz/saveBasketPrices  
    CallPrice: @Example_LongstaffSchwartz/getCallPrice  
    PutPrice: @Example_LongstaffSchwartz/getPutPrice  
    Prices: @Example_LongstaffSchwartz/getBasketPrices
```

Now simulate independent trials of equity index prices over 3 calendar months using the default `simByEuler` method. No outputs are requested from the simulation methods; in fact the simulated prices of the individual indices which comprise the basket are unnecessary. Call option prices are reported for convenience:

```

reset(s)

simByEuler(GBM1, nPeriods, 'nTrials' , nTrials, 'DeltaTime', dt, ...
           'Processes', f.LongstaffSchwartz);

BrownianMotionCallPrice = f.CallPrice(strike, r);
BrownianMotionPutPrice  = f.PutPrice (strike, r);

```

```

reset(s)

z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 'Gaussian');
f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM2, nPeriods, 'nTrials' , nTrials, 'DeltaTime', dt, ...
           'Processes', f.LongstaffSchwartz, 'Z', z);

GaussianCopulaCallPrice = f.CallPrice(strike, r);
GaussianCopulaPutPrice  = f.PutPrice (strike, r);

```

Now repeat the copula simulation with the t copula dependence structure. You use the same model object for both copulas; only the random number generator and option pricing functions need to be re-initialized.

```

reset(s)

z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 't');
f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM2, nPeriods, 'nTrials' , nTrials, 'DeltaTime', dt, ...
           'Processes', f.LongstaffSchwartz, 'Z', z);

tCopulaCallPrice = f.CallPrice(strike, r);
tCopulaPutPrice  = f.PutPrice (strike, r);

```

Finally, compare the American put and call option prices obtained from all models.

```

disp(' ')
fprintf('          # of Monte Carlo Trials: %8d\n' , nTrials)
fprintf('          # of Time Periods/Trial: %8d\n\n' , nPeriods)
fprintf(' Brownian Motion American Call Basket Price: %8.4f\n\n' , BrownianMotionCallPr
fprintf(' Brownian Motion American Put  Basket Price: %8.4f\n\n' , BrownianMotionPutPri
fprintf(' Gaussian Copula American Call Basket Price: %8.4f\n\n' , GaussianCopulaCallPr
fprintf(' Gaussian Copula American Put  Basket Price: %8.4f\n\n' , GaussianCopulaPutPri
fprintf('          t Copula American Call Basket Price: %8.4f\n\n' , tCopulaCallPrice)
fprintf('          t Copula American Put  Basket Price: %8.4f\n\n' , tCopulaPutPrice)

```

```
# of Monte Carlo Trials:      100
# of Time Periods/Trial:      63

Brownian Motion American Call Basket Price:  25.9456
Brownian Motion American Put  Basket Price:  16.4132

Gaussian Copula American Call Basket Price:  24.5711
Gaussian Copula American Put  Basket Price:  17.4229

t Copula American Call Basket Price:  22.6220
t Copula American Put  Basket Price:  20.9983
```

This analysis represents only a small-scale simulation. If the simulation is repeated with 100,000 trials, the following results are obtained:

```
# of Monte Carlo Trials:      100000
# of Time Periods/Trial:      63

Brownian Motion American Call Basket Price:  20.2214
Brownian Motion American Put  Basket Price:  16.5355

Gaussian Copula American Call Basket Price:  20.6097
Gaussian Copula American Put  Basket Price:  16.5539

t Copula American Call Basket Price:  21.1273
t Copula American Put  Basket Price:  16.6873
```

Interestingly, the results agree closely. Put option prices obtained from copulas exceed those of Brownian motion by less than 1%.

A Note on Volatility and Interest Rate Scaling

The same option prices could also be obtained by working with unannualized (in this case, daily) centered returns and riskless rates, where the time increment $dt = 1$ day rather than $1/252$ years. In other words, portfolio prices would still be simulated every trading day; the data is simply scaled differently.

Although not executed, and by first resetting the random stream to its initial internal state, the following code segments work with daily centered returns and riskless rates and produce the same option prices.

Gaussian Distribution/Brownian Motion & Daily Data:

```
reset(s)
```



```

f      = Example_LongstaffSchwartz(nPeriods, nTrials);
GBM1 = gbm(diag(r(ones(1,nIndices))/252), diag(std(returns)), 'StartState', X, ...
           'Correlation', correlation);

simByEuler(GBM1, nPeriods, 'nTrials' , nTrials, 'DeltaTime', 1, ...
           'Processes', f.LongstaffSchwartz);

BrownianMotionCallPrice = f.CallPrice(strike, r/252)
BrownianMotionPutPrice  = f.PutPrice (strike, r/252)

```

Gaussian Copula & Daily Data:

```

reset(s)

z      = Example_CopulaRNG(returns, nPeriods, 'Gaussian');
f      = Example_LongstaffSchwartz(nPeriods, nTrials);
GBM2 = gbm(diag(r(ones(1,nIndices))/252), eye(nIndices), 'StartState', X);

simByEuler(GBM2, nPeriods, 'nTrials' , nTrials, 'DeltaTime', 1, ...
           'Processes', f.LongstaffSchwartz , 'Z', z);

GaussianCopulaCallPrice = f.CallPrice(strike, r/252)
GaussianCopulaPutPrice  = f.PutPrice (strike, r/252)

```

t Copula & Daily Data:

```

reset(s)

z = Example_CopulaRNG(returns, nPeriods, 't');
f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM2, nPeriods, 'nTrials' , nTrials, 'DeltaTime', 1, ...
           'Processes', f.LongstaffSchwartz , 'Z', z);

tCopulaCallPrice = f.CallPrice(strike, r/252)
tCopulaPutPrice  = f.PutPrice (strike, r/252)

```


Functions — Alphabetical List

abs2active

Convert constraints from absolute to active format

Syntax

```
ActiveConSet = abs2active(AbsConSet, Index)
```

Description

`ActiveConSet = abs2active(AbsConSet, Index)` transforms a constraint matrix to an equivalent matrix expressed in active weight format (relative to the index).

Input Arguments

AbsConSet

Portfolio linear inequality constraint matrix expressed in absolute weight format. `AbsConSet` is formatted as $[A \ b]$ such that $A \cdot w \leq b$, where A is a number of constraints (`NCONSTRAINTS`) by number of assets (`NASSETS`) weight coefficient matrix, and b and w are column vectors of length `NASSETS`. The value w represents a vector of absolute asset weights whose elements sum to the total portfolio value. See the output `ConSet` from `portcons` for additional details about constraint matrices.

Index

`NASSETS`-by-1 vector of index portfolio weights. The sum of the index weights must equal the total portfolio value (for example, a standard portfolio optimization imposes a sum-to-one budget constraint).

Output Arguments

ActiveConSet

The transformed portfolio linear inequality constraint matrix expressed in active weight format, also of the form $[A \ b]$ such that $A^*w \leq b$. The value w represents a vector of active asset weights (relative to the index portfolio) whose elements sum to zero.

Definitions

`abs2active` transforms a constraint matrix to an equivalent matrix expressed in active weight format (relative to the index). The transformation equation is

$$Aw_{absolute} = A(w_{active} + w_{index}) \leq b_{absolute}.$$

Therefore

$$Aw_{active} \leq b_{absolute} - Aw_{index} = \hat{b}_{active}.$$

The initial constraint matrix consists of `NCONSTRAINTS` portfolio linear inequality constraints expressed in absolute weight format. The index portfolio vector contains `NASSETS` assets.

Examples

Set up constraints for a portfolio optimization for portfolio `w0` with constraints in the form $A^*w \leq b$, where w is absolute portfolio weights. (Absolute weights do not depend on the tracking portfolio.) Use `abs2active` to convert constraints in terms of absolute weights into constraints in terms of active portfolio weights, defined relative to the tracking portfolio `w0`. Assume three assets with the following mean and covariance of asset returns:

```
m = [ 0.14; 0.10; 0.05 ];
C = [ 0.29^2 0.4*0.29*0.17 0.1*0.29*0.08; 0.4*0.29*0.17 0.17^2 0.3*0.17*0.08; ...
      0.1*0.29*0.08 0.3*0.17*0.08 0.08^2 ];
```

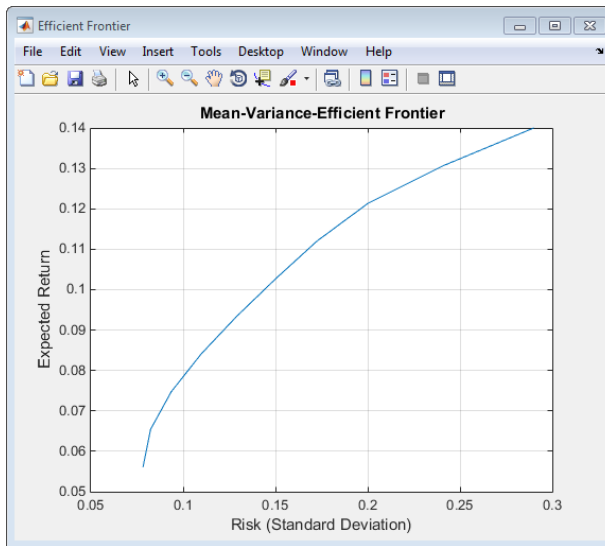
Absolute portfolio constraints are the typical ones (weights sum to 1 and fall from 0 through 1), create the `A` and `b` matrices using `portcons`:

```
AbsCons = portcons('PortValue',1,3,'AssetLims', [0; 0; 0], [1; 1; 1]);
```

The efficient frontier is:

```
portopt(m, C, [], [], AbsCons);
```

This returns:



The tracking portfolio w0 is:

```
w0 = [ 0.1; 0.55; 0.35 ];
```

Use `abs2active` to compute the constraints for active portfolio weights:

```
ActCons = abs2active(AbsCons, w0)
```

This returns:

```
ActCons =
```

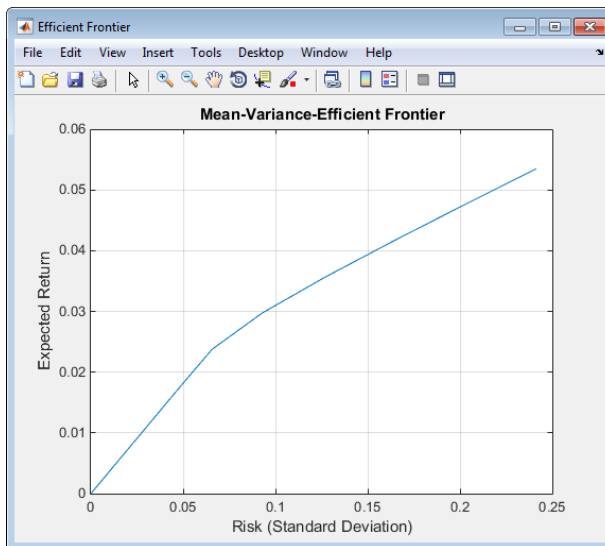
1.0000	1.0000	1.0000	0
-1.0000	-1.0000	-1.0000	0
1.0000	0	0	0.9000
0	1.0000	0	0.4500
0	0	1.0000	0.6500

-1.0000	0	0	0.1000
0	-1.0000	0	0.5500
0	0	-1.0000	0.3500

The efficient frontier demonstrates expected returns and risk relative to the tracking portfolio w_0 :

```
portopt(m, C, [], [], ActCons);
```

This returns:



See Also

[active2abs](#) | [pcalims](#) | [pcgcomp](#) | [pcglims](#) | [pcpval](#) | [portcons](#)

AbstractPortfolio class

Abstract portfolio object for portfolio optimization and analysis

Description

The mean-variance portfolio object (see `Portfolio`), the conditional value-at-risk portfolio object (see `PortfolioCVaR`), and the mean-absolute deviation object (`PortfolioMAD`) derive from the abstract class `AbstractPortfolio`.

Construction

There is no constructor for the abstract class. To construct a mean-variance portfolio object, see the `Portfolio` class, to construct a conditional value-at-risk (CVaR) portfolio object, see `PortfolioCVaR`, and to construct a mean-absolute deviation (MAD) portfolio object, see the `PortfolioMAD` class.

Properties

Name

Name for instance of the `Portfolio` or `PortfolioCVaR` object ([] or [string]).

Attributes:

<code>SetAccess</code>	<code>public</code>
<code>GetAccess</code>	<code>public</code>

NumAssets

Number of assets in the universe ([] or [integer scalar]).

Attributes:

<code>SetAccess</code>	<code>public</code>
------------------------	---------------------

GetAccess public

AssetList

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Attributes:

SetAccess public

GetAccess public

InitPort

Initial portfolio ([] or vector).

Attributes:

SetAccess public

GetAccess public

AInequality

Linear inequality constraint matrix ([] or [matrix]).

Attributes:

SetAccess public

GetAccess public

bInequality

Linear inequality constraint vector ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

AEquality

Linear equality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

bEquality

Linear equality constraint vector ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerBound

Lower-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

UpperBound

Upper-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerBudget

Lower-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

UpperBudget

Upper-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess public

GetAccess public

GroupMatrix

Group membership matrix ([] or [matrix]).

Attributes:

SetAccess public

GetAccess public

LowerGroup

Lower-bound group constraint ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

UpperGroup

Upper-bound group constraint ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

GroupA

Group A weights to be bounded by weights in group B ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

GroupB

Group B weights ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

LowerRatio

Minimum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

UpperRatio

Maximum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

Instance Hierarchy

The `AbstractPortfolio` class has three subclasses, `Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` that inherit properties and methods from the `AbstractPortfolio` class.

Attributes

Abstract	true
----------	------

To learn about attributes of classes, see “Class Attributes” in the MATLAB Object-Oriented Programming documentation.

Copy Semantics

Value. To learn how value classes affect copy operations, see “Copying Objects” in the MATLAB documentation.

Alternatives

You can perform portfolio optimization using a collection of special-purpose functions in Financial Toolbox. For more information, see “Portfolio Optimization Functions”.

See Also

PortfolioCVaR | PortfolioMAD | Portfolio

How To

- “Class Attributes”
- “Property Attributes”

accrfrac

Fraction of coupon period before settlement

Syntax

```
Fraction = accrfrac(Settle, Maturity)
Fraction = accrfrac(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
StartDate)
```

Description

`Fraction = accrfrac(Settle, Maturity)` returns the fraction of the coupon period before settlement.

`Fraction = accrfrac(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns the fraction of the coupon period before settlement with optional inputs.

Use `accrfrac` for computing accrued interest. `accrfrac` calculates accrued interest for bonds with regular or odd first or last coupon periods.

Input Arguments

Settle

Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

Default: If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

StartDate

Future implementation.

Output Arguments

Fraction

NBONDS-by-1 vector of accrued interest fractions.

Examples

Find Accrued Interest for a Bond

This example shows how to find the accrued interest for given bond data.

```
Settle = '14-Mar-1997';  
Maturity = ['30-Nov-2000'  
            '31-Dec-2000'  
            '31-Jan-2001'];
```



```
Period = 2;  
Basis = 0;  
EndMonthRule = 1;
```

```
Fraction = accrfrac(Settle, Maturity, Period, Basis,...  
EndMonthRule)
```

```
Fraction =
```

```
0.5714  
0.4033  
0.2320
```

See Also

[cfdates](#) | [cfamounts](#) | [cpncount](#) | [cpndaten](#) | [cpndatenq](#) | [cpndatep](#) | [cpndatepq](#) | [cpndaysn](#) | [cpndaysp](#) | [cpnpersz](#)

acrubond

Accrued interest of security with periodic interest payments

Syntax

```
AccruInterest = acrubond(IssueDate, Settle, FirstCouponDate, Face,
CouponRate, Period, Basis)
```

Arguments

IssueDate	Enter as serial date number or date string.
Settle	Enter as serial date number or date string.
FirstCouponDate	Enter as serial date number or date string.
Face	Redemption (par, face) value.
CouponRate	Enter as decimal fraction.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Description

`AccruInterest = acrubond(IssueDate, Settle, FirstCouponDate, Face, CouponRate, Period, Basis)` returns the accrued interest for a security with periodic interest payments. This function computes the accrued interest for securities with standard, short, and long first coupon periods.

Note `cfamounts` or `accrfrac` is recommended when calculating accrued interest beyond the first period.

Examples

Find Accrued Interest of a Bond with Periodic Interest Payments

This example shows how to find the accrued interest for a bond with semiannual interest payments.

```
AccruInterest = acrubond('31-jan-1983', '1-mar-1993', ...
    '31-jul-1983', 100, 0.1, 2, 0)
```

```
AccruInterest =
```

```
0.8011
```

See Also

`accrfrac` | `acrudisc` | `bndprice` | `bndyield` | `cfamounts` | `datenum`

acrudisc

Accrued interest of discount security paying at maturity

Syntax

AccruInterest = acrudisc(Settle, Maturity, Face, Discount, Period, Basis)

Arguments

Settle	Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Enter as serial date number or date string.
Face	Redemption (par, face) value.
Discount	Discount rate of the security. Enter as decimal fraction.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA)

- 12 = actual/actual (ISDA)
 - 13 = BUS/252
- For more information, see **basis**.

Description

AccruInterest = acrudisc(Settle, Maturity, Face, Discount, Period, Basis) returns the accrued interest of a discount security paid at maturity.

Examples

Find Accrued Interest of a Discount Security Paid at Maturity

This example shows how to find the accrued interest of a discount security paid at maturity.

```
AccruInterest = acrudisc('05/01/1992', '07/15/1992', ...
                        100, 0.1, 2, 0)
```

```
AccruInterest =
```

```
2.0604
```

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula D.

See Also

acrubond | yldmat | prdisc | prmat | ylddisc

active2abs

Convert constraints from active to absolute format

Syntax

```
AbsConSet = active2abs(ActiveConSet, Index)
```

Arguments

ActiveConSet	Portfolio linear inequality constraint matrix expressed in active weight format. ActiveConSet is formatted as $[A \ b]$ such that $A \cdot w \leq b$, where A is a number of constraints (NCONSTRAINTS) by number of assets (NASSETS) weight coefficient matrix, and b and w are column vectors of length NASSETS . The value w represents a vector of active asset weights (relative to the index portfolio) whose elements sum to 0. See the output ConSet from portcons for additional details about constraint matrices.
Index	NASSETS -by-1 vector of index portfolio weights. The sum of the index weights must equal the total portfolio value (for example, a standard portfolio optimization imposes a sum-to-one budget constraint).

Description

`AbsConSet = active2abs(ActiveConSet, Index)` transforms a constraint matrix to an equivalent matrix expressed in absolute weight format. The transformation equation is

$$A w_{active} = A (w_{absolute} - w_{index}) \leq b_{active}.$$

Therefore

$$Aw_{absolute} \leq b_{active} + Aw_{index} = b_{absolute}.$$

The initial constraint matrix consists of NCONSTRAINTS portfolio linear inequality constraints expressed in active weight format (relative to the index portfolio). The index portfolio vector contains NASSETS assets.

AbsConSet is the transformed portfolio linear inequality constraint matrix expressed in absolute weight format, also of the form [A b] such that $A*w \leq b$. The value w represents a vector of active asset weights (relative to the index portfolio) whose elements sum to the total portfolio value.

See Also

abs2active | pcalims | pcgcomp | pcglims | pcpval | portcons

addEquality

Class: Portfolio

Add linear equality constraints for portfolio weights to existing constraints

Syntax

```
obj = addEquality(obj, AEquality, bEquality)
```

Description

`obj = addEquality(obj, AEquality, bEquality)` adds linear equality constraints for portfolio weights to existing constraints.

Given a linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

$$AEquality * Port = bEquality$$

This method "stacks" additional linear equality constraints onto any existing linear equality constraints that already exist in the input Portfolio object. If no constraints already exist, this method is the same as `setEquality`.

Tips

- You can also use dot notation to add the linear equality constraints for portfolio weights.

```
obj = obj.addEquality(AEquality, bEquality)
```

- You can also remove linear equality constraints from a Portfolio object using dot notation.

```
obj = obj.setEquality([], [])
```


Input Arguments

obj

Portfolio object [Portfolio].

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Note: It is an error if **AEquality** is empty and **bEquality** is non-empty or if **AEquality** is non-empty and **bEquality** is empty.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add a Linear Equality Constraint

Use the `addEquality` method to create linear equality constraints. Add another linear equality constraint to ensure that the last three assets constitute 50% of a portfolio.

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % First equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % Second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000
```

- “Working with Linear Equality Constraints” on page 4-69

See Also

Portfolio | setEquality

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

addEquality

Class: PortfolioCVaR

Add linear equality constraints for portfolio weights to existing constraints in PortfolioCVaR object

Syntax

```
obj = addEquality(obj, AEquality, bEquality)
```

Description

`obj = addEquality(obj, AEquality, bEquality)` adds linear equality constraints for portfolio weights to existing constraints.

Given a linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

$$AEquality * Port = bEquality$$

This method “stacks” additional linear equality constraints onto any existing linear equality constraints that already exist in the input PortfolioCVaR object. If no constraints already exist, this method is the same as `setEquality`.

Tips

- You can also use dot notation to add the linear equality constraints for portfolio weights.

```
obj = obj.addEquality(AEquality, bEquality)
```

- You can also remove linear equality constraints from a CVaR portfolio object using dot notation.

```
obj = obj.setEquality([ ], [ ])
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Note: An error results if **AEquality** is empty and **bEquality** is nonempty or if **AEquality** is nonempty and **bEquality** is empty.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add a Linear Equality Constraint

Use the `addEquality` method to create linear equality constraints. Add another linear equality constraint to ensure that the last three assets constitute 50% of a portfolio.

```
p = PortfolioCVaR;
A = [ 1 1 1 0 0 ]; % First equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ]; % Second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

5

1 1 1 0 0
0 0 1 1 1

0.5000
0.5000
```

- “Working with Linear Equality Constraints” on page 5-68

See Also

PortfolioCVaR | setEquality

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

addEquality

Class: PortfolioMAD

Add linear equality constraints for portfolio weights to existing constraints in PortfolioMAD object

Syntax

```
obj = addEquality(obj, AEquality, bEquality)
```

Description

`obj = addEquality(obj, AEquality, bEquality)` adds linear equality constraints for portfolio weights to existing constraints.

Given a linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

$$AEquality * Port = bEquality$$

This method “stacks” additional linear equality constraints onto any existing linear equality constraints that already exist in the input PortfolioMAD object. If no constraints already exist, this method is the same as `setEquality`.

Tips

- You can also use dot notation to add the linear equality constraints for portfolio weights.

```
obj = obj.addEquality(AEquality, bEquality)
```

- You can also remove linear equality constraints from a MAD portfolio object using dot notation.

```
obj = obj.setEquality([ ], [ ])
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Note: An error results if **AEquality** is empty and **bEquality** is nonempty or if **AEquality** is nonempty and **bEquality** is empty.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add a Linear Equality Constraint

Use the `addEquality` method to create linear equality constraints. Add another linear equality constraint to ensure that the last three assets constitute 50% of a portfolio.

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ]; % First equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ]; % Second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);

5

1 1 1 0 0
0 0 1 1 1

0.5000
0.5000
```

- “Setting Linear Inequality Constraints Using the setInequality and addInequality Methods” on page 6-71

See Also

PortfolioMAD | setEquality

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

addGroupRatio

Class: Portfolio

Add group ratio constraints for portfolio weights to existing group ratio constraints

Syntax

```
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,
UpperRatio)
```

Description

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)` adds group ratio constraints for the portfolio weights to existing group ratio constraints with just a lower bound on the ratio between groups.

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` adds group ratio constraints for the portfolio weights to existing group ratio constraints with an additional option for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and, either `LowerRatio`, or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Tips

- You can also use dot notation to add group ratio constraints for the portfolio weights to existing group ratio constraints.

```
obj = obj.addGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio)
```

- To remove group ratio constraints from a Portfolio object using dot notation, enter empty arrays for the corresponding arrays.

Input Arguments

obj

Portfolio object [Portfolio].

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

Note: The group matrices **GroupA** and **GroupB** are usually indicators of membership in groups, which means that their elements are usually either 0 or 1. Because of this interpretation, the **GroupA** and **GroupB** matrices can be logical or numerical arrays.

LowerRatio

Lower-bound for ratio of **GroupB** groups to **GroupA** groups [vector].

Note: If input is scalar, **LowerRatio** undergoes scalar expansion to be conformable with the group matrices.

UpperRatio

(Optional) Upper-bound for ratio of **GroupB** groups to **GroupA** groups [vector].

Note: If input is scalar, **UpperRatio** undergoes scalar expansion to be conformable with the group matrices.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Group Ratio Constraints

Set a group ratio constraint to ensure that the weight in financial assets does not exceed 50% of the weight in nonfinancial assets. Then add another group ratio constraint to ensure that the weight in financial assets constitute at least 20% of the weight in nonfinancial assets of the portfolio.

```
p = Portfolio;
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ]; % odd-numbered companies
GB = [ false false false true true true ]; % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);
```

```
6
1    1    1    0    0    0
1    0    1    0    1    0
0    0    0    1    1    1
0    0    0    1    1    1
-Inf
0.2000
0.5000
Inf
```

- “Working with Group Ratio Constraints” on page 4-66

See Also

Portfolio | setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

addGroupRatio

Class: PortfolioCVaR

Add group ratio constraints for portfolio weights to existing group ratio constraints for PortfolioCVaR object

Syntax

```
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,
UpperRatio)
```

Description

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)` adds group ratio constraints for the portfolio weights to existing group ratio constraints with just a lower bound on the ratio between groups.

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` adds group ratio constraints for the portfolio weights to existing group ratio constraints with an additional option for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and, either `LowerRatio`, or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

$$(\text{GroupB} * \text{Port}) .* \text{LowerRatio} \leq \text{GroupA} * \text{Port} \leq (\text{GroupB} * \text{Port}) .* \text{UpperRatio}$$

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Tips

- You can also use dot notation to add group ratio constraints for the portfolio weights to existing group ratio constraints.

```
obj = obj.addGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio)
```

- To remove group ratio constraints from a CVaR portfolio object using dot notation, enter empty arrays for the corresponding arrays.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

Note: The group matrices **GroupA** and **GroupB** often indicate membership in groups, which means that their elements are usually either 0 or 1. Because of this interpretation, the **GroupA** and **GroupB** matrices can be logical or numerical arrays.

LowerRatio

Lower bound for ratio of **GroupB** groups to **GroupA** groups [vector].

Note: If input is scalar, **LowerRatio** undergoes scalar expansion to conform with the group matrices.

UpperRatio

(Optional) Upper bound for ratio of **GroupB** groups to **GroupA** groups [vector].

Note: If input is scalar, **UpperRatio** undergoes scalar expansion to conform with the group matrices.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Group Ratio Constraints

Set a group ratio constraint to ensure that the weight in financial assets does not exceed 50% of the weight in nonfinancial assets. Then add another group ratio constraint to ensure that the weight in financial assets constitute at least 20% of the weight in nonfinancial assets of the portfolio.

```
p = PortfolioCVaR;
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ]; % odd-numbered companies
GB = [ false false false true true true ]; % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);
```

```
6
1    1    1    0    0    0
1    0    1    0    1    0
0    0    0    1    1    1
0    0    0    1    1    1
-Inf
0.2000
0.5000
Inf
```

- “Working with Group Ratio Constraints” on page 5-66

See Also

PortfolioCVaR | setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

addGroupRatio

Class: PortfolioMAD

Add group ratio constraints for portfolio weights to existing group ratio constraints for PortfolioMAD object

Syntax

```
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,
UpperRatio)
```

Description

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)` adds group ratio constraints for the portfolio weights to existing group ratio constraints with just a lower bound on the ratio between groups.

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` adds group ratio constraints for the portfolio weights to existing group ratio constraints with an additional option for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and, either `LowerRatio`, or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

$$(\text{GroupB} * \text{Port}) .* \text{LowerRatio} \leq \text{GroupA} * \text{Port} \leq (\text{GroupB} * \text{Port}) .* \text{UpperRatio}$$

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Tips

- You can also use dot notation to add group ratio constraints for the portfolio weights to existing group ratio constraints.

```
obj = obj.addGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio)
```

- To remove group ratio constraints from a MAD portfolio object using dot notation, enter empty arrays for the corresponding arrays.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

Note: The group matrices **GroupA** and **GroupB** often indicate membership in groups, which means that their elements are usually either 0 or 1. Because of this interpretation, the **GroupA** and **GroupB** matrices can be logical or numerical arrays.

LowerRatio

Lower bound for ratio of **GroupB** groups to **GroupA** groups [vector].

Note: If input is scalar, **LowerRatio** undergoes scalar expansion to conform with the group matrices.

UpperRatio

(Optional) Upper bound for ratio of **GroupB** groups to **GroupA** groups [vector].

Note: If input is scalar, **UpperRatio** undergoes scalar expansion to conform with the group matrices.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Group Ratio Constraints

Set a group ratio constraint to ensure that the weight in financial assets does not exceed 50% of the weight in nonfinancial assets. Then add another group ratio constraint to ensure that the weight in financial assets constitute at least 20% of the weight in nonfinancial assets of the portfolio.

```
p = PortfolioMAD;
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ]; % odd-numbered companies
GB = [ false false false true true true ]; % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);
```

```
6
1    1    1    0    0    0
1    0    1    0    1    0
0    0    0    1    1    1
0    0    0    1    1    1
-Inf
0.2000
0.5000
Inf
```

- “Working with Group Ratio Constraints”

See Also

PortfolioMAD | setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

addGroups

Class: Portfolio

Add group constraints for portfolio weights to existing group constraints

Syntax

```
obj = addGroups(obj, GroupMatrix, LowerGroup)
```

```
obj = addGroups(obj, GroupMatrix, LowerGroup, UpperGroup)
```

Description

`obj = addGroups(obj, GroupMatrix, LowerGroup)` adds group constraints for portfolio weights to existing group constraints subject to a lower bound on groups.

`obj = addGroups(obj, GroupMatrix, LowerGroup, UpperGroup)` adds the group constraints for portfolio weights to existing group constraints with an additional option for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

$$\text{LowerGroup} \leq \text{GroupMatrix} * \text{Port} \leq \text{UpperGroup}$$

Tips

- You can also use dot notation to add group constraints for portfolio weights.
`obj = obj.addGroups(GroupMatrix, LowerGroup, UpperGroup)`
- To remove group constraints from a `Portfolio` object using dot notation, enter empty arrays for the corresponding arrays.

Input Arguments

obj

Portfolio object [Portfolio].

GroupMatrix

Group constraint matrix [matrix].

Note: The group matrix `GroupMatrix` often indicates membership in groups, which means that its elements are usually either 0 or 1. Because of this interpretation, `GroupMatrix` can be a logical or numerical matrix.

LowerGroup

Lower bound for group constraints [vector].

Note: If input is scalar, `LowerGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

UpperGroup

(Optional) Upper bound for group constraints [vector].

Note: If input is scalar, `UpperGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Group Constraints

Set a group constraint to ensure that the first three assets constitute at most 30% of a portfolio. Then add another group constraint to ensure that the odd-numbered assets constitute at least 20% of a portfolio.

```
p = Portfolio;
G = [ true true true false false ]; % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ]; % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
5
1     1     1     0     0
1     0     1     0     1

-Inf
0.2000

0.3000
Inf
```

- “Working with Group Constraints” on page 4-63

See Also

Portfolio | setGroups

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

addGroups

Class: PortfolioCVaR

Add group constraints for portfolio weights to existing group constraints for PortfolioCVaR object

Syntax

```
obj = addGroups(obj, GroupMatrix, LowerGroup)
obj = addGroups(obj, GroupMatrix, LowerGroup, UpperGroup)
```

Description

`obj = addGroups(obj, GroupMatrix, LowerGroup)` adds group constraints for portfolio weights to existing group constraints subject to a lower bound on groups.

`obj = addGroups(obj, GroupMatrix, LowerGroup, UpperGroup)` adds the group constraints for portfolio weights to existing group constraints with an additional option for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

$$\text{LowerGroup} \leq \text{GroupMatrix} * \text{Port} \leq \text{UpperGroup}$$

Tips

- You can also use dot notation to add group constraints for portfolio weights.

```
obj = obj.addGroups(GroupMatrix, LowerGroup, UpperGroup)
```

- To remove group constraints from a CVaR portfolio object using dot notation, enter empty arrays for the corresponding arrays.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

GroupMatrix

Group constraint matrix [matrix].

Note: The group matrix `GroupMatrix` often indicates membership in groups, which means that its elements are usually either 0 or 1. Because of this interpretation, `GroupMatrix` can be a logical or numerical matrix.

LowerGroup

Lower bound for group constraints [vector].

Note: If input is scalar, `LowerGroup` undergoes scalar expansion to conform with `GroupMatrix`.

UpperGroup

(Optional) Upper bound for group constraints [vector].

Note: If input is scalar, `UpperGroup` undergoes scalar expansion to conform with `GroupMatrix`.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Group Constraints

Set a group constraint to ensure that the first three assets constitute at most 30% of a portfolio. Then add another group constraint to ensure that the odd-numbered assets constitute at least 20% of a portfolio.

```
p = PortfolioCVaR;
G = [ true true true false false ]; % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ]; % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);

5

1 1 1 0 0
1 0 1 0 1

-Inf
0.2000

0.3000
Inf
```

- “Working with Group Constraints” on page 5-63

See Also

PortfolioCVaR | setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

addGroups

Class: PortfolioMAD

Add group constraints for portfolio weights to existing group constraints for PortfolioMAD object

Syntax

```
obj = addGroups(obj, GroupMatrix, LowerGroup)
obj = addGroups(obj, GroupMatrix, LowerGroup, UpperGroup)
```

Description

`obj = addGroups(obj, GroupMatrix, LowerGroup)` adds group constraints for portfolio weights to existing group constraints subject to a lower bound on groups.

`obj = addGroups(obj, GroupMatrix, LowerGroup, UpperGroup)` adds group constraints for portfolio weights to existing group constraints with an additional option for UpperGroup.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

$$\text{LowerGroup} \leq \text{GroupMatrix} * \text{Port} \leq \text{UpperGroup}$$

Tips

- You can also use dot notation to add group constraints for portfolio weights.

```
obj = obj.addGroups(GroupMatrix, LowerGroup, UpperGroup)
```

- To remove group constraints from a MAD portfolio object using dot notation, enter empty arrays for the corresponding arrays.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

GroupMatrix

Group constraint matrix [matrix].

Note: The group matrix `GroupMatrix` often indicates membership in groups, which means that its elements are usually either 0 or 1. Because of this interpretation, `GroupMatrix` can be a logical or numerical matrix.

LowerGroup

Lower bound for group constraints [vector].

Note: If input is scalar, `LowerGroup` undergoes scalar expansion to conform with `GroupMatrix`.

UpperGroup

(Optional) Upper bound for group constraints [vector].

Note: If input is scalar, `UpperGroup` undergoes scalar expansion to conform with `GroupMatrix`.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Group Constraints

Set a group constraint to ensure that the first three assets constitute at most 30% of a portfolio. Then add another group constraint to ensure that the odd-numbered assets constitute at least 20% of a portfolio.

```
p = PortfolioMAD;
G = [ true true true false false ]; % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ]; % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
5
```

```
1 1 1 0 0
1 0 1 0 1
```

```
-Inf
0.2000
```

```
0.3000
Inf
```

- “Working with Group Constraints”

See Also

PortfolioMAD | setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

addInequality

Class: Portfolio

Add linear inequality constraints for portfolio weights to existing constraints

Syntax

```
obj = addInequality(obj,AInequality,bInequality)
```

Description

`obj = addInequality(obj,AInequality,bInequality)` adds linear inequality constraints for portfolio weights to existing constraints.

Given linear inequality constraint matrix `AInequality` and vector `bInequality`, every weight in a portfolio `Port` must satisfy the following:

$$AInequality * Port \leq bInequality$$

Tips

- You can also use dot notation to add linear inequality constraints for portfolio weights to existing constraints.

```
obj = obj.addInequality(AInequality, bInequality)
```

- You can remove linear inequality constraints for portfolio weights from a Portfolio object using dot notation.

```
obj = obj.setInequality([ ], [ ])
```

Input Arguments

obj

Portfolio object [Portfolio].

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Note: An error results if **AInequality** is empty and **bInequality** is nonempty, or if **AInequality** is nonempty and **bInequality** is empty.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Linear Inequality Constraint

Set a linear inequality constraint to ensure that the first three assets constitute at most 50% of a portfolio. Then add another linear inequality constraint to ensure that the last three assets constitute at least 50% of a portfolio.

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % first inequality constraint
```

```
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);

    5

    1     1     1     0     0
    0     0    -1    -1    -1

    0.5000
   -0.5000
```

- “Working with Linear Inequality Constraints” on page 4-71

See Also

Portfolio | setInequality

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

addInequality

Class: PortfolioCVaR

Add linear inequality constraints for portfolio weights to existing constraints for PortfolioCVaR object

Syntax

```
obj = addInequality(obj,AInequality,bInequality)
```

Description

`obj = addInequality(obj,AInequality,bInequality)` adds linear inequality constraints for portfolio weights to existing constraints.

Given linear inequality constraint matrix `AInequality` and vector `bInequality`, every weight in portfolio `Port` must satisfy the following:

$$AInequality * Port \leq bInequality$$

Tips

- You can also use dot notation to add linear inequality constraints for portfolio weights to existing constraints.

```
obj = obj.addInequality(AInequality, bInequality)
```

- You can remove linear inequality constraints for portfolio weights from a CVaR portfolio object using dot notation.

```
obj = obj.setInequality([ ], [ ])
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Note: An error results if **AInequality** is empty and **bInequality** is nonempty, or if **AInequality** is nonempty and **bInequality** is empty.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Linear Inequality Constraint

Set a linear inequality constraint to ensure that the first three assets constitute at most 50% of a portfolio. Then add another linear inequality constraint to ensure that the last three assets constitute at least 50% of a portfolio.

```
p = PortfolioCVaR;  
A = [ 1 1 1 0 0 ]; % first inequality constraint
```

```
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);

    5

    1     1     1     0     0
    0     0    -1    -1    -1

    0.5000
   -0.5000
```

- “Working with Linear Inequality Constraints” on page 5-70

See Also

PortfolioCVaR | setInequality

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

addInequality

Class: PortfolioMAD

Add linear inequality constraints for portfolio weights to existing constraints for PortfolioMAD object

Syntax

```
obj = addInequality(obj,AInequality,bInequality)
```

Description

`obj = addInequality(obj,AInequality,bInequality)` adds linear inequality constraints for portfolio weights to existing constraints.

Given linear inequality constraint matrix `AInequality` and vector `bInequality`, every weight in portfolio `Port` must satisfy the following:

$$AInequality * Port \leq bInequality$$

`addInequality` “stacks” additional linear inequality constraints onto any existing linear inequality constraints that already exist in the input PortfolioMAD object. If no constraints already exist, this method is the same as `setInequality`.

Tips

- You can also use dot notation to add linear inequality constraints for portfolio weights to existing constraints.

```
obj = obj.addInequality(AInequality, bInequality)
```

- You can remove linear inequality constraints for portfolio weights from a MAD portfolio object using dot notation.

```
obj = obj.setInequality([ ], [ ])
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Note: An error results if **AInequality** is empty and **bInequality** is nonempty, or if **AInequality** is nonempty and **bInequality** is empty.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Add Linear Inequality Constraint

Set a linear inequality constraint to ensure that the first three assets constitute at most 50% of a portfolio. Then add another linear inequality constraint to ensure that the last three assets constitute at least 50% of a portfolio.

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ]; % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ]; % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);

5

1 1 1 0 0
0 0 -1 -1 -1

0.5000
-0.5000
```

- “Working with Linear Inequality Constraints” on page 6-70

See Also

PortfolioMAD | setInequality

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

adline

Accumulation/Distribution line

Syntax

```
adln = adline(highp, lowp, closep, tvolume)
```

```
adln = adline([highp lowp closep tvolume])
```

```
adlnts = adline(tsobj)
```

```
adlnts = adline(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

highp	High price (vector)
lowp	Low price (vector)
closep	Closing price (vector)
tvolume	Volume traded (vector)
tsobj	Time series object

Description

`adln = adline(highp, lowp, closep, tvolume)` computes the Accumulation/Distribution line for a set of stock price and volume traded data. The prices required for this function are the high (**highp**), low (**lowp**), and closing (**closep**) prices.

`adln = adline([highp lowp closep tvolume])` accepts a four-column matrix as input. The first column contains the high prices, the second contains the low prices, the third contains the closing prices, and the fourth contains the volume traded.

`adlnts = adline(tsobj)` computes the Williams Accumulation/Distribution line for a set of stock price data contained in the financial time series object **tsobj**. The object

must contain the high, low, and closing prices plus the volume traded. The function assumes that the series are named `High`, `Low`, `Close`, and `Volume`. All are required. `adlnts` is a financial time series object with the same dates as `tsobj` but with a single series named `ADLine`.

`adlnts = adline(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

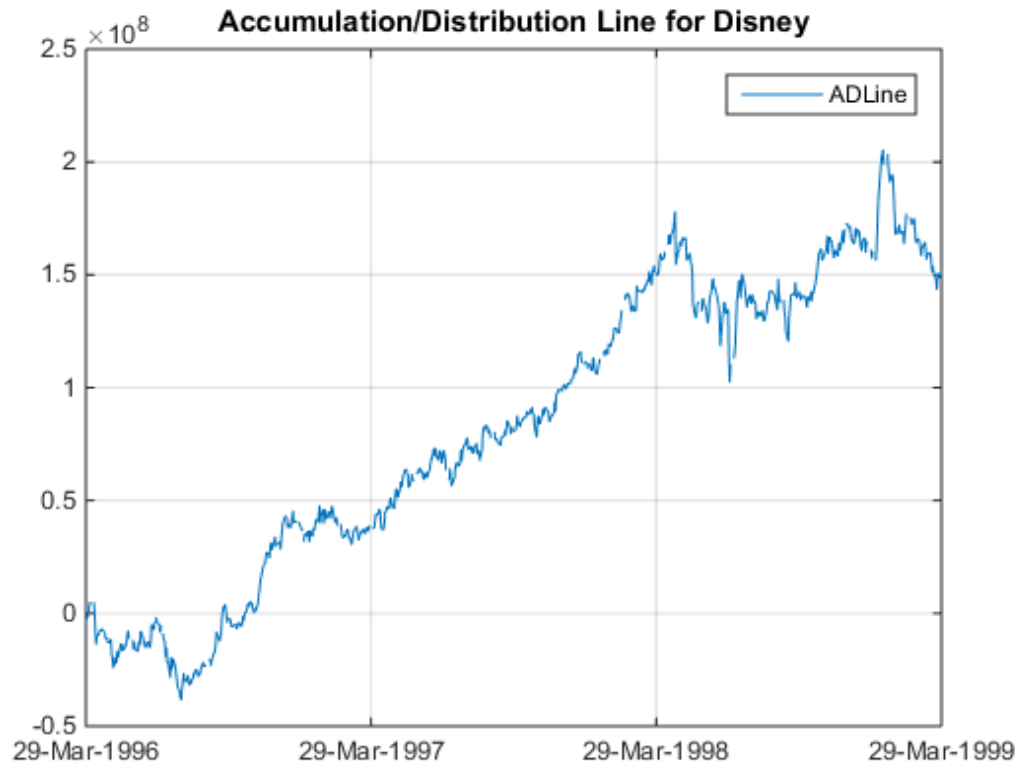
Parameter values are the strings that represent the valid parameter names.

Examples

Calculate the Accumulation/Distribution Line for a Stock

This example shows how to compute the Accumulation/Distribution line for Disney stock and plot the results.

```
load disney.mat
dis_ADLine = adline(dis);
plot(dis_ADLine)
title('Accumulation/Distribution Line for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second Edition, McGraw-Hill, 1995, pp. 56-58.

See Also

adosc | willad | willpctr

adosc

Accumulation/Distribution oscillator

Syntax

```
ado = adosc(highp, lowp, openp, closep)
```

```
ado = adosc([highp lowp openp closep])
```

```
adots = adosc(tsobj)
```

```
adots = adosc(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

highp	High price (vector)
lowp	Low price (vector)
openp	Opening price (vector)
closep	Closing price (vector)
tsobj	Time series object

Description

`ado = adosc(highp, lowp, openp, closep)` returns a vector, `ado`, that represents the Accumulation/Distribution (A/D) oscillator. The A/D oscillator is calculated based on the high, low, opening, and closing prices of each period. Each period is treated individually.

`ado = adosc([highp lowp openp closep])` accepts a four-column matrix as input. The order of the columns must be high, low, opening, and closing prices.

`adots = adosc(tsobj)` calculates the Accumulation/Distribution (A/D) oscillator, `adots`, for the set of stock price data contained in the financial time series object `tsobj`.

The object must contain the high, low, opening, and closing prices. The function assumes that the series are named `High`, `Low`, `Open`, and `Close`. All are required. `adots` is a financial time series object with similar dates to `tsobj` and only one series named `ADOSC`.

`adots = adosc(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name-parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `OpenName`: opening prices series name
- `CloseName`: closing prices series name

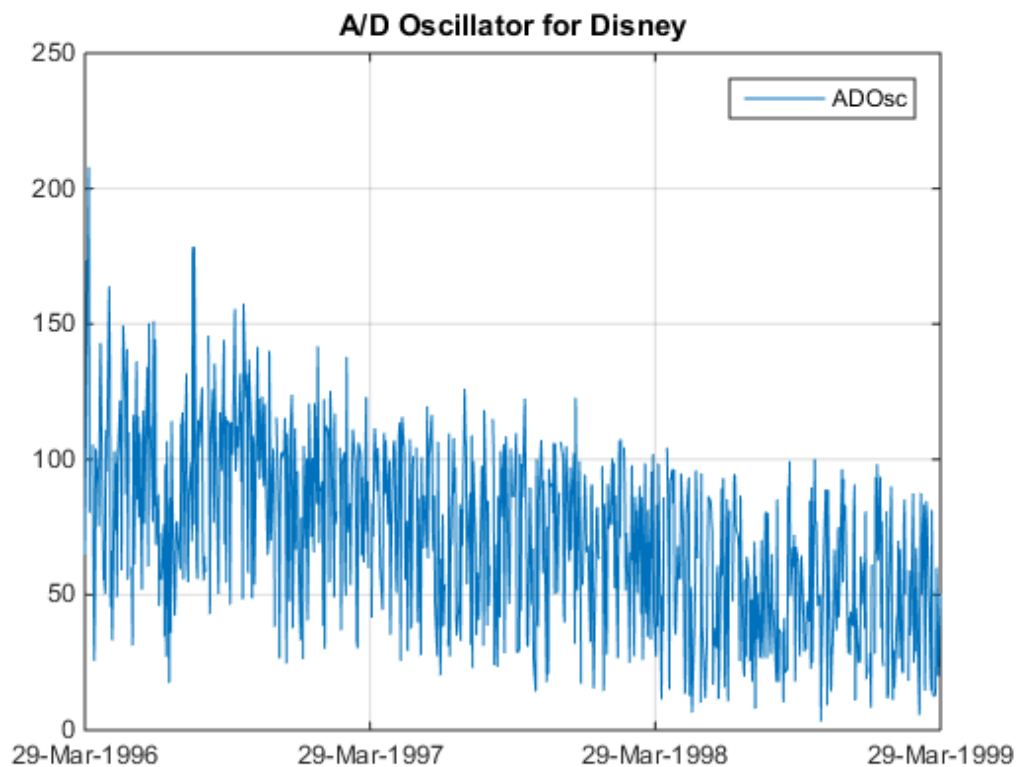
Parameter values are the strings that represents the valid parameter names.

Examples

Calculate the Accumulation/Distribution Oscillator for a Stock

This example shows how to find the Accumulation/Distribution oscillator for Disney stock and plot the results.

```
load disney.mat
dis_ADOsc = adosc(dis);
plot(dis_ADOsc)
title('A/D Oscillator for Disney')
```



References

Kaufman, P. J., *The New Commodity Trading Systems and Methods*, John Wiley and Sons, New York, 1987.

See Also

adline | willad

amortize

Amortization schedule

Syntax

```
[Principal, Interest, Balance, Payment] = amortize(Rate,
NumPeriods, PresentValue, FutureValue, Due)
```

Arguments

Rate	Interest rate per period, as a decimal fraction.
NumPeriods	Number of payment periods.
PresentValue	Present value of the loan.
FutureValue	(Optional) Future value of the loan. Default = 0.
Due	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

[Principal, Interest, Balance, Payment] = amortize(Rate, NumPeriods, PresentValue, FutureValue, Due) returns the principal and interest payments of a loan, the remaining balance of the original loan amount, and the periodic payment.

Principal	Principal paid in each period. A 1-by-NumPeriods vector.
Interest	Interest paid in each period. A 1-by-NumPeriods vector.
Balance	Remaining balance of the loan in each payment period. A 1-by-NumPeriods vector.
Payment	Payment per period. A scalar.

Examples

Compute an Amortization Schedule for a Conventional 30-Year, Fixed-Rate Mortgage With Fixed Monthly Payments

Compute an amortization schedule for a conventional 30-year, fixed-rate mortgage with fixed monthly payments and assume a fixed rate of 12% APR and an initial loan amount of \$100,000.

```
Rate          = 0.12/12;    % 12 percent APR = 1 percent per month
NumPeriods    = 30*12;     % 30 years = 360 months
PresentValue  = 100000;
```

```
[Principal, Interest, Balance, Payment] = amortize(Rate, ...
NumPeriods, PresentValue);
```

The output argument `Payment` contains the fixed monthly payment.

```
format bank
```

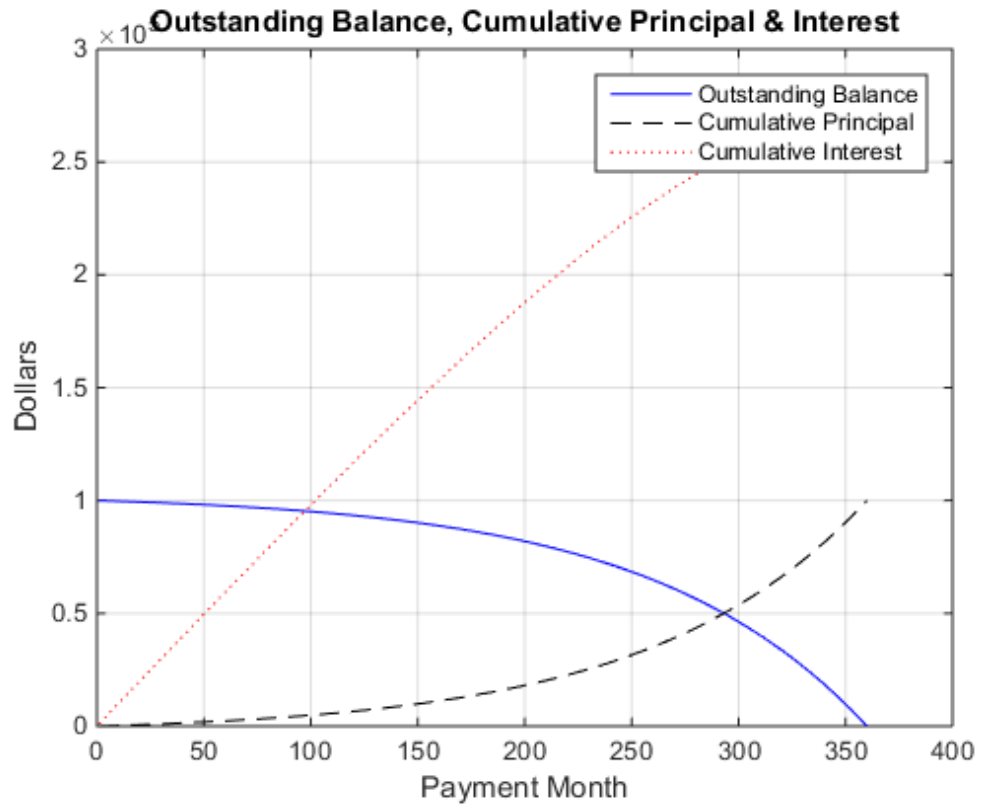
```
Payment
```

```
Payment =
    1028.61
```

Summarize the amortization schedule graphically by plotting the current outstanding loan balance, the cumulative principal, and the interest payments over the life of the mortgage. In particular, note that total interest paid over the life of the mortgage exceeds \$270,000, far in excess of the original loan amount.

```
plot(Balance, 'b'), hold('on')
plot(cumsum(Principal), '--k')
plot(cumsum(Interest), ':r')

xlabel('Payment Month')
ylabel('Dollars')
grid('on')
title('Outstanding Balance, Cumulative Principal & Interest')
legend('Outstanding Balance', 'Cumulative Principal', ...
'Cumulative Interest')
```

The solid blue line represents the declining principal over the 30-year period. The dotted red line indicates the increasing cumulative interest payments. Finally, the dashed black line represents the cumulative principal payments, reaching \$100,000 after 30 years.

See Also

annurate | payadv | payodd | payper | annuterm

annurate

Periodic interest rate of annuity

Syntax

Rate = annurate(NumPeriods, Payment, PresentValue, FutureValue, Due)

Arguments

NumPeriods	Number of payment periods.
Payment	Payment per period.
PresentValue	Present value of the loan or annuity.
FutureValue	(Optional) Future value of the loan or annuity. Default = 0.
Due	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

Rate = annurate(NumPeriods, Payment, PresentValue, FutureValue, Due) returns the periodic interest rate paid on a loan or annuity.

Examples

Calculate the Periodic Interest Rate Paid on a Loan or Annuity

This example shows how to find the periodic interest rate of a four-year, \$5000 loan with a \$130 monthly payment made at the end of each month.

Rate = annurate(4*12, 130, 5000, 0, 0)

Rate =

0.0094

Rate multiplied by 12 gives an annual interest rate of 11.32% on the loan.

See Also

`amortize` | `annuterm` | `bndyield` | `irr`

annuterm

Number of periods to obtain value

Syntax

NumPeriods = annuterm(Rate, Payment, PresentValue, FutureValue, Due)

Arguments

Rate	Interest rate per period, as a decimal fraction.
Payment	Payment per period.
PresentValue	Present value.
FutureValue	(Optional) Future value. Default = 0.
Due	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

NumPeriods = annuterm(Rate, Payment, PresentValue, FutureValue, Due) calculates the number of periods needed to obtain a future value. To calculate the number of periods needed to pay off a loan, enter the payment or the present value as a negative value.

Examples

Calculate the Number of Periods Needed to Obtain a Future Value

This example shows a savings account with a starting balance of \$1500. \$200 is added at the end of each month and the account pays 9% interest, compounded monthly. How many years will it take to save \$5,000?

NumPeriods = annuterm(0.09/12, 200, 1500, 5000, 0)

NumPeriods =

15.6752

See Also

annurate | fvfix | pvfix | amortize

arith2geom

Arithmetic to geometric moments of asset returns

Syntax

```
[mg, Cg] = arith2geom(ma, Ca)
```

```
[mg, Cg] = arith2geom(ma, Ca, t)
```

Arguments

ma	Arithmetic mean of asset-return data (n-vector).
Ca	Arithmetic covariance of asset-return data (n-by-n symmetric, positive-semidefinite matrix).
t	(Optional) Target period of geometric moments in terms of periodicity of arithmetic moments with default value 1 (scalar).

Description

`arith2geom` transforms moments associated with a simple Brownian motion into equivalent continuously-compounded moments associated with a geometric Brownian motion with a possible change in periodicity.

`[mg, Cg] = arith2geom(ma, Ca, t)` returns `mg`, continuously-compounded or "geometric" mean of asset returns over the target period (n-vector), and `Cg`, which is a continuously-compounded or "geometric" covariance of asset returns over the target period (n-by-n matrix).

Arithmetic returns over period t_A are modeled as multivariate normal random variables with moments

$$E[X] = m_A$$

and

$$\text{cov}(\mathbf{X}) = \mathbf{C}_A$$

Geometric returns over period t_G are modeled as multivariate lognormal random variables with moments

$$E[\mathbf{Y}] = 1 + \mathbf{m}_G$$

$$\text{cov}(\mathbf{Y}) = \mathbf{C}_G$$

Given $t = t_G / t_A$, the transformation from geometric to arithmetic moments is

$$1 + m_{G_i} = \exp(tm_{A_i} + \frac{1}{2}tC_{A_{ii}})$$

$$C_{G_{ij}} = (1 + m_{G_i})(1 + m_{G_j})(\exp(tC_{A_{ij}}) - 1)$$

For $i, j = 1, \dots, n$.

Note: If $t = 1$, then $\mathbf{Y} = \exp(\mathbf{X})$.

This function has no restriction on the input mean \mathbf{m}_A but requires the input covariance \mathbf{C}_A to be a symmetric positive-semidefinite matrix.

The functions `arith2geom` and `geom2arith` are complementary so that, given \mathbf{m} , \mathbf{C} , and t , the sequence

$$\begin{aligned} [\mathbf{m}_G, \mathbf{C}_G] &= \text{arith2geom}(\mathbf{m}, \mathbf{C}, t); \\ [\mathbf{m}_A, \mathbf{C}_A] &= \text{geom2arith}(\mathbf{m}_G, \mathbf{C}_G, 1/t); \end{aligned}$$

yields $\mathbf{m}_A = \mathbf{m}$ and $\mathbf{C}_A = \mathbf{C}$.

Examples

Example 1. Given arithmetic mean \mathbf{m} and covariance \mathbf{C} of monthly total returns, obtain annual geometric mean \mathbf{m}_G and covariance \mathbf{C}_G . In this case, the output period (1 year) is 12 times the input period (1 month) so that $t = 12$ with

```
[mg, Cg] = arith2geom(m, C, 12);
```

Example 2. Given annual arithmetic mean m and covariance C of asset returns, obtain monthly geometric mean mg and covariance Cg . In this case, the output period (1 month) is $1/12$ times the input period (1 year) so that $t = 1/12$ with

```
[mg, Cg] = arith2geom(m, C, 1/12);
```

Example 3. Given arithmetic means m and standard deviations s of daily total returns (derived from 260 business days per year), obtain annualized continuously-compounded mean mg and standard deviations sg with

```
[mg, Cg] = arith2geom(m, diag(s.^2), 260);  
sg = sqrt(diag(Cg));
```

Example 4. Given arithmetic mean m and covariance C of monthly total returns, obtain quarterly continuously-compounded return moments. In this case, the output is 3 of the input periods so that $t = 3$ with

```
[mg, Cg] = arith2geom(m, C, 3);
```

Example 5. Given arithmetic mean m and covariance C of 1254 observations of daily total returns over a 5-year period, obtain annualized continuously-compounded return moments. Since the periodicity of the arithmetic data is based on 1254 observations for a 5-year period, a 1-year period for geometric returns implies a target period of $t = 1254/5$ so that

```
[mg, Cg] = arith2geom(m, C, 1254/5);
```

See Also

geom2arith

ascii2fts

Create financial time series object from ASCII file

Syntax

```
tsobj = ascii2fts(filename, descrow, colheadrow, skiprows)
```

```
tsobj = ascii2fts(filename, timedata, descrow, colheadrow, skiprows)
```

Arguments

filename	ASCII data file
descrow	(Optional) Row number in the data file that contains the description to be used for the description field of the financial time series object
colheadrow	(Optional) Row number that has the column headers/names
skiprows	(Optional) Scalar or vector of row numbers to be skipped in the data file
timedata	Set to 'T' if time-of-day data is present in the ASCII data file or to 'NT' if no time-of-day data is present.

Description

`tsobj = ascii2fts(filename, descrow, colheadrow, skiprows)` creates a financial time series object `tsobj` from the ASCII file named `filename`. This form of the function can only read a data file without time-of-day information and create a financial time series object without time information. If time information is present in the ASCII file, an error message appears.

The general format of the text data file is

- Can contain header text lines.

- Can contain column header information. The column header information must immediately precede the data series columns unless `skiprows` is specified.
- Leftmost column must be the date column.
- Dates must be in a valid date string format:
 - `'ddmmyy'` or `'ddmmyyyy'`
 - `'mm/dd/yy'` or `'mm/dd/yyyy'`
 - `'dd-mmm-yy'` or `'dd-mmm-yyyy'`
 - `'mmm.dd,yy'` or `'mmm.dd,yyyy'`
- Time information must be in `'hh:mm'` format.
- Each column must be separated either by spaces or a tab.

`tsobj = ascii2fts(filename, timedata, descrow, colheadrow, skiprows)` creates a financial time series object containing time-of-day data. Set `timedata` to `'T'` to create a financial time series object containing time-of-day data. Note that the `ascii` time information must be in `'hh:mm'` format for `ascii2fts`.

Examples

Example 1. If your data file contains no description or column header rows,

```
1/3/95  36.75  36.9063  36.6563  36.875  1167900
1/4/95  37     37.2813  36.625  37.1563  1994700  ...
```

you can create a financial time series object from it with the simplest form of the `ascii2fts` function:

```
myinc = ascii2fts('my_inc.dat');

myinc =

desc:  my_inc.dat
freq:  Unknown (0)

'dates: (2)'  'series1: (2)'  'series2: (2)'  'series3: (2)'...
'03-Jan-1995' [  36.7500]      [  36.9063]      [  36.6563]
'04-Jan-1995' [           37]      [  37.2813]      [  36.6250]
```

Example 2: If your data file contains description and column header information with the data series immediately following the column header row,

```

International Business Machines Corporation (IBM)
Daily prices (1/3/95 to 4/5/99)
DATE      OPEN      HIGH      LOW      CLOSE      VOLUME
1/3/95    36.75    36.9063   36.6563   36.875    1167900
1/4/95    37       37.2813   36.625    37.1563   1994700 ...

```

you must specify the row numbers containing the description and column headers:

```
ibm = ascii2fts('ibm9599.dat', 1, 3);
```

```
ibm =
```

```

desc: International Business Machines Corporation (IBM)
freq: Unknown (0)
'dates: (2)' 'OPEN: (2)' 'HIGH: (2)' 'LOW: (2)' ...
'03-Jan-1995' [ 36.7500] [ 36.9063] [ 36.6563]
'04-Jan-1995' [ 37] [ 37.2813] [ 36.6250]

```

Example 3: If your data file contains rows between the column headers and the data series, for example,

```

Staples, Inc. (SPLS)
Daily prices
DATE      OPEN      HIGH      LOW      CLOSE      VOLUME
Starting date: 04/08/1996
Ending date: 04/07/1999
4/8/96    19.50    19.75    19.25    19.375    548500
4/9/96    19.75    20.125   19.375    20        1135900 ...

```

you need to indicate to `ascii2fts` the rows in the file that must be skipped. Assume that you have called the data file containing the Staples data above `staples.dat`. The command

```
spls = ascii2fts('staples.dat', 1, 3, [4 5]);
```

indicates that the fourth and fifth rows in the file should be skipped in creating the financial time series object:

```
spls =
```

```

desc: Staples, Inc. (SPLS)
freq: Unknown (0)
'dates: (2)' 'OPEN: (2)' 'HIGH: (2)' 'LOW: (2)'
'08-Apr-1996' [ 19.5000] [ 19.7500] [ 19.2500]

```

```
'09-Apr-1996' [ 19.7500] [ 20.1250] [19.3750]
```

Example 4: Create a financial time series object containing time-of-day information.

First create a data file with time information:

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...  
'02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];  
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];  
serial_dates_times = [datenum(dates), datenum(times)];  
data = round(10*rand(6,2));  
stat = fts2ascii('myfts_file2.txt', serial_dates_times, data, ...  
{'dates'; 'times'; 'Data1'; 'Data2'}, 'My FTS with Time');
```

Now read the data file back and create a financial time series object:

```
MyFts = ascii2fts('myfts_file2.txt', 't', 1, 2, 1)
```

```
MyFts =
```

```
desc: My FTS with Time  
freq: Unknown (0)
```

```
 'dates: (6)'   'times: (6)'   'Data1: (6)'   'Data2: (6)'  
'01-Jan-2001' '11:00'      [          9]  [          4]  
'    "    '    '12:00'      [          7]  [          9]  
'02-Jan-2001' '11:00'      [          2]  [          1]  
'    "    '    '12:00'      [          4]  [          4]  
'03-Jan-2001' '11:00'      [          9]  [          8]  
'    "    '    '12:00'      [          9]  [          0]
```

See Also

fints | fts2ascii

bar, barh

Bar chart

Syntax

```
bar(tsobj)
```

```
bar(tsobj, width)
```

```
bar(..., 'style')
```

```
hbar = bar(...)
```

```
barh(...)
```

```
hbarh = barh(...)
```

Arguments

<code>tsobj</code>	Financial time series object.
<code>width</code>	Width of the bars and separation of bars within a group. (Default = 0.8.) If width is 1, the bars within a group touch one another. Values > 1 produce overlapping bars.
<code>style</code>	'grouped' (default) or 'stacked'.

Description

`bar`, `barh` draw vertical and horizontal bar charts.

`bar(tsobj)` draws the columns of data series of the object `tsobj`. The number of data series dictates the number of vertical bars per group. Each group is the data for one particular date.

`bar(tsobj, width)` specifies the width of the bars.

`bar(..., 'style')` changes the style of the bar chart.

`hbar = bar(...)` returns a vector of bar handles.

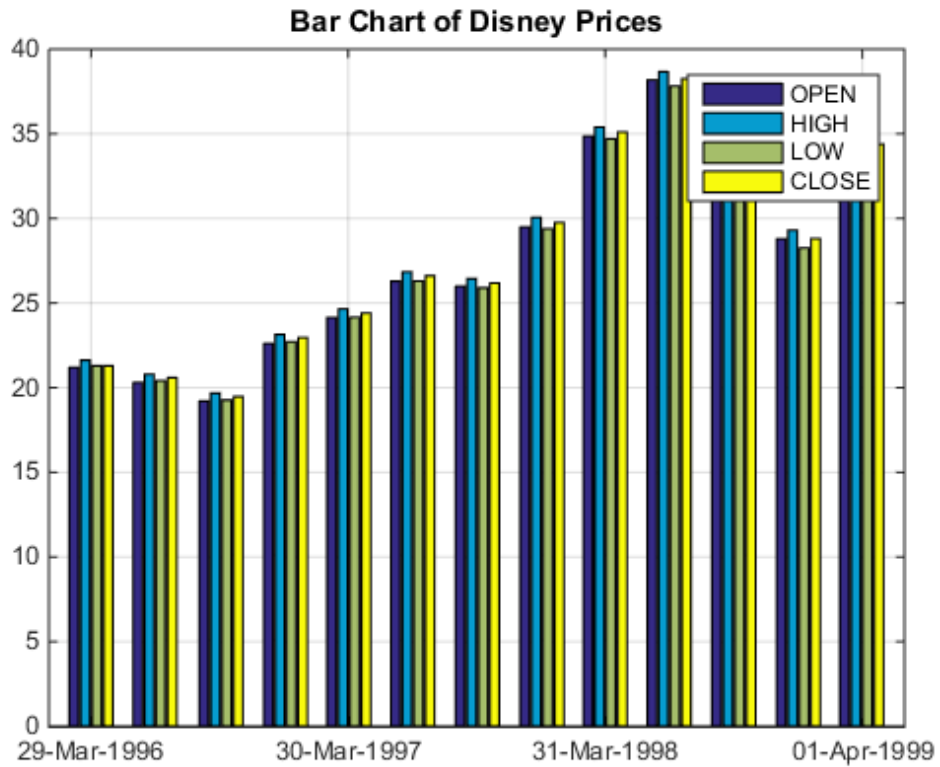
Use the MATLAB command `shading faceted` to put edges on the bars. Use `shading flat` to turn edges off.

Examples

Create a Bar Chart for a Stock

This example shows how to create a bar chart for Disney stock showing high, low, opening, and closing prices.

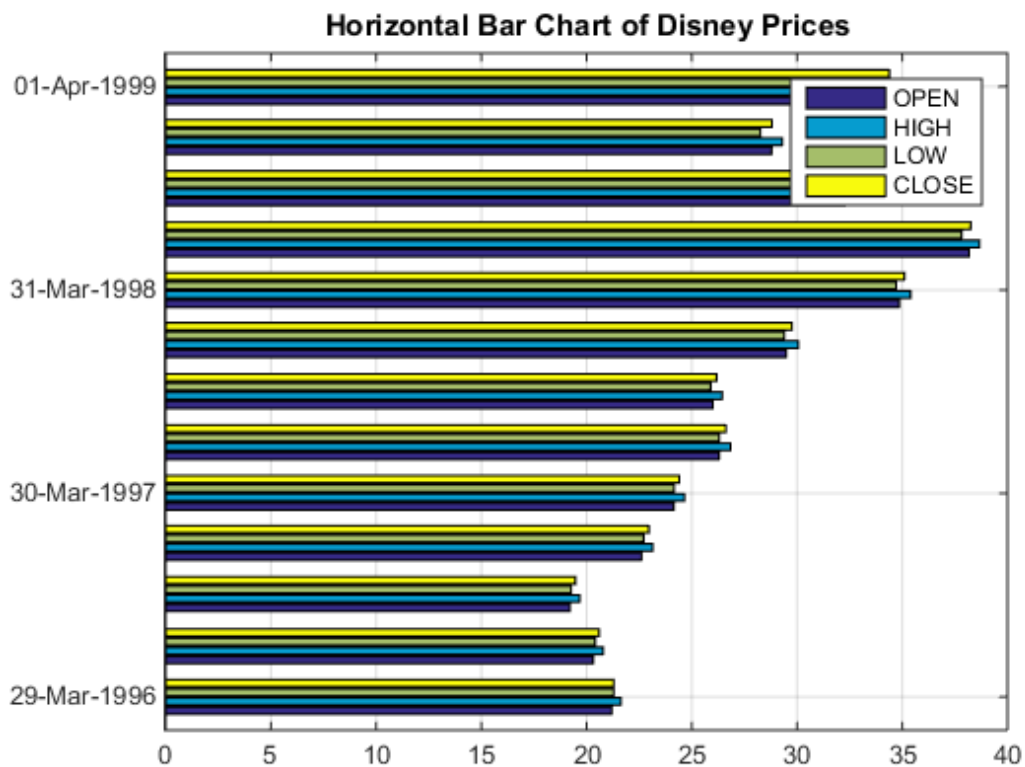
```
load disney
bar(q_dis)
title('Bar Chart of Disney Prices')
```



Create a Horizontal Bar Chart for a Stock

This example shows how to create a horizontal bar chart for Disney stock showing high, low, opening, and closing prices.

```
load disney
barh(q_dis)
title('Horizontal Bar Chart of Disney Prices')
```



See Also

bar3, bar3h | highlow | candle

bar3, bar3h

3-D bar chart

Syntax

```
bar3(tsobj)
```

```
bar3(tsobj, width)
```

```
bar3(..., 'style')
```

```
hbar3 = bar3(...)
```

```
bar3h(...)
```

```
hbar3h = bar3h(...)
```

Arguments

<code>tsobj</code>	Financial time series object.
<code>width</code>	Width of the bars and separation of bars within a group. (Default = 0.8.) If width is 1, the bars within a group touch one another. Values > 1 produce overlapping bars.
<code>style</code>	'detached' (default), 'grouped', or 'stacked'.

Description

`bar3`, `bar3h` draw three-dimensional vertical and horizontal bar charts.

`bar3(tsobj)` draws the columns of data series of the object `tsobj`. The number of data series dictates the number of vertical bars per group. Each group is the data for one particular date.

`bar3(tsobj, width)` specifies the width of the bars.

`bar3(..., 'style')` changes the style of the bar chart.

`hbar3 = bar3(...)` returns a vector of bar handles.

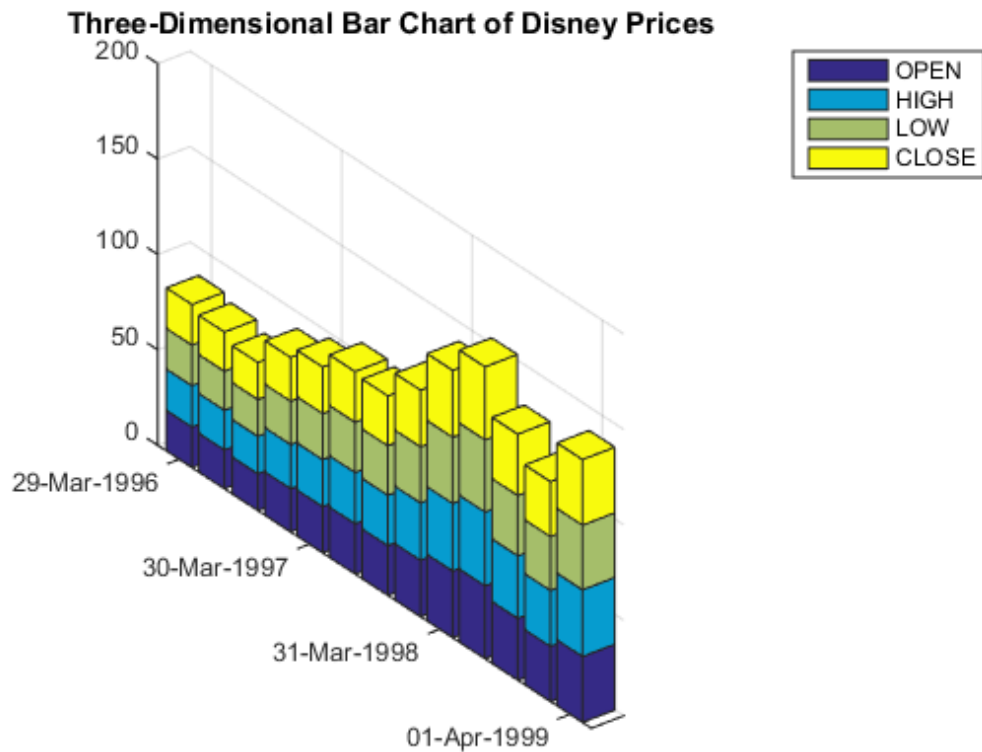
Use the MATLAB command `shading faceted` to put edges on the bars. Use `shading flat` to turn edges off.

Examples

Create a Three-Dimensional Bar Chart

This example shows how to create a three-dimensional bar chart for Disney stock showing high, low, opening, and closing prices.

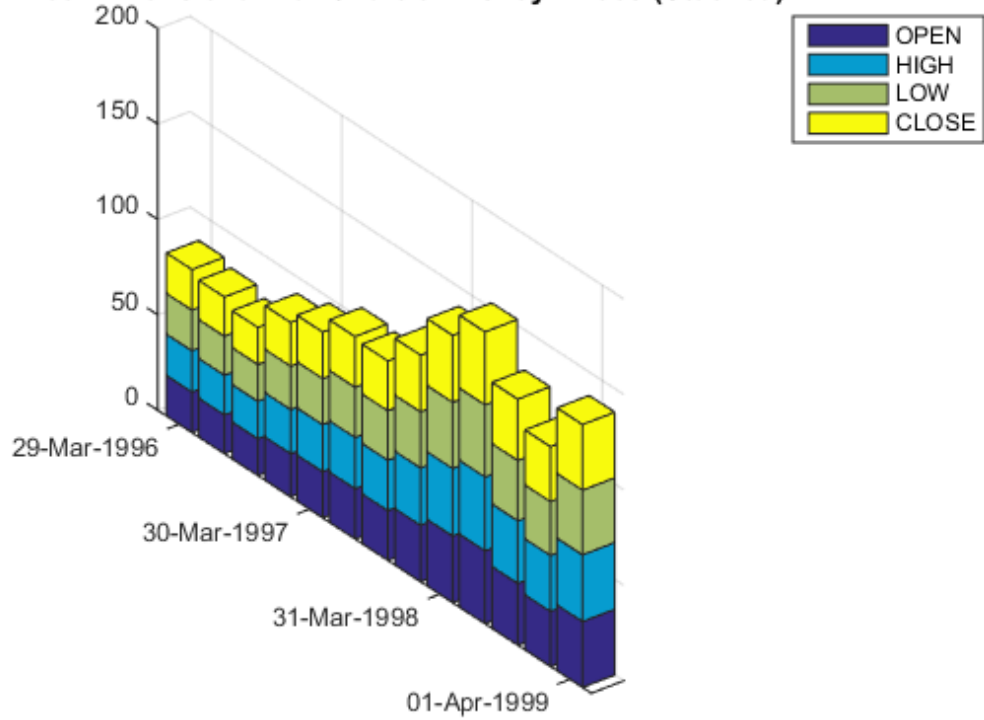
```
load disney
bar3(q_dis, 'stacked')
title('Three-Dimensional Bar Chart of Disney Prices')
```



Create a Three-Dimensional (Stacked) Bar Chart

This example shows how to create a three-dimensional, stacked bar chart for Disney stock showing high, low, opening, and closing prices.

```
load disney
bar3(q_dis, 'stacked')
title('Three-Dimensional Bar Chart of Disney Prices (Stacked)')
```

Three-Dimensional Bar Chart of Disney Prices (Stacked)**See Also**

bar, barh | highlow | candle

beytbill

Bond equivalent yield for Treasury bill

Syntax

```
Yield = beytbill(Settle, Maturity, Discount)
```

Arguments

Settle	Enter as serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Enter as serial date numbers or date strings.
Discount	Discount rate of the Treasury bill. Enter as decimal fraction.

Description

`Yield = beytbill(Settle, Maturity, Discount)` returns the bond equivalent yield for a Treasury bill.

Examples

Find the Bond Equivalent Yield for a Treasury Bill

This example shows how to find the bond equivalent yield for a Treasury bill that has a settlement date of February 11, 2000, a maturity date of August 7, 2000, and a discount rate is 5.77.

```
Yield = beytbill('2/11/2000', '8/7/2000', 0.0577)
```

```
Yield =
```

```
0.0602
```

See Also

`datenum` | `prtbill` | `yldtbill`

binprice

Binomial put and call pricing

Syntax

[AssetPrice, OptionValue] = binprice(Price, Strike, Rate, Time, Increment, Volatility, Flag, DividendRate, Dividend, ExDiv)

Arguments

Price	Underlying asset price. A scalar.
Strike	Option exercise price. A scalar.
Rate	Risk-free interest rate. A scalar. Enter as a decimal fraction.
Time	Option's time until maturity in years. A scalar.
Increment	Time increment. A scalar. Increment is adjusted so that the length of each interval is consistent with the maturity time of the option. (Increment is adjusted so that Time divided by Increment equals an integer number of increments.)
Volatility	Asset's volatility. A scalar.
Flag	Specifies whether the option is a call (Flag = 1) or a put (Flag = 0). A scalar.
DividendRate	(Optional) The dividend rate, as a decimal fraction. A scalar. Default = 0. If you enter a value for DividendRate , set Dividend and ExDiv = 0 or do not enter them. If you enter values for Dividend and ExDiv , set DividendRate = 0.
Dividend	(Optional) The dividend payment at an ex-dividend date, ExDiv . A row vector. For each dividend payment, there must be a corresponding ex-dividend date. Default = 0. If you enter values for Dividend and ExDiv , set DividendRate = 0.
ExDiv	(Optional) Ex-dividend date, specified in number of periods. A row vector. Default = 0.

Description

[AssetPrice, OptionValue] = binprice(Price, Strike, Rate, Time, Increment, Volatility, Flag, DividendRate, Dividend, ExDiv) prices an American option using the Cox-Ross-Rubinstein binomial pricing model.

Examples

Price an American Option Using the Cox-Ross-Rubinstein Binomial Pricing Model

This example shows how to price an American put option with an exercise price of \$50 that matures in 5 months. The current asset price is \$52, the risk-free interest rate is 10%, and the volatility is 40%. There is one dividend payment of \$2.06 in 3-1/2 months. When specifying the input argument `ExDiv` in terms of number of periods, divide the ex-dividend date, specified in years, by the time `Increment`.

$$\text{ExDiv} = (3.5/12) / (1/12) = 3.5$$

$$[\text{Price}, \text{Option}] = \text{binprice}(52, 50, 0.1, 5/12, 1/12, 0.4, 0, 0, 2.06, 3.5)$$

Price =

52.0000	58.1367	65.0226	72.7494	79.3515	89.0642
0	46.5642	52.0336	58.1706	62.9882	70.6980
0	0	41.7231	46.5981	49.9992	56.1192
0	0	0	37.4120	39.6887	44.5467
0	0	0	0	31.5044	35.3606
0	0	0	0	0	28.0688

Option =

4.4404	2.1627	0.6361	0	0	0
0	6.8611	3.7715	1.3018	0	0
0	0	10.1591	6.3785	2.6645	0
0	0	0	14.2245	10.3113	5.4533
0	0	0	0	18.4956	14.6394
0	0	0	0	0	21.9312

The output returned is the asset price and American option value at each node of the binary tree.

References

Cox, J., S. Ross, and M. Rubenstein, "Option Pricing: A Simplified Approach", *Journal of Financial Economics* 7, Sept. 1979, pp. 229-263.

Hull, John C., *Options, Futures, and Other Derivative Securities*, 2nd edition, Chapter 14.

See Also

blkprice | blsprice

blkimpv

Implied volatility for futures options from Black model

Syntax

```
Volatility = blkimpv(Price, Strike, Rate, Time, Value, Limit,
Tolerance, Class)
```

Arguments

Price	Current price of the underlying asset (a futures contract).
Strike	Exercise price of the futures option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Value	Price of a European futures option from which the implied volatility of the underlying asset is derived.
Limit	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. If <code>Limit</code> is empty or unspecified, the default = 10, or 1000% per annum.
Tolerance	(Optional) Implied volatility termination tolerance. A positive scalar. Default = 1e-6.
Class	(Optional) Option class (call or put) indicating the option type from which the implied volatility is derived. May be either a logical indicator or a cell array of characters. To specify call options, set <code>Class = true</code> or <code>Class = {'call'}</code> ; to specify put options, set <code>Class = false</code> or <code>Class = {'put'}</code> . If <code>Class</code> is empty or unspecified, the default is a call option.

Description

`Volatility = blkimpv(Price, Strike, Rate, Time, CallPrice, MaxIterations, Tolerance)` computes the implied volatility of a futures price from the market value of European futures options using Black's model.

`Volatility` is the implied volatility of the underlying asset derived from European futures option prices, expressed as a decimal number. If no solution is found, `blkimpv` returns NaN.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to compute the implied volatility of all the options. If more than one input is a vector or matrix, the dimensions of all nonscalar inputs must be identical.

Rate and Time must be expressed in consistent units of time.

Examples

Find Implied Volatility for Futures Options from Black's Model

This example shows how to find the implied volatility for a European call futures option that expires in four months, trades at \$1.1166, and has an exercise price of \$20. Assume that the current underlying futures price is also \$20 and that the risk-free rate is 9% per annum. Furthermore, assume that you are interested in implied volatilities no greater than 0.5 (50% per annum). Under these conditions, the following commands all return an implied volatility of 0.25, or 25% per annum.

```
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5);  
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5, [], {'Call'});  
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5, [], true)
```

```
Volatility =  
    0.2500
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003, pp. 287-288.

Black, Fischer, “The Pricing of Commodity Contracts,” *Journal of Financial Economics*, March 3, 1976, pp. 167-79.

See Also

blkprice | blsimpv | blsprice

blkprice

Black model for pricing futures options

Syntax

```
[Call, Put] = blkprice(Price, Strike, Rate, Time, Volatility)
```

Arguments

Price	Current price of the underlying asset (a futures contract).
Strike	Strike or exercise price of the futures option.
Rate	Annualized, continuously compounded, risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time until expiration of the option, expressed in years. Must be greater than 0.
Volatility	Annualized futures price volatility, expressed as a positive decimal number.

Description

`[Call, Put] = blkprice(ForwardPrice, Strike, Rate, Time, Volatility)` uses Black's model to compute European put and call futures option prices.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to compute the implied volatility from all options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

Rate, Time, and Volatility must be expressed in consistent units of time.

Examples

Compute European Put and Call Futures Option Prices Using Black's Model

This example shows how to price European futures options with exercise prices of \$20 that expire in four months. Assume that the current underlying futures price is also \$20 with a volatility of 25% per annum. The risk-free rate is 9% per annum.

```
[Call, Put] = blkprice(20, 20, 0.09, 4/12, 0.25)
```

```
Call =
```

```
1.1166
```

```
Put =
```

```
1.1166
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003, pp. 287-288.

Black, Fischer, "The Pricing of Commodity Contracts," *Journal of Financial Economics*, March 3, 1976, pp. 167-179.

See Also

binprice | blsprice

blsdelta

Black-Scholes sensitivity to underlying price change

Syntax

```
[CallDelta, PutDelta] = blsdelta(Price, Strike, Rate, Time, Volatility, Yield)
```

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, Yield could represent the dividend yield. For currency options, Yield could be the foreign risk-free interest rate.

Description

[CallDelta, PutDelta] = blsdelta(Price, Strike, Rate, Time, Volatility, Yield) returns delta, the sensitivity in option value to change in the underlying asset price. Delta is also known as the hedge ratio. blsdelta uses normcdf, the normal cumulative distribution function in the Statistics Toolbox.

Note: blsdelta can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument Yield as:

`Yield = Rate`

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

`Yield = ForeignRate`

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Find the Sensitivity in Option Value to Change in the Underlying Asset Price

This example shows how to find the Black-Scholes delta sensitivity for an underlying asset price change.

```
[CallDelta, PutDelta] = blsdelta(50, 50, 0.1, 0.25, 0.3, 0)
```

```
CallDelta =
```

```
    0.5955
```

```
PutDelta =
```

```
   -0.4045
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

See Also

`blsgamma` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

blsgamma

Black-Scholes sensitivity to underlying delta change

Syntax

Gamma = blsgamma(Price, Strike, Rate, Time, Volatility, Yield)

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, Yield could represent the dividend yield. For currency options, Yield could be the foreign risk-free interest rate.

Description

Gamma = blsgamma(Price, Strike, Rate, Time, Volatility, Yield) returns gamma, the sensitivity of delta to change in the underlying asset price. blsgamma uses normpdf, the probability density function in the Statistics Toolbox.

Note: blsgamma can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument **Yield** as:

Yield = Rate

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

Yield = ForeignRate

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Find Gamma for a Change in the Underlying Asset Price

This example shows how to find the gamma, the sensitivity of delta to a change in the underlying asset price.

```
Gamma = blsgamma(50, 50, 0.12, 0.25, 0.3, 0)
```

```
Gamma =
```

```
0.0512
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

See Also

`blsdelta` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

blsimpv

Black-Scholes implied volatility

Syntax

Volatility = blsimpv(Price, Strike, Rate, Time, Value, Limit, Yield, Tolerance, Class)

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Value	Price of a European option from which the implied volatility of the underlying asset is derived.
Limit	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. If Limit is empty or unspecified, the default = 10, or 1000% per annum.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, Yield could represent the dividend yield. For currency options, Yield could be the foreign risk-free interest rate.
Tolerance	(Optional) Implied volatility termination tolerance. A positive scalar. Default = 1e-6.
Class	(Optional) Option class (call or put) indicating the option type from which the implied volatility is derived. May be either a logical indicator or a cell array of characters. To specify call options, set Class = true or Class = {'call'} ; to specify put options, set Class = false or Class = {'put'} . If Class is empty or unspecified, the default is a call option.

Description

`Volatility = blsimpv(Price, Strike, Rate, Time, Value, Limit, Yield, Tolerance, Class)` using a Black-Scholes model computes the implied volatility of an underlying asset from the market value of European call and put options.

`Volatility` is the implied volatility of the underlying asset derived from European option prices, expressed as a decimal number. If no solution is found, `blsimpv` returns `NaN`.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to price all the options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

`Rate`, `Time`, and `Yield` must be expressed in consistent units of time.

Examples

Compute the Implied Volatility of an Underlying Asset Using a Black-Scholes Model

This example shows how to compute the implied volatility for a European call option trading at \$10 with an exercise price of \$95 and three months until expiration. Assume that the underlying stock pays no dividend and trades at \$100. The risk-free rate is 7.5% per annum. Furthermore, assume that you are interested in implied volatilities no greater than 0.5 (50% per annum). Under these conditions, the following statements all compute an implied volatility of 0.3130, or 31.30% per annum.

```
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5);  
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5, 0, [], {'Call'});  
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5, 0, [], true)
```

```
Volatility =  
  
    0.3130
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

Luenberger, David G., *Investment Science*, Oxford University Press, 1998.

See Also

blsdelta | blsgamma | blslambda | blsprice | blsrho | blstheta

blambda

Black-Scholes elasticity

Syntax

```
[CallEl, PutEl] = blambda(Price, Strike, Rate, Time, Volatility, Yield)
```

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, Yield could represent the dividend yield. For currency options, Yield could be the foreign risk-free interest rate.

Description

[CallEl, PutEl] = blambda(Price, Strike, Rate, Time, Volatility, Yield) returns the elasticity of an option. CallEl is the call option elasticity or leverage factor, and PutEl is the put option elasticity or leverage factor. Elasticity (the leverage of an option position) measures the percent change in an option price per one percent change in the underlying asset price. blambda uses normcdf, the normal cumulative distribution function in the Statistics Toolbox.

Note: blslambda can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

`Yield = Rate`

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

`Yield = ForeignRate`

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Find the Black-Scholes Elasticity (Lambda) for an Option

This example shows how to find the Black-Scholes elasticity, or leverage, of an option position.

```
[CallEl, PutEl] = blslambda(50, 50, 0.12, 0.25, 0.3)
```

```
CallEl =
```

```
8.1274
```

```
PutEl =
```

```
-8.6466
```

References

Daigler, *Advanced Options Trading*, Chapter 4.

See Also

[blsdelta](#) | [blsgamma](#) | [blsprice](#) | [blsrho](#) | [blstheta](#) | [blsvega](#)

blsprice

Black-Scholes put and call option pricing

Syntax

[Call, Put] = blsprice(Price, Strike, Rate, Time, Volatility, Yield)

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, Yield could represent the dividend yield. For currency options, Yield could be the foreign risk-free interest rate.

Description

[Call, Put] = blsprice(Price, Strike, Rate, Time, Volatility, Yield) computes European put and call option prices using a Black-Scholes model.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to price all the options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

Rate, Time, Volatility, and Yield must be expressed in consistent units of time.

Note: blsprice can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

`Yield = Rate`

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

`Yield = ForeignRate`

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Compute European Put and Call Option Prices Using a Black-Scholes Model

This example shows how to price European stock options that expire in three months with an exercise price of \$95. Assume that the underlying stock pays no dividend, trades at \$100, and has a volatility of 50% per annum. The risk-free rate is 10% per annum.

```
[Call, Put] = blsprice(100, 95, 0.1, 0.25, 0.5)
```

```
Call =
```

```
    13.6953
```

```
Put =
```

```
    6.3497
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

Luenberger, David G., *Investment Science*, Oxford University Press, 1998.

See Also

blkprice | blsdelta | blsgamma | blsimpv | blslambda | blsrho | blstheta | blsvega

blsrho

Black-Scholes sensitivity to interest rate change

Syntax

```
[CallRho, PutRho]= blsrho(Price, Strike, Rate, Time, Volatility, Yield)
```

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, <code>Yield</code> could represent the dividend yield. For currency options, <code>Yield</code> could be the foreign risk-free interest rate.

Description

`[CallRho, PutRho]= blsrho(Price, Strike, Rate, Time, Volatility, Yield)` returns the call option rho `CallRho`, and the put option rho `PutRho`. Rho is the rate of change in value of derivative securities with respect to interest rates. `blsrho` uses `normcdf`, the normal cumulative distribution function in the Statistics Toolbox

Note: `blsrho` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

Yield = Rate

When pricing currencies (Garman-Kohlhagen model), enter the input argument Yield as:

Yield = ForeignRate

where ForeignRate is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Find the Black-Scholes Sensitivity (Rho) to Interest-Rate Change

This example shows how to find the Black-Scholes sensitivity, rho, to interest-rate change.

```
[CallRho, PutRho] = blsrho(50, 50, 0.12, 0.25, 0.3, 0)
```

```
CallRho =
```

```
6.6686
```

```
PutRho =
```

```
-5.4619
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

See Also

blsdelta | blsgamma | blslambda | blsprice | blstheta | blsvega

blstheta

Black-Scholes sensitivity to time-until-maturity change

Syntax

```
[CallTheta, PutTheta] = blstheta(Price, Strike, Rate, Time,
Volatility, Yield)
```

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, Yield could represent the dividend yield. For currency options, Yield could be the foreign risk-free interest rate.

Description

[CallTheta, PutTheta] = blstheta(Price, Strike, Rate, Time, Volatility, Yield) returns the call option theta CallTheta, and the put option theta PutTheta.

Theta is the sensitivity in option value with respect to time and is measured in years. CallTheta or PutTheta can be divided by 365 to get Theta per calendar day or by 252 to get Theta by trading day.

blstheta uses normcdf, the normal cumulative distribution function in the Statistics Toolbox.

Note: blstheta can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

`Yield = Rate`

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

`Yield = ForeignRate`

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Compute the Black-Scholes Sensitivity to Time-Until-Maturity Change (Theta)

This example shows how to compute theta, the sensitivity in option value with respect to time.

```
[CallTheta, PutTheta] = blstheta(50, 50, 0.12, 0.25, 0.3, 0)
```

```
CallTheta =
```

```
    -8.9630
```

```
PutTheta =
```

```
    -3.1404
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

See Also

blsdelta | blsgamma | blslambda | blsprice | blsrho | blsvega

blsvega

Black-Scholes sensitivity to underlying price volatility

Syntax

```
Vega = blsvega(Price, Strike, Rate, Time, Volatility, Yield)
```

Arguments

Price	Current price of the underlying asset.
Strike	Exercise price of the option.
Rate	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
Time	Time to expiration of the option, expressed in years.
Volatility	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
Yield	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, <code>Yield</code> could represent the dividend yield. For currency options, <code>Yield</code> could be the foreign risk-free interest rate.

Description

`Vega = blsvega(Price, Strike, Rate, Time, Volatility, Yield)` returns `Vega`, the rate of change of the option value with respect to the volatility of the underlying asset. `blsvega` uses `normpdf`, the normal probability density function in the Statistics Toolbox.

Note: `blsvega` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

Yield = Rate

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

Yield = ForeignRate

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Compute Black-Scholes Sensitivity to Underlying Price Volatility (Vega)

This example shows how to compute vega, the rate of change of the option value with respect to the volatility of the underlying asset.

```
Vega = blsvega(50, 50, 0.12, 0.25, 0.3, 0)
```

```
Vega =
```

```
9.6035
```

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

See Also

`blsdelta` | `blsgamma` | `blslambda` | `blsprice` | `blsrho` | `blstheta`

bm

Brownian motion models

Syntax

```
BM = bm(Mu, Sigma)
```

```
BM = bm(Mu, Sigma, 'Name1', Value1, 'Name2', Value2, ...)
```

Class

BM

Description

This constructor creates and displays 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. 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:

$$dX_t = \mu(t)dt + V(t)dW_t$$

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

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Mu	<p>Mu represents μ. 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). As a deterministic function of time, when Mu is called with a real-valued scalar time \mathbf{t} as its only input, Mu must produce an NVARs-by-NVARs matrix. If you specify Mu as a function of time and state, it calculates the expected instantaneous rate of drift. This function must generate an NVARs-by-1 column vector when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V. 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. As a deterministic function of time, when Sigma is called with a real-valued scalar time \mathbf{t} as its only input,</p>

Sigma must produce an NVARs-by-NBROWNS matrix. If you specify **Sigma** as a function of time and state, it must generate an NVARs-by-NBROWNS matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t .
- 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

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, bm applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, bm applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, bm applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>

Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>
Simulation	<p>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).</p>

Output Arguments

BM	<p>Object of class BM with the following displayed parameters:</p> <ul style="list-style-type: none"> • 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
----	--

Algorithm

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 17-24

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | diffusion | sdeld

bndconvp

Bond convexity given price

Syntax

```
[YearConvexity, PerConvexity] = bndconvp(Price, CouponRate,
Settle, Maturity)
[YearConvexity, PerConvexity] = bndconvp(Price,
CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,
IssueDate, FirstCouponDate, LastCouponDate, StartDate,
Face)
[YearConvexity, PerConvexity] = bndconvp(Price,
CouponRate, Settle, Maturity,
'ParameterName', 'ParameterValue ...)
```

Description

[YearConvexity, PerConvexity] = bndconvp(Price, CouponRate, Settle, Maturity) computes the convexity of NUMBONDS fixed income securities given a clean price for each bond.

[YearConvexity, PerConvexity] = bndconvp(Price, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) computes the convexity of NUMBONDS fixed income securities given a clean price for each bond using optional arguments.

[YearConvexity, PerConvexity] = bndconvp(Price, CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...) computes the convexity of NUMBONDS fixed income securities given a clean price for each bond and accepts optional inputs as one or more comma-separated parameter/value pairs. *ParameterName* ' is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Price

Clean price (excludes accrued interest).

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified

LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Face

Face or par value.

Default: 100

Parameter-Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Output Arguments

YearConvexity

NUMBONDS-by-1 vector for the yearly (annualized) convexity.

PerConvexity

NUMBONDS-by-1 vector for the periodic convexity reported on a semiannual bond basis (in accordance with SIA convention).

Definitions

`bndconvp` determines the convexity for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). This function also determines the convexity of a zero coupon bond.

All specified arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalar arguments. Use an empty matrix ([]) as a placeholder for an optional argument. Fill in unspecified entries input vectors with NaNs. Dates can be serial date numbers or date strings.

Examples

Find Bond Convexity Given Price

This example shows how to compute the convexity of three bonds given their prices.

```
Price = [106; 100; 98];  
CouponRate = 0.055;  
Settle = '02-Aug-1999';  
Maturity = '15-Jun-2004';  
Period = 2;  
Basis = 0;  
  
[YearConvexity, PerConvexity] = bndconvp(Price,...  
CouponRate,Settle, Maturity, Period, Basis)
```

```
YearConvexity =
```

```
    21.4447  
    21.0363  
    20.8951
```

PerConvexity =

85.7788

84.1454

83.5803

More About

- “Yield Conventions”

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

cfconv | bndconvy | bnddurp | bnddury | cfdur

bndconvy

Bond convexity given yield

Syntax

```
[YearConvexity, PerConvexity] = bndconvy(Yield, CouponRate,
Settle, Maturity)
[YearConvexity, PerConvexity] = bndconvy(Yield,
CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,
IssueDate, FirstCouponDate, LastCouponDate, StartDate,
Face)
[YearConvexity, PerConvexity] = bndconvy(Yield,
CouponRate, Settle, Maturity,
'ParameterName', 'ParameterValue ...)
```

Description

[YearConvexity, PerConvexity] = bndconvy(Yield, CouponRate, Settle, Maturity) computes the convexity of NUMBONDS fixed income securities given the yield to maturity for each bond.

[YearConvexity, PerConvexity] = bndconvy(Yield, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) computes the convexity of NUMBONDS fixed income securities given the yield to maturity for each bond using optional arguments.

[YearConvexity, PerConvexity] = bndconvy(Yield, CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...) computes the convexity of NUMBONDS fixed income securities given the yield to maturity for each bond and accepts optional inputs as one or more comma-separated parameter/value pairs. 'ParameterName' is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Yield

Yield to maturity on a semiannual basis.

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified

LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Face

Face or par value.

Default: 100

Parameter-Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Output Arguments

YearConvexity

NUMBONDS-by-1 vector for the yearly (annualized) convexity.

PerConvexity

NUMBONDS-by-1 vector for the periodic convexity reported on a semiannual bond basis (in accordance with SIA convention).

Definitions

bndconvy determines the convexity for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). This function also determines the convexity of a zero coupon bond.

All specified arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalar arguments. Use an empty matrix ([]) as a placeholder for an optional argument. Fill in unspecified entries input vectors with NaNs. Dates can be serial date numbers or date strings.

Examples

Find Bond Convexity Given Yield

This example shows how to compute the convexity of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];  
CouponRate = 0.055;  
Settle = '02-Aug-1999';  
Maturity = '15-Jun-2004';  
Period = 2;  
Basis = 0;
```

```
[YearConvexity, PerConvexity]=bndconvy(Yield, CouponRate,...  
Settle, Maturity, Period, Basis)
```

```
YearConvexity =  
  
    21.4825  
    21.0358  
    20.8885
```


PerConvexity =

85.9298

84.1434

83.5541

More About

- “Yield Conventions”

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

cfconv | bndconvp | bnddurp | bnddury | cfdur

bnddurp

Bond duration given price

Syntax

```
[ModDuration, YearDuration, PerDuration] = bnddurp(Price,  
CouponRate, Settle, Maturity)  
[ModDuration, YearDuration, PerDuration] = bnddurp(Price,  
CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,  
IssueDate, FirstCouponDate, LastCouponDate, StartDate,  
Face)  
[ModDuration, YearDuration, PerDuration] = bnddurp(Price,  
CouponRate, Settle, Maturity,  
'ParameterName', 'ParameterValue ...)
```

Description

[ModDuration, YearDuration, PerDuration] = bnddurp(Price, CouponRate, Settle, Maturity) computes the Macaulay and modified duration of NUMBONDS fixed-income securities given a clean price for each bond.

[ModDuration, YearDuration, PerDuration] = bnddurp(Price, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) computes the Macaulay and modified duration of NUMBONDS fixed-income securities given a clean price for each bond using optional arguments.

[ModDuration, YearDuration, PerDuration] = bnddurp(Price, CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...) computes the Macaulay and modified duration of NUMBONDS fixed-income securities given a clean price for each bond and accepts optional inputs as one or more comma-separated parameter/value pairs. 'ParameterName' is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Price

Clean price (excludes accrued interest).

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified

LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Face

Face or par value.

Default: 100

Parameter-Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Output Arguments

ModDuration

NUMBONDS-by-1 vector for the modified duration in years, reported on a semiannual bond basis (in accordance with SIA convention).

YearDuration

NUMBONDS-by-1 vector for the Macaulay duration in years.

PerDuration

NUMBONDS-by-1 vector for the periodic Macaulay duration reported on a semiannual bond basis (in accordance with SIA convention).

Definitions

`bnddurp` determines the Macaulay and modified duration for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). This function also determines the Macaulay and modified duration for a zero coupon bond.

All specified arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalar arguments. Use an empty matrix (`[]`) as a placeholder for an optional argument. Fill in unspecified entries input vectors with NaNs. Dates can be serial date numbers or date strings.

Examples

Find Bond Duration Given Price

This example shows how to compute the duration of three bonds given their prices.

```
Price = [106; 100; 98];  
CouponRate = 0.055;  
Settle = '02-Aug-1999';  
Maturity = '15-Jun-2004';  
Period = 2;  
Basis = 0;
```

```
[ModDuration, YearDuration, PerDuration] = bnddurp(Price,...  
CouponRate, Settle, Maturity, Period, Basis)
```

```
ModDuration =
```

4.2400
4.1925
4.1759

YearDuration =

4.3275
4.3077
4.3007

PerDuration =

8.6549
8.6154
8.6014

More About

- “Yield Conventions”

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

bndconvy | bndconvp | bnddury | bndkrdur

bnddury

Bond duration given yield

Syntax

```
[ModDuration, YearDuration, PerDuration] = bnddury(Yield,
CouponRate, Settle, Maturity)
[ModDuration, YearDuration, PerDuration] = bnddury(Yield,
CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,
IssueDate, FirstCouponDate, LastCouponDate, StartDate,
Face)
[ModDuration, YearDuration, PerDuration] = bnddury(Yield,
CouponRate, Settle, Maturity,
'ParameterName', 'ParameterValue ...)
```

Description

[ModDuration, YearDuration, PerDuration] = bnddury(Yield, CouponRate, Settle, Maturity) computes the Macaulay and modified duration of NUMBONDS fixed income securities given yield to maturity for each bond.

[ModDuration, YearDuration, PerDuration] = bnddury(Yield, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) computes the Macaulay and modified duration of NUMBONDS fixed income securities given yield to maturity for each bond using optional arguments.

[ModDuration, YearDuration, PerDuration] = bnddury(Yield, CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...) computes the Macaulay and modified duration of NUMBONDS fixed income securities given yield to maturity for each bond and accepts optional inputs as one or more comma-separated parameter/value pairs. 'ParameterName' is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Yield

Yield to maturity on a semiannual basis.

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified

LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Face

(Optional) Face or par value.

Default: 100

Parameter-Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Output Arguments

ModDuration

NUMBONDS-by-1 vector for the modified duration in years, reported on a semiannual bond basis (in accordance with SIA convention).

YearDuration

NUMBONDS-by-1 vector for the Macaulay duration in years.

PerDuration

NUMBONDS-by-1 vector for the periodic Macaulay duration reported on a semiannual bond basis (in accordance with SIA convention).

Definitions

`bnddurp` determines the duration for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). This function also determines the Macaulay and modified duration for a zero coupon bond.

All specified arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalar arguments. Use an empty matrix ([]) as a placeholder for an optional argument. Fill in unspecified entries input vectors with NaNs. Dates can be serial date numbers or date strings.

Examples**Find Bond Duration Given Yield**

This example shows how to compute the duration of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];  
CouponRate = 0.055;  
Settle = '02-Aug-1999';  
Maturity = '15-Jun-2004';  
Period = 2;  
Basis = 0;
```

```
[ModDuration,YearDuration,PerDuration]=bnddury(Yield,...  
CouponRate, Settle, Maturity, Period, Basis)
```

```
ModDuration =
```

4.2444
4.1924
4.1751

YearDuration =

4.3292
4.3077
4.3004

PerDuration =

8.6585
8.6154
8.6007

More About

- “Yield Conventions”

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

bndconvp | bndconvy | bnddurp | bndkrdur

bndkrdur

Bond key rate duration given zero curve

Syntax

```
KRDUR = bndkrdur(ZeroData, CouponRate,
Settle, Maturity)
```

```
KRDUR = bndkrdur(ZeroData, CouponRate, Settle,
Maturity, 'Parameter1', Value1, 'Parameter2',
Value2, ...)
```

Arguments

ZeroData	Zero curve represented as a <code>numRates</code> -by-2 matrix where the first column is a MATLAB date number and the second column is accompanying zero rates.
CouponRate	<code>numBonds</code> -by-1 vector of coupon rates in decimal form.
Settle	Scalar MATLAB date number for the settlement date for all the bonds and the zero data. Settle must be the same settlement date for all the bonds and the zero curve.
Maturity	<code>numBonds</code> -by-1 vector of maturity dates.
Period	(Optional) Coupons per year of the bond. A vector of integers. Acceptable values are 0, 1, 2 (default), 3, 4, 6, and 12.
InterpMethod	(Optional) Interpolation method used to obtain points from the zero curve. Acceptable values are: <ul style="list-style-type: none"> • 'linear' (default) • 'cubic' • 'pchip'
ShiftValue	(Optional) Scalar value that zero curve is shifted up and down to compute duration. Default is .01 (100 basis points).

KeyRates	(Optional) Rates to perform the duration calculation, specified as a time to maturity. By default, KeyRates is set to each of the zero dates.
CurveCompounding	(Optional) Compounding frequency of the curve. Default is semiannual.
CurveBasis	(Optional) Basis of the curve, where the choices are identical to Basis below. Default is 0 (actual/actual).
Basis	<p>(Optional) Day-count basis of the bond instrument. A vector of integers:</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	<p>(Optional) End-of-month rule. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. The values are:</p> <ul style="list-style-type: none"> • 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. • 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.
StartDate	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify StartDate , the effective start date is the Settle date.
Face	(Optional) Face or par value. Default = 100. Face has no impact on key rate duration.

Note: You must enter the optional arguments as parameter/value pairs.

Description

```
KRDUR = bndkrdur(ZeroData, CouponRate, Settle, Maturity)
```

```
KRDUR = bndkrdur(ZeroData, CouponRate, Settle, Maturity,
'Parameter1', Value1, 'Parameter2', Value2, ...)
```

The output argument **KRDUR** is a **numBonds-by-numRates** matrix of key rate durations.

bndkrdur computes the key rate durations for one or more bonds given a zero curve and a set of key rates. By default, the key rates are each of the zero curve rates. For each key rate, the duration is computed by shifting the zero curve up and down by a specified amount (`ShiftValue`) at that particular key rate, computing the present value of the bond in each case with the new zero curves, and then evaluating the following:

$$krdur_i = \frac{(PV_{down} - PV_{up})}{(PV \times ShiftValue \times 2)}$$

Note: The shift to the curve is computed by shifting the particular key rate by the `ShiftValue` and then interpolating the values of the curve in the interval between the previous and next key rates. For the first key rate, any curve values before the date are equal to the `ShiftValue`; likewise, for the last key rate, any curve values after the date are equal to the `ShiftValue`.

Examples

Find the Bond Key Rate Duration Given the Zero Curve

This example shows how to compute the key rate duration of a bond for key rate times of 2, 5, 10, and 30 years.

```
ZeroRates = [0.0476 .0466 .0465 .0468 .0473 .0478 ...
             .0493 .0539 .0572 .0553 .0530]';

ZeroDates = daysadd('31-Dec-1998',[30 360 360*2 360*3 360*5 ...
                               360*7 360*10 360*15 360*20 360*25 360*30],1);

ZeroData = [ZeroDates ZeroRates];

krdur = bndkrdur(ZeroData,.0525,'12/31/1998',...
               '11/15/2028','KeyRates',[2 5 10 30])

krdur =

    0.2986    0.8791    4.1353    9.5814
```

References

Golub, B.W. and L.M. Tilman, *Risk Management: Approaches for Fixed Income Markets* Wiley, 2000.

Tuckman, B. *Fixed Income Securities: Tools for Today's Markets* Wiley, 2002.

See Also

bndconvp | bndconvy | bnddurp | bnddury

bndprice

Price fixed-income security from yield to maturity

Syntax

```
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,
Maturity)
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,
Maturity, Period, Basis, EndMonthRule, IssueDate,
FirstCouponDate, LastCouponDate, StartDate, Face)
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,
Maturity, 'ParameterName', 'ParameterValue ...')
```

Description

[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, Maturity) given bonds with SIA date parameters and semiannual yields to maturity, returns the clean prices and accrued interest due.

[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) given bonds with SIA date parameters and semiannual yields to maturity and optional inputs, returns the clean prices and accrued interest due.

[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...') accepts optional inputs as one or more comma-separated parameter/value pairs. 'ParameterName' is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Yield

Bond yield to maturity is on a semiannual basis for `basis` values 0 through 7 and an annual basis for `basis` values 8 through 12.

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.

Settle

Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter value pairs. You cannot mix ordered syntax with parameter value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

Default: If you do not specify an **IssueDate**, the cash flow payment dates are determined from other inputs.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.

Default: If you do not specify `StartDate`, the effective start date is the `Settle` date.

Face

Face or par value.

Default: 100

Parameter–Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

Default: SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Default: SIA bases use the `actual/actual` day count to compute discount factors.

LastCouponInterest

Compounding convention for computing the yield of a bond in the last coupon period. This is based on only the last coupon and the face value to be repaid. Acceptable values are `simple` or `compound`.

Default: `compound`

Output Arguments

Price

NUMBONDS-by-1 vector for the clean price of the bond. The dirty price of the bond is the clean price plus the accrued interest. It equals the present value of the bond cash flows of the yield to maturity with semiannual compounding.

AccruedInt

NUMBONDS-by-1 vector for the accrued interest payable at settlement.

Definitions

Given NBONDS with date parameters and yields to maturity, `bndprice` returns the clean prices and the accrued interest due.

All nonscalar or empty matrix input arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors. Fill in unspecified entries input vectors with NaNs. Dates can be serial date numbers or date strings.

Examples

Price a Treasury Bond from Yield to Maturity

This example shows how to price a treasury bond at three different yield values.

```
Yield = [0.04; 0.05; 0.06];  
CouponRate = 0.05;  
Settle = '20-Jan-1997';
```

```
Maturity = '15-Jun-2002';  
Period = 2;  
Basis = 0;  
  
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,...  
Maturity, Period, Basis)
```

```
Price =  
  
    104.8106  
     99.9951  
     95.4384
```

```
AccruedInt =  
  
    0.4945  
    0.4945  
    0.4945
```

Price a Treasury Bond with Different Yield Values

This example shows how to price a Treasury bond at two different yield values that include parameter/value pairs for `CompoundingFrequency`, `DiscountBasis`, and `LastCouponPeriodInterest`.

```
bndprice(.04,0.08,'5/25/2004','4/21/2005','Period',1,'Basis',8, ...  
'LastCouponInterest','simple')
```

```
ans =  
  
    103.4743
```

- “Pricing Functions”

More About

Algorithms

For SIA conventions, the following formula defines bond price and yield:

$$PV = \sum_{i=1}^n \left(\frac{CF}{\left(1 + \frac{z}{f}\right)^{TF}} \right)$$

where:

$PV =$	Present value of a cash flow.
$CF =$	Cash flow amount.
$z =$	Risk-adjusted annualized rate or yield corresponding to a given cash flow. The yield is quoted on a semiannual basis.
$f =$	Frequency of quotes for the yield. Default is 2 for Basis values 0 to 7 and 13 and 1 for Basis values 8 to 12. The default can be overridden by specifying the CompoundingFrequency name/value pair.
$TF =$	Time factor for a given cash flow. The time factor is computed using the compounding frequency and the discount basis. If these values are not specified, then the defaults are as follows: CompoundingFrequency default is 2 for Basis values 0 to 7 and 13 and 1 for Basis values 8 to 12. DiscountBasis is 0 for Basis values 0 to 7 and 13 and the input Basis for Basis values 8 to 12.

Note: The **Basis** is always used to compute accrued interest.

For ISMA conventions, the frequency of annual coupon payments determines bond price and yield.

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, "Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures", *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

`bndyield` | `cfamounts`

bndspread

Static spread over spot curve

Syntax

```
Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity)
Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity,
Period, Basis, EndMonthRule, IssueDate, FirstCouponDate,
LastCouponDate, StartDate, Face)
Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity,
'ParameterName', 'ParameterValue ...)
```

Description

`Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity)` computes the static spread (Z-spread) to benchmark in basis points.

`Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` computes the static spread (Z-spread) to benchmark in basis points including optional inputs.

`Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity, 'ParameterName', 'ParameterValue ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. *'ParameterName'* is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to *'ParameterName'*. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

SpotInfo

This input can either be a matrix of two columns, or an annualized term structure created by `intenvset`.

- Matrix of two columns— The first column is the **SpotDate**, and the second column, **ZeroRates**, is the zero-rate corresponding to maturities on the **SpotDate**. It is recommended that the spot-rates are spaced as evenly apart as possible, perhaps one that is built from 3-months deposit rates. For example, using the 3-month deposit rates:

```
SpotInfo = ...  
[datenum('2-Jan-2004') , 0.03840;  
 datenum('2-Jan-2005') , 0.04512;  
 datenum('2-Jan-2006') , 0.05086];
```

- Annualized term structure — Refer to **intenvset** to create an annualized term structure. For example:

```
Settle = datenum('1-Jan-2004');  
Rates = [0.03840; 0.04512; 0.05086];  
EndDates = [datenum('2-Jan-2004'); datenum('2-Jan-2005');...  
            datenum('2-Jan-2006')];  
SpotInfo = intenvset('StartDates' , Settle ,...  
                    'Rates'      , Rates ,...  
                    'EndDates'   , EndDates,...  
                    'Compounding', 2     ,...  
                    'Basis'      , 0);
```

Price

Price for every \$100 notional amount of bonds whose spreads are computed. This is the clean price of the bond (current price without accrued interest).

Coupon

Annual coupon rate of bonds whose spreads are computed.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon

payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified **LastCouponDate** determines the coupon structure of the bond. The coupon structure of a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Face

Face or par value.

Default: 100

Parameter–Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Output Arguments

Spread

Static spread to benchmark, in basis points.

Examples

Compute the Static Spread Over a Spot Curve

This example shows how to compute a Federal National Mortgage Association (FNMA) 4 3/8 spread over a Treasury spot curve and plot the results.

```
RefMaturity = [datenum('02/27/2003');
               datenum('05/29/2003');
               datenum('10/31/2004');
               datenum('11/15/2007');
               datenum('11/15/2012');
               datenum('02/15/2031')];
```

```
RefCpn = [0;
          0;
          2.125;
          3;
          4;
          5.375] / 100;
```

```
RefPrices = [99.6964;
             99.3572;
             100.3662;
```

```
        99.4511;
        99.4299;
        106.5756];

RefBonds = [RefPrices, RefMaturity, RefCpn];
Settle    = datenum('26-Nov-2002');
[ZeroRates, CurveDates] = zbtprice(RefBonds(:, 2:end), ...
RefPrices, Settle)

% FNMA 4 3/8 maturing 10/06 at 4.30 pm Tuesday
Price     = 105.484;
Coupon    = 0.04375;
Maturity  = datenum('15-Oct-2006');

% All optional inputs are supposed to be accounted by default,
% except the accrued interest under 30/360 (SIA), so:
Period    = 2;
Basis     = 1;
SpotInfo  = [CurveDates, ZeroRates];

% Compute static spread over treasury curve, taking into account
% the shape of curve as derived by bootstrapping method embedded
% within bndspread.

SpreadInBP = bndspread(SpotInfo, Price, Coupon, Settle, ...
Maturity, Period, Basis)

plot(CurveDates, ZeroRates*100, 'b', CurveDates, ...
ZeroRates*100+SpreadInBP/100, 'r--')
legend({'Treasury'; 'FNMA 4 3/8'})
xlabel('Curve Dates')
ylabel('Spot Rate [%]')
grid;

ZeroRates =

    0.0121
    0.0127
    0.0194
    0.0317
    0.0423
    0.0550
```


CurveDates =

731639

731730

732251

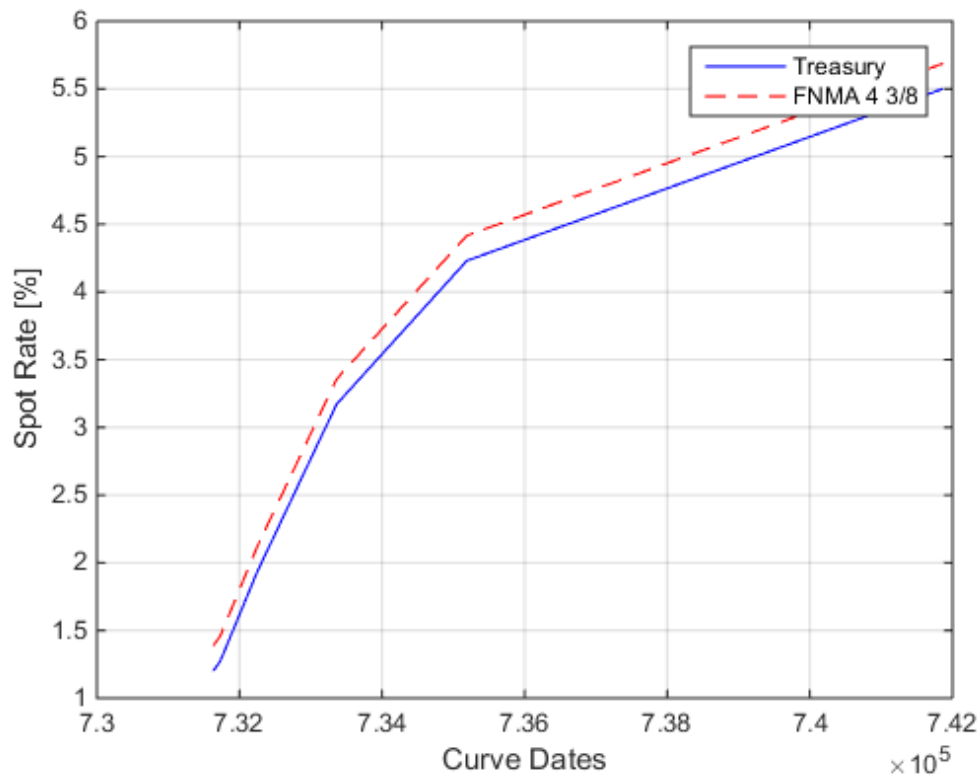
733361

735188

741854

SpreadInBP =

18.5669



References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, "Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures", *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

bndyield | bndprice

bndtotalreturn

Total return of fixed-coupon bond

Syntax

```
[BondEquiv,EffectiveRate] =  
bndtotalreturn(Price,CouponRate,Settle,Maturity,ReinvestRate)  
[BondEquiv,EffectiveRate] =  
bndtotalreturn(Price,CouponRate,Settle,Maturity,  
ReinvestRate,Name,Value)
```

Description

```
[BondEquiv,EffectiveRate] =  
bndtotalreturn(Price,CouponRate,Settle,Maturity,ReinvestRate)  
calculates the total return for fixed-coupon bonds to maturity or to a specific investment  
horizon.
```

```
[BondEquiv,EffectiveRate] =  
bndtotalreturn(Price,CouponRate,Settle,Maturity,  
ReinvestRate,Name,Value) calculates the total return for fixed-coupon bonds to  
maturity or to a specific investment horizon with additional options specified by one or  
more Name,Value pair arguments.
```

Input Arguments

Price

NINST-by-1 matrix for the clean price at the settlement date.

CouponRate

NINST-by-1 matrix for the coupon rate in decimal form.

Settle

NINST-by-1 vector for the settlement date of the fixed-coupon bond.

Maturity

NINST-by-1 vector of dates representing the maturity date of the fixed-coupon bond.

ReinvestRate

NINST-by-1 vector for the reinvestment rate in decimal form. This is the rate earned by reinvesting the coupons.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, . . . ,NameN,ValueN.

'HorizonDate'

Investment horizon date. If unspecified, the total return is calculated to maturity.

Default:**'HorizonPrice'**

Price at investment horizon date. If unspecified, the price at horizon will be calculated based on the reinvestment rate. If the horizon date equals the maturity date, this price is ignored and the total return to maturity is calculated based on the face value.

Default:**'Period'**

Number of coupon payments per year. Values are: 0, 1, 2, 3, 4, 6, 12.

Default: 2**'Basis'**

NINST-by-1 vector representing day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)

- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see “”.

Default: 0 (actual/actual)

'EndMonthRule'

Value for the end-of-month rule.

- 0 – Rule is not in effect for the bond(s).
- 1 – Rule is in effect for the bond(s). This means that a security that pays coupon interest on the last day of the month will always make payment on the last day of the month.

Default: 1 (in effect)

'IssueDate'

Bond issue date.

Default: If you do not specify an **IssueDate**, the cash flow payment dates are determined from other inputs.

'FirstCouponDate'

Irregular or normal first coupon date.

Default: If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

'LastCouponDate'

Irregular or normal last coupon date.

Default: If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

'StartDate'

Forward starting date of payments.

Default: If you do not specify a `StartDate`, the effective start date is the `Settle` date.

'Face'

Face value of the bond.

Default: 100

'CompoundingFrequency'

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semi-annual compounding convention and ISMA bases (8-12) use an annual compounding convention.

Default:

'DiscountBasis'

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the `actual/actual` day count to compute discount factors, and for ISMA day counts (8-12) and BUS/252 to use the specified basis.

Default:

Output Arguments

BondEquiv

NUMBONDS-by-1 vector for the total return in bond equivalent basis.

EffectiveRate

NUMBONDS-by-1 vector for the total return in effective rate basis.

Examples

Compute the Total Return of a Fixed-Coupon Bond

Use `bndtotalreturn` to compute the total return for a fixed-coupon bond, given an investment horizon date.

Define fixed-coupon bond.

```
Price = 101;
CouponRate = 0.05;
Settle = '15-Nov-2011';
Maturity = '15-Nov-2031';
ReinvestRate = 0.04;
```

Calculate the total return to maturity.

```
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...
Settle, Maturity, ReinvestRate)
```

```
BondEquiv =
```

```
0.0460
```

```
EffectiveRate =
```

```
0.0466
```

Specify an investment horizon.

```
HorizonDate = '15-Nov-2021';
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...
Settle, Maturity, ReinvestRate, 'HorizonDate', HorizonDate)
```

```
BondEquiv =
```



```
0.0521
```

```
EffectiveRate =
```

```
0.0528
```

Perform scenario analysis on the reinvestment rate.

```
ReinvestRate = [0.03; 0.035; 0.04; 0.045; 0.05];  
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...  
Settle, Maturity, ReinvestRate, 'HorizonDate', HorizonDate)
```

```
BondEquiv =
```

```
0.0557  
0.0538  
0.0521  
0.0505  
0.0490
```

```
EffectiveRate =
```

```
0.0565  
0.0546  
0.0528  
0.0511  
0.0496
```

References

Fabozzi, Frank J., Mann, Steven V., *Introduction to Fixed Income Analytics: Relative Value Analysis, Risk Measures and Valuation*, John Wiley and Sons, New York, 2010.

See Also

| bndyield | bndprice | cfamounts

bndyield

Yield to maturity for fixed-income security

Syntax

```
Yield = bndyield(Price, CouponRate, Settle, Maturity)
Yield = bndyield(Price, CouponRate, Settle, Maturity,
Period, Basis, EndMonthRule, IssueDate, FirstCouponDate,
LastCouponDate, StartDate, Face)
Yield = bndyield(Price, CouponRate, Settle, Maturity,
'ParameterName', 'ParameterValue ...)
```

Description

`Yield = bndyield(Price, CouponRate, Settle, Maturity)`, given `NUMBONDS` bonds with SIA date parameters and clean prices (excludes accrued interest), returns the bond equivalent yields to maturity.

`Yield = bndyield(Price, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` bonds with SIA date parameters and clean prices (excludes accrued interest) and optional inputs, returns the bond equivalent yields to maturity.

`Yield = bndyield(Price, CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. *ParameterName* is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to *ParameterName*. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Price

Clean price of the bond (current price without accrued interest).

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2**Basis**

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)

- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

Default: If you do not specify an **IssueDate**, the cash flow payment dates are determined from other inputs.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified **LastCouponDate** determines the coupon structure of the bond. The coupon structure of

a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Default: If you do not specify **StartDate**, the effective start date is the **Settle** date.

Face

Face or par value.

Default: 100

Parameter–Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

Default: SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Default: SIA bases use the **actual/actual** day count to compute discount factors.

LastCouponInterest

Compounding convention for computing the yield of a bond in the last coupon period. This computation is based on only the last coupon and the face value to be repaid. Acceptable values are `simple` or `compound`.

Default: `compound`

Output Arguments

Yield

NUMBONDS-by-1 vector of the yield to maturity with semiannual compounding.

Definitions

All nonscalar or empty matrix input arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors. Fill in unspecified entries input vectors with NaNs. Dates can be serial date numbers or date strings.

Examples

Compute Yield to Maturity for a Treasury Bond

This example shows how to compute the yield of a Treasury bond at three different price values.

```
Price = [95; 100; 105];  
CouponRate = 0.05;  
Settle = '20-Jan-1997';  
Maturity = '15-Jun-2002';  
Period = 2;  
Basis = 0;
```

```
Yield = bndyield(Price, CouponRate, Settle,...  
Maturity, Period, Basis)
```

```
Yield =
```

0.0610
 0.0500
 0.0396

More About

Algorithms

For SIA conventions, the following formula defines bond price and yield:

$$PV = \frac{CF}{\left(1 + \frac{z}{f}\right)^{TF}},$$

where:

$PV =$	Present value of a cash flow.
$CF =$	The cash flow amount.
$z =$	The risk-adjusted annualized rate or yield corresponding to a given cash flow. The yield is quoted on a semiannual basis.
$f =$	The frequency of quotes for the yield.
$TF =$	Time factor for a given cash flow. Time is measured in semiannual periods from the settlement date to the cash flow date. In computing time factors, use SIA actual/actual day count conventions for all time factor calculations.

For ISMA conventions, the frequency of annual coupon payments determines bond price and yield.

- “Yield Functions”

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

bndprice | cfamounts

bolling

Bollinger band chart

Syntax

```
bolling(Asset, Samples, Alpha, Width)
```

```
[Movavgv, UpperBand, LowerBand] = bolling(Asset, Samples, Alpha, Width)
```

Arguments

Asset	Vector of asset data.
Samples	Number of samples to use in computing the moving average.
Alpha	(Optional) Exponent used to compute the element weights of the moving average. Default = 0 (simple moving average).
Width	(Optional) Number of standard deviations to include in the envelope. A multiplicative factor specifying how tight the bands should be around the simple moving average. Default = 2.

Description

`bolling(Asset, Samples, Alpha, Width)` plots Bollinger bands for given `Asset` data. This form of the function does not return any data.

`[Movavgv, UpperBand, LowerBand] = bolling(Asset, Samples, Alpha, Width)` returns `Movavgv` with the moving average of the `Asset` data, `UpperBand` with the upper band data, and `LowerBand` with the lower band data. This form of the function does not plot any data.

Note The standard deviations are normalized by $N-1$, where N = the sequence length.

Examples

If `Asset` is a column vector of closing stock prices

```
bolling(Asset, 20, 1)
```

plots linear 20-day moving average Bollinger bands based on the stock prices.

```
[Movavgv, UpperBand, LowerBand] = bolling(Asset, 20, 1)
```

returns `Movavgv`, `UpperBand`, and `LowerBand` as vectors containing the moving average, upper band, and lower band data, without plotting the data.

See Also

`candle` | `dateaxis` | `movavg` | `highlow` | `pointfig`

bollinger

Time series Bollinger band

Syntax

```
[mid, uppr, lowr] = bollinger(data, wsize, wts, nstd)
```

```
[midfts, upprfts, lowrfts] = bollinger(tsobj, wsize, wts, nstd)
```

Arguments

data	Data vector.
wsize	(Optional) Window size. Default = 20.
wts	(Optional) Weight factor. Determines the type of moving average used. Default = 0 (box). 1 = linear.
nstd	(Optional) Number of standard deviations for upper and lower bands. Default = 2.
tsobj	Financial time series object.

Description

`[mid, uppr, lowr] = bollinger(data, wsize, wts, nstd)` calculates the middle (**mid**), upper (**uppr**), and lower (**lowr**) bands that make up the Bollinger bands from the vector **data**.

mid is the vector that represents the middle band, a simple moving average with a window size of **wsize**. **uppr** and **lowr** are vectors that represent the upper and lower bands. **uppr** is a vector representing the upper band that is **+nstd** times. **lowr** is a vector representing the lower band that is **-nstd** times.

`[midfts, upprfts, lowrfts] = bollinger(tsobj, wsize, wts, nstd)` calculates the middle, upper, and lower bands that make up the Bollinger bands from a financial time series object **tsobj**.

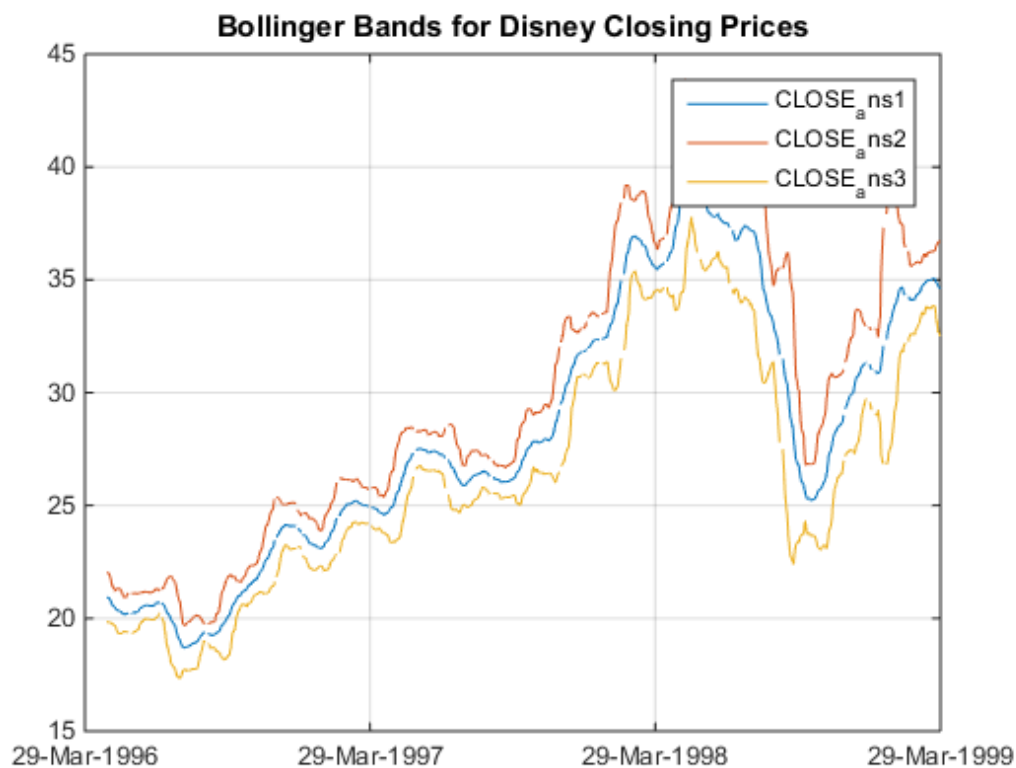
`midfts` is a financial time series object that represents the middle band for all series in `tsobj`. Both `upprfts` and `lowrfts` are financial time series objects that represent the upper and lower bands of all series, which are `+nstd` times and `-nstd` times moving standard deviations away from the middle band.

Examples

Create a Bollinger Bands Plot

This example shows how to compute the Bollinger bands for Disney stock closing prices and plot the results.

```
load disney.mat
[dis_Mid,dis_Uppr,dis_Lowr]= bollinger(dis);
dis_CloseBolling = [dis_Mid.CLOSE, dis_Uppr.CLOSE,...
dis_Lowr.CLOSE];
plot(dis_CloseBolling)
title('Bollinger Bands for Disney Closing Prices')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second Edition, McGraw-Hill, 1995, pp. 72-74.

See Also

tsmovavg

boxcox

Box-Cox transformation

Syntax

```
[transdat, lambda] = boxcox(data)
```

```
[transfts, lambda] = boxcox(tsobj)
```

```
transdat = boxcox(lambda, data)
```

```
transfts = boxcox(lambda, tsobj)
```

Arguments

<code>data</code>	Data vector. Must be positive and specified as a column data vector.
<code>tsobj</code>	Financial time series object.

Description

`boxcox` transforms nonnormally distributed data to a set of data that has approximately normal distribution. The Box-Cox transformation is a family of power transformations.

If λ is not = 0, then

$$data(\lambda) = \frac{data^\lambda - 1}{\lambda}$$

If λ is = 0, then

$$data(\lambda) = \log(data)$$

The logarithm is the natural logarithm (log base e). The algorithm calls for finding the λ value that maximizes the Log-Likelihood Function (LLF). The search is conducted using `fminsearch`.

`[transdat, lambda] = boxcox(data)` transforms the data vector `data` using the Box-Cox transformation method into `transdat`. It also estimates the transformation parameter λ .

`[transfts, lambda] = boxcox(tsojb)` transforms the financial time series object `tsojb` using the Box-Cox transformation method into `transfts`. It also estimates the transformation parameter λ .

If the input data is a vector, `lambda` is a scalar. If the input is a financial time series object, `lambda` is a structure with fields similar to the components of the object; for example, if the object contains series names `Open` and `Close`, `lambda` has fields `lambda.Open` and `lambda.Close`.

`transdat = boxcox(lambda, data)` and `transfts = boxcox(lambda, tsojb)` transform the data using a certain specified λ for the Box-Cox transformation. This syntax does not find the optimum λ that maximizes the LLF.

See Also

`fminsearch`

busdate

Next or previous business day

Syntax

```
Busday = busdate(Date, DirFlag, Holiday, Weekend)
```

Arguments

Date	Reference date. Enter scalar, vector, or matrix of reference business dates as serial date numbers or date strings.
DirFlag	(Optional) String or cell array of strings of business day convention with possible values: <code>follow</code> (default), <code>modifiedfollow</code> , <code>previous</code> , <code>modifiedprevious</code> . Also, <code>DirFlag</code> may be a scalar, vector, or matrix of search directions, where Next is <code>DIREC = 1</code> (default) or Previous is <code>DIREC = -1</code> .
Holiday	(Optional) Vector of holidays and nontrading-day dates. All dates in <code>Holiday</code> must be the same format: either serial date numbers or date strings. (Using serial date numbers improves performance.) If <code>Holiday</code> is not specified, the non-trading day default vector is determined by the routine <code>holidays</code> function.
Weekend	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), <code>Weekend = [1 0 0 0 0 0 1]</code> .

Description

`Busday = busdate(Date, DirFlag, Holiday, Weekend)` returns the scalar, vector, or matrix of the next or previous business day(s), depending on `Holiday`.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

Example 1.

```
Busday = busdate('3-Jul-2001', 1)
```

```
Busday =
```

```
    731037
```

```
datestr(Busday)
```

```
ans =
```

```
05-Jul-2001
```

Example 2. You can indicate that Saturday is a business day by appropriately setting the `Weekend` argument.

```
Weekend = [1 0 0 0 0 0 0];
```

July 4, 2003 falls on a Friday. Use `busdate` to verify that Saturday, July 5, is actually a business day.

```
Date = datestr(busdate('3-Jul-2003', 1, [], Weekend))
```

See Also

`holidays` | `isbusday`

busdays

Business days in serial date format

Syntax

```
bdates = busdays(sdate, edate, bdmode)
```

```
bdates = busdays(sdate, edate, bdmode, holvec)
```

Arguments

sdate	Start date in string or serial date format.
edate	End date in string or serial date format.
bdmode	<p>(Optional) Frequency of business days:</p> <ul style="list-style-type: none"> • DAILY, Daily, daily, D, d, 1 (default) • WEEKLY, Weekly, weekly, W, w, 2 • MONTHLY, Monthly, monthly, M, m, 3 • QUARTERLY, Quarterly, quarterly, Q, q, 4 • SEMIANNUAL, Semiannual, semiannual, S, s, 5 • ANNUAL, Annual, annual, A, a, 6 <p>Strings must be enclosed in single quotation marks.</p>
holvec	(Optional) Holiday dates vector in string or serial date format. Note that using a <code>holvec</code> value of <code>NaN</code> will use a holiday list that has no dates.

Description

`bdates = busdays(sdate, edate, bdmode)` generates a vector of business days, `bdates`, in serial date format between the last business date of the period that contains the start date, and the last business date of period that contains the end date. If `holvec`

is not supplied, the dates are generated based on United States holidays. If you do not supply `bdmode`, `busdays` generates a daily vector.

For example:

```
vec = datestr(busdays('1/2/01','1/9/01','weekly'))
```

```
vec =  
05-Jan-2001  
12-Jan-2001
```

The end of the week is considered to be a Friday. Between 1/2/01 (Monday) and 1/9/01 (Tuesday) there is only one end-of-week day, 1/5/01 (Friday).

Because 1/9/01 is part of following week, the following Friday (1/12/01) is also reported.

`bdates = busdays(sdate, edate, bdmode, holvec)` lets you supply a vector of holidays, `holvec`, used to generate business days. `holvec` can either be in serial date format or date string format. If you use this syntax, you need to supply the frequency `bdmode`.

The output, `bdates`, is a column vector of business dates in serial date format.

Setting `holvec` to `''` (empty string) or `[]` (empty vector) results in `BUSDAYS` using a default holiday schedule. The default holiday schedule is the NYSE holiday schedule.

candle

Candlestick chart

Syntax

```
candle(HighPrices, LowPrices, ClosePrices, OpenPrices)
```

```
candle(HighPrices, LowPrices, ClosePrices, OpenPrices,  
Color, Dates, Dateform)
```

Arguments

HighPrices	High prices for a security. A column vector.
LowPrices	Low prices for a security. A column vector.
ClosePrices	Closing prices for a security. A column vector.
OpenPrices	Opening prices for a security. A column vector.
Color	(Optional) Candlestick color. A string. MATLAB software supplies a default color if none is specified. The default color differs depending on the background color of the figure window. See <code>ColorSpec</code> in the MATLAB documentation for color names.
Dates	(Optional) Column vector of dates for user specified X-axis tick labels.
Dateform	(Optional) Date string format used as the x-axis tick labels. (See <code>datetick</code> in the MATLAB documentation.) You can specify a <code>dateform</code> only when <code>tsobj</code> does not contain time-of-day data. If <code>tsobj</code> contains time-of-day data, <code>dateform</code> is restricted to 'dd-mmm-yyyy HH:MM'.

Description

`candle(HighPrices, LowPrices, ClosePrices, OpenPrices)` plots a candlestick chart given column vectors with the high, low, closing, and opening prices of a security.

If the closing price is greater than the opening price, the body (the region between the opening and closing price) is unfilled.

If the opening price is greater than the closing price, the body is filled.

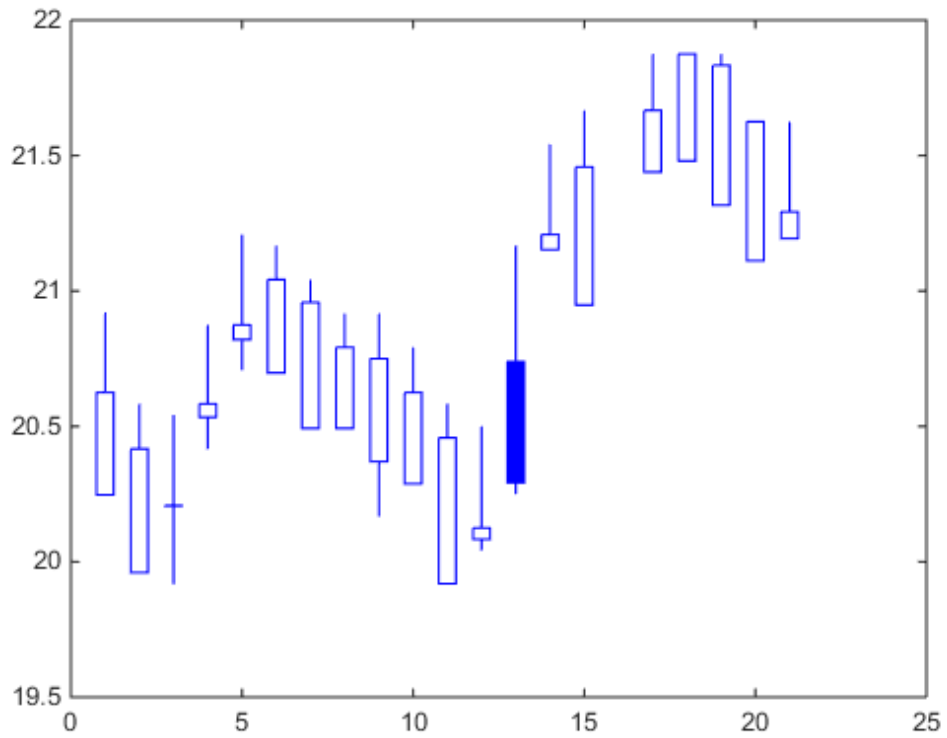
`candle(HighPrices, LowPrices, ClosePrices, OpenPrices, Color, Dates, Dateform)` plots a candlestick chart given column vectors with the high, low, closing, and opening prices of a security. In addition, the optional arguments `Color`, `Dates`, and `Dateform` specify the color of the candle box and the date string format used as the *x*-axis tick labels.

Examples

Create a Candlestick Chart

This example shows how to create a candlestick chart, with blue candles, for the most recent 21 days in `disney.mat`.

```
load disney;
candle(dis_HIGH(end-20:end), dis_LOW(end-20:end), dis_CLOSE(end-20:end),...
dis_OPEN(end-20:end), 'b');
```



See Also

`bollinger` | `candle` | `dateaxis` | `highlow` | `movavg` | `pointfig`

candle (fts)

Time series candle plot

Syntax

```
candle(tsobj)
```

```
candle(tsobj, color)
```

```
candle(tsobj, color, dateform)
```

```
candle(tsobj, color, dateform, ParameterName, ParameterValue, ...)
```

```
hcdl = candle(tsobj, color, dateform, ParameterName,  
ParameterValue, ...)
```

Arguments

<code>tsobj</code>	Financial time series object
<code>color</code>	(Optional) A three-element row vector representing RGB or a color identifier. (See <code>plot</code> in the MATLAB documentation.)
<code>dateform</code>	(Optional) Date string format used as the <i>x</i> -axis tick labels. (See <code>datetick</code> in the MATLAB documentation.) You can specify a <code>dateform</code> only when <code>tsobj</code> does not contain time-of-day data. If <code>tsobj</code> contains time-of-day data, <code>dateform</code> is restricted to 'dd-mmm-yyyy HH:MM'.

Description

`candle(tsobj)` generates a candle plot of the data in the financial time series object `tsobj`. `tsobj` must contain at least four data series representing the high, low, open, and closing prices. These series must have the names `High`, `Low`, `Open`, and `Close` (case-insensitive).

`candle(tsobj, color)` additionally specifies the color of the candle box.

`candle(tsobj, color, dateform)` additionally specifies the date string format used as the *x*-axis tick labels. See `datestr` for a list of date string formats.

`candle(tsobj, color, dateform, ParameterName, ParameterValue, ...)` indicates the actual name(s) of the required data series if the data series do not have the default names. `ParameterName` can be

- `HighName`: high prices series name
- `LowName`: low prices series name
- `OpenName`: open prices series name
- `CloseName`: closing prices series name

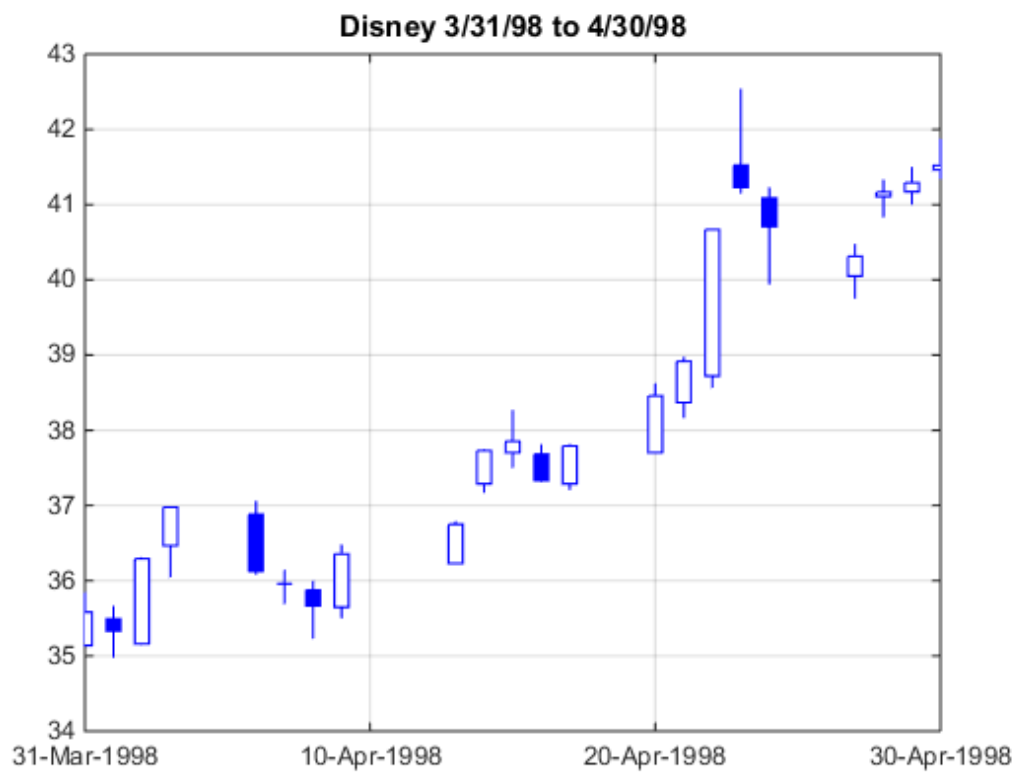
`hcdl = candle(tsobj, color, dateform, ParameterName, ParameterValue, ...)` returns the handle to the patch objects and the line object that make up the candle plot. `hcdl` is a three-element column vector representing the handles to the two patches and one line that forms the candle plot.

Examples

Create a Candle Plot for a Financial Time Series Object

This example shows how to create a candle plot for Disney stock for the dates March 31, 1998 through April 30, 1998.

```
load disney.mat
candle(dis('3/31/98':4/30/98'))
title('Disney 3/31/98 to 4/30/98')
```

See Also

`candle` | `chartfts` | `highlow` | `plot`

cdai

Accrued interest on certificate of deposit

Syntax

`AccrInt = cdai(CouponRate, Settle, Maturity, IssueDate, Basis)`

Arguments

CouponRate	Annual interest rate in decimal.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.
IssueDate	Issue date.
Basis	<p>(Optional) Day-count basis of the instrument.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see “”.</p>

Each required input must be some certificates of deposit (NCDS)-by-1 or 1-by-NCDS conforming vector or scalar. The optional `Basis` argument may be either a NCDS-by-1 or a 1-by-NCDS vector, a scalar, or the empty matrix (`[]`).

Description

`AccrInt = cdai(CouponRate, Settle, Maturity, IssueDate, Basis)` computes the accrued interest on a certificate of deposit.

`AccrInt` represents the accrued interest per \$100 of face value.

This function assumes that the certificates of deposit pay interest at maturity. Because of the simple interest treatment of these securities, the function is best used for short-term maturities (less than 1 year). The default simple interest calculation is the actual/360 convention (SIA).

Examples

Find the Accrued Interest on a Certificate of Deposit

This example shows how to compute the accrued interest due, given a certificate of deposit with the following characteristics.

```
CouponRate = 0.05;
Settle      = '02-Jan-02';
Maturity    = '31-Mar-02';
IssueDate   = '1-Oct-01';
```

```
AccrInt = cdai(CouponRate, Settle, Maturity, IssueDate)
```

```
AccrInt =
    1.2917
```

See Also

`accfrac` | `bndyield` | `stepcpnyield` | `tbillyield` | `cdprice` | `cdyield` | `zeroyield`

cdprice

Price of certificate of deposit

Syntax

```
[Price, AccrInt] = cdprice(Yield, CouponRate, Settle, Maturity,
IssueDate, Basis)
```

Arguments

Yield	Simple yield to maturity over the basis denominator.
CouponRate	Coupon interest rate in decimal.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.
IssueDate	Issue date.
Basis	<p>(Optional) Day-count basis of the instrument.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA)

- 13 = BUS/252

For more information, see “”.

Each required input must be some certificates of deposit (NCDS)-by-1 or 1-by-NCDS conforming vector or scalar. The optional **Basis** argument may be either a NCDS-by-1 or a 1-by-NCDS vector, a scalar, or the empty matrix ([]).

Description

[Price, AccrInt] = cdprice(Yield, CouponRate, Settle, Maturity, IssueDate, Basis) computes the price of a certificate of deposit given its yield.

Price is the clean price of the certificate of deposit per \$100 of face value.

AccruedInt is the accrued interest payable at settlement per unit of face value.

This function assumes that the certificates of deposit pay interest at maturity. Because of the simple interest treatment of these securities, the function is best used for short-term maturities (less than 1 year). The default simple interest calculation is the actual/360 convention.

Examples

Compute the Price and Accrued Interest for a Certificate of Deposit

This example shows how to compute the price and the accrued interest due on the settlement date, given a certificate of deposit with the following characteristics.

```
Yield          = 0.0525;
CouponRate     = 0.05;
Settle         = '02-Jan-02';
Maturity       = '31-Mar-02';
IssueDate      = '1-Oct-01';
```

```
[Price, AccruedInt] = cdprice(Yield, CouponRate, Settle, ...
Maturity, IssueDate)
```

```
Price =
```

99.9233

AccruedInt =

1.2917

See Also

bndprice | cdai | cdyield | stepcpnprice | tbillprice

cdyield

Yield on certificate of deposit (CD)

Syntax

Yield = cdyield(Price, CouponRate, Settle, Maturity, IssueDate, Basis)

Arguments

Price	Clean price of the certificate of deposit per \$100 face. If you have a vector of dirty or cash prices of CDs, compute the accrued interest portion using <code>cdai</code> .
CouponRate	Annual interest rate in decimal.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.
IssueDate	Issue date.
Basis	(Optional) Day-count basis of the instrument. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA)

- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see “”.

Each required input must be some certificates of deposit (NCDS)-by-1 or 1-by-NCDS conforming vector or scalar. The optional **Basis** argument may be either a NCDS-by-1 or a 1-by-NCDS vector, a scalar, or the empty matrix ([]).

Description

`Yield = cdyield(Price, CouponRate, Settle, Maturity, IssueDate, Basis)` computes the yield to maturity of a certificate of deposit given its clean price.

This function assumes that the certificates of deposit pay interest at maturity. Because of the simple interest treatment of these securities, the function is best used for short-term maturities (less than 1 year). The default simple interest calculation is the actual/360 convention.

Examples

Compute the Yield to Maturity of a Certificate of Deposit

This example shows how to compute the yield on the certificate of deposit (CD), given a CD with the following characteristics.

```
Price      = 101.125;
CouponRate = 0.05;
Settle     = '02-Jan-02';
Maturity   = '31-Mar-02';
IssueDate  = '1-Oct-01';
```

```
Yield = cdyield(Price, CouponRate, Settle, Maturity, IssueDate)
```

```
Yield =
    0.0039
```


See Also

bndprice | cdai | cdprice | stepcpnprice | tbillprice

cev

Construct constant elasticity of variance models (objects of class CEV)

Syntax

```
CEV = cev(Return, Alpha, Sigma)
```

```
CEV = cev(Return, Alpha, Sigma, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Class

CEV

Description

This constructor creates and displays CEV objects, which derive from the SDELD (SDE with drift rate expressed in linear form) class. 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:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

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

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Return	<p>Return represents the parameter μ.</p> <p>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. As a deterministic function of time, when Return is called with a real-valued scalar time \mathbf{t} as its only input, Return must produce an NVARs-by-NVARs matrix.</p> <p>If you specify Return as a function of time and state, it calculates the expected instantaneous rate of return. This function must generate an NVARs-by-NVARs matrix when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Alpha	<p>Alpha determines the format of the parameter D.</p> <p>If you specify Alpha as an array, it represents an NVARs-by-1 column vector of exponents. As a deterministic function of time, when Alpha</p>

	<p>is called with a real-valued scalar time t as its only input, Alpha must produce an NVARs-by-1 matrix.</p> <p>If you specify it as a function of time and state, Alpha must return an NVARs-by-1 column vector of exponents when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V.</p> <p>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. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARs-by-NBROWNS matrix.</p> <p>If you specify it as a function of time and state, Sigma must generate an NVARs-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.

Note: Although the constructor does not enforce restrictions on the signs of these input arguments, each argument is usually specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, cev applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, cev applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, cev applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>
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	<p>Object of class CEV with the following displayed parameters:</p> <ul style="list-style-type: none"> • 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
-----	---

Examples

- “Creating Constant Elasticity of Variance (CEV) Models” on page 17-25
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`drift` | `diffusion` | `sdeld`

cfamounts

Cash flow and time mapping for bond portfolio

Syntax

```
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags,  
CFPrincipal] = cfamounts(CouponRate, Settle, Maturity)  
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags,  
CFPrincipal] = cfamounts(CouponRate, Settle, Maturity, Period,  
Basis, EndMonthRule, IssueDate, FirstCouponDate,  
LastCouponDate, StartDate, Face)  
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags,  
CFPrincipal] = cfamounts(CouponRate, Settle, Maturity,  
'ParameterName', 'ParameterValue ...')
```

Description

[CFlowAmounts, CFlowDates, TFactors, CFlowFlags, CFPrincipal] = cfamounts(CouponRate, Settle, Maturity) returns matrices of cash flow amounts, cash flow dates, time factors, and cash flow flags for a portfolio of NUMBONDS fixed-income securities.

[CFlowAmounts, CFlowDates, TFactors, CFlowFlags, CFPrincipal] = cfamounts(CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) returns matrices of cash flow amounts, cash flow dates, time factors, and cash flow flags for a portfolio of NUMBONDS fixed-income securities defined using required and optional inputs.

[CFlowAmounts, CFlowDates, TFactors, CFlowFlags, CFPrincipal] = cfamounts(CouponRate, Settle, Maturity, 'ParameterName', 'ParameterValue ...') accepts optional inputs as one or more comma-separated parameter/value pairs. 'ParameterName' is the name of the parameter inside single quotes. ParameterValue is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

CouponRate

Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond. **CouponRate** is 0 for zero coupon bonds.

Note: **CouponRate** and **Face** can change over the life of the bond. Schedules for **CouponRate** and **Face** can be specified with an **NINST**-by-1 cell array, where each element is a **NumDates**-by-2 matrix or cell array, where the first column is dates and the second column is associated rates. The date indicates the last day that the coupon rate or face value is valid.

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360

- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days.

- 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

Default: If you do not specify an **IssueDate**, the cash flow payment dates are determined from other inputs.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

Default: If you do not specify `StartDate`, the effective start date is the `Settle` date.

Face

Face or par value.

Note: `CouponRate` and `Face` can change over the life of the bond. Schedules for `CouponRate` and `Face` can be specified with an NINST-by-1 cell array where each element is a NumDates-by-2 matrix or cell array, where the first column is dates and the second column is associated rates. The date indicates the last day that the coupon rate or face value is valid.

Default: 100

Parameter-Value Pairs

Enter the following inputs only as parameter/value pairs.

AdjustCashFlowsBasis

Adjust the cash flows based on the actual period day count. NINST-by-1 of logicals.

Default: False

BusinessDayConvention

Require payment dates to be business dates. NINST-by-1 cell array with possible choices of business day convention:

- actual
- follow
- modifiedfollow
- previous
- modifiedprevious

Default: actual

CompoundingFrequency

Compounding frequency for yield calculation.

Default: SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. If you use ISMA day counts and BUS/252, the specified basis are used.

Default: SIA bases use the `actual/actual` day count to compute discount factors.

Holidays

Holidays used for business day convention. NHOLIDAYS-by-1 of MATLAB date numbers.

Default: If no dates are specified, `holidays.m` is used.

PrincipalType

Type of principal for case when a `Face` schedule is specified. The principal type is either `sinking` or `bullet`. If `sinking`, principal cash flows are returned throughout the life of the bond. If `bullet`, principal cash flow is only returned at maturity.

Default: sinking

Output Arguments

CFlowAmounts

Cash flow matrix of a portfolio of bonds. Each row represents the cash flow vector of a single bond. Each element in a column represents a specific cash flow for that bond.

CFlowDates

Cash flow date matrix of a portfolio of bonds. Each row represents a single bond in the portfolio. Each element in a column represents a cash flow date of that bond.

TFactors

Matrix of time factors for a portfolio of bonds. Each row corresponds to the vector of time factors for each bond. Each element in a column corresponds to the specific time factor associated with each cash flow of a bond. Time factors help determine the present value of a stream of cash flows. The term *time factor* refers to the exponent *TF* in the discounting equation

$$PV = \sum_{i=1}^n \left(\frac{CF}{\left(1 + \frac{z}{f}\right)^{TF}} \right)$$

where:

<i>PV</i> =	Present value of a cash flow.
<i>CF</i> =	Cash flow amount.
<i>z</i> =	Risk-adjusted annualized rate or yield corresponding to a given cash flow. The yield is quoted on a semiannual basis.
<i>f</i> =	Frequency of quotes for the yield. Default is 2 for Basis values 0 to 7 and 13 and 1 for Basis values 8 to 12. The default can be overridden by specifying the CompoundingFrequency name/value pair.

$TF =$	Time factor for a given cash flow. The time factor is computed using the compounding frequency and the discount basis. If these values are not specified, then the defaults are as follows: CompoundingFrequency default is 2 for Basis values 0 to 7 and 13 and 1 for Basis values 8 to 12. DiscountBasis is 0 for Basis values 0 to 7 and 13 and the input Basis for Basis values 8 to 12.
--------	---

Note: The **Basis** is always used to compute accrued interest.

CFlowFlags

Matrix of cash flow flags for a portfolio of bonds. Each row corresponds to the vector of cash flow flags for each bond. Each element in a column corresponds to the specific flag associated with each cash flow of a bond. Flags identify the type of each cash flow (for example, nominal coupon cash flow, front, or end partial, or “stub” coupon, maturity cash flow).

Flag	Cash Flow Type
0	Accrued interest due on a bond at settlement.
1	Initial cash flow amount smaller than normal due to a “stub” coupon period. A stub period is created when the time from issue date to first coupon date is shorter than normal.
2	Larger than normal initial cash flow amount because the first coupon period is longer than normal.
3	Nominal coupon cash flow amount.
4	Normal maturity cash flow amount (face value plus the nominal coupon amount).
5	End “stub” coupon amount (last coupon period is abnormally short and actual maturity cash flow is smaller than normal).
6	Larger than normal maturity cash flow because the last coupon period longer than normal.
7	Maturity cash flow on a coupon bond when the bond has less than one coupon period to maturity.
8	Smaller than normal maturity cash flow when the bond has less than one coupon period to maturity.

Flag	Cash Flow Type
9	Larger than normal maturity cash flow when the bond has less than one coupon period to maturity.
10	Maturity cash flow on a zero coupon bond.
11	Sinking principal and initial cash flow amount smaller than normal due to a "stub" coupon period. A stub period is created when the time from issue date to first coupon date is shorter than normal.
12	Sinking principal and larger than normal initial cash flow amount because the first coupon period is longer than normal.
13	Sinking principal and nominal coupon cash flow amount.

CFPrincipal

CFPrincipal contains the principal cash flows. If PrincipalType is bullet, CFPrincipal is all zeros and, at Maturity, the appropriate Face value.

Definitions

The elements contained in the cfamounts cash flow matrix, time factor matrix, and cash flow flag matrix correspond to the cash flow dates for each security. The first element of each row in the cash flow matrix is the accrued interest payable on each bond. This accrued interest is zero in the case of all zero coupon bonds. cfamounts determines all cash flows and time mappings for a bond whether or not the coupon structure contains odd first or last periods. All output matrices are padded with NaNs as necessary to ensure that all rows have the same number of elements.

Examples

Compute the Cash Flow Structure and Time Factors for a Bond Portfolio

This example shows how to compute the cash flow structure and time factors for a bond portfolio that contains a corporate bond paying interest quarterly and a Treasury bond paying interest semiannually.

```
Settle = '01-Nov-1993';
Maturity = ['15-Dec-1994'; '15-Jun-1995'];
```

```
CouponRate= [0.06; 0.05];
Period = [4; 2];
Basis = [1; 0];
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ...
cfamounts(CouponRate,Settle, Maturity, Period, Basis)
```

```
CFlowAmounts =
```

```
    -0.7667    1.5000    1.5000    1.5000    1.5000   101.5000
    -1.8989    2.5000    2.5000    2.5000   102.5000         NaN
```

```
CFlowDates =
```

```
    728234    728278    728368    728460    728552    728643
    728234    728278    728460    728643    728825         NaN
```

```
TFactors =
```

```
    0    0.2404    0.7403    1.2404    1.7403    2.2404
    0    0.2404    1.2404    2.2404    3.2404         NaN
```

```
CFlowFlags =
```

```
    0    3    3    3    3    4
    0    3    3    3    4   NaN
```

Compute the Cash Flow Structure and Time Factors for a Bond Portfolio Using Optional Name-Value Pairs

This example shows how to compute the cash flow structure and time factors for a bond portfolio that contains a corporate bond paying interest quarterly and a Treasury bond paying interest semiannually. This example uses the following Name-Value pairs for `Period`, `Basis`, `BusinessDayConvention`, and `AdjustCashFlowsBasis`.

```
Settle = '01-Jun-2010';
Maturity = ['15-Dec-2011'; '15-Jun-2012'];
CouponRate= [0.06; 0.05];
Period = [4; 2];
Basis = [1; 0];
```



```
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ...
cfamounts(CouponRate,Settle, Maturity, 'Period',Period, ...
'Basis', Basis, 'AdjustCashFlowsBasis', true,...
'BusinessDayConvention','modifiedfollow')
```

CFlowAmounts =

Columns 1 through 7

-1.2667	1.5000	1.5000	1.5000	1.5000	1.5000	1.5000
-2.3077	2.4932	2.5068	2.4932	2.5000	102.5000	NaN

Column 8

101.5000
NaN

CFlowDates =

Columns 1 through 6

734290	734304	734396	734487	734577	734669
734290	734304	734487	734669	734852	735035

Columns 7 through 8

734761	734852
NaN	NaN

TFactors =

Columns 1 through 7

0	0.0778	0.5778	1.0778	1.5778	2.0778	2.5778
0	0.0769	1.0769	2.0769	3.0769	4.0769	NaN

Column 8

3.0778
NaN

```
CFlowFlags =  
  
    0    3    3    3    3    3    3    4  
    0    3    3    3    3    4   NaN   NaN
```

Use cfamounts With a CouponRate Schedule

This example shows how to use `cfamounts` with a `CouponRate` schedule. For `CouponRate` and `Face` that change over the life of the bond, schedules for `CouponRate` and `Face` can be specified with an NINST-by-1 cell array, where each element is a `NumDates`-by-2 matrix where the first column is dates and the second column is associated rates.

```
CouponSchedule = {[datenum('15-Mar-2012') .04;datenum('15- Mar -2013') .05;...  
datenum('15- Mar -2015') .06]}  
cfamounts(CouponSchedule,'01-Mar-2011','15-Mar-2015' )
```

```
CouponSchedule =
```

```
 [3x2 double]
```

```
ans =
```

```
Columns 1 through 7
```

```
 -1.8453    2.0000    2.0000    2.0000    2.5000    2.5000    3.0000
```

```
Columns 8 through 10
```

```
 3.0000    3.0000  103.0000
```

Use cfamounts With a Face Schedule

This example shows how to use `cfamounts` with a `Face` schedule. For `CouponRate` and `Face` that change over the life of the bond, schedules for `CouponRate` and `Face` can be specified with an NINST-by-1 cell array, where each element is a `NumDates`-by-2 matrix where the first column is dates and the second column is associated rates.

```
FaceSchedule = {[datenum('15-Mar-2012') 100;datenum('15- Mar -2013') 90;...
```

```
datenum('15-Mar-2015') 80]}
cfamounts(.05,'01-Mar-2011','15-Mar-2015','Face',FaceSchedule)
```

```
FaceSchedule =
```

```
[3x2 double]
```

```
ans =
```

```
Columns 1 through 7
```

```
-2.3066    2.5000    2.5000    12.5000    2.2500    12.2500    2.0000
```

```
Columns 8 through 10
```

```
2.0000    2.0000    82.0000
```

Use cfamounts to Generate the Cash Flows for a Sinking Bond

This example shows how to use cfamounts to generate the cash flows for a sinking bond.

```
[CFlowAmounts,CFDates,TFactors,CFFlags,CFPrincipal] = cfamounts(.05,'04-Nov-2010',...
{'15-Jul-2014';'15-Jul-2015'},'Face',{[datenum('15-Jul-2013') 100;datenum('15-Jul-2014'
90;datenum('15-Jul-2015') 80]})
```

```
CFlowAmounts =
```

```
Columns 1 through 7
```

```
-1.5217    2.5000    2.5000    2.5000    2.5000    2.5000    12.5000
-1.5217    2.5000    2.5000    2.5000    2.5000    2.5000    12.5000
```

```
Columns 8 through 11
```

```
2.2500    92.2500         NaN         NaN
2.2500    12.2500    2.0000    82.0000
```

```
CFDates =
```

```
Columns 1 through 6
```

734446	734518	734699	734883	735065	735249
734446	734518	734699	734883	735065	735249

Columns 7 through 11

735430	735614	735795	NaN	NaN
735430	735614	735795	735979	736160

TFactors =

Columns 1 through 7

0	0.3913	1.3913	2.3913	3.3913	4.3913	5.3913
0	0.3913	1.3913	2.3913	3.3913	4.3913	5.3913

Columns 8 through 11

6.3913	7.3913	NaN	NaN
6.3913	7.3913	8.3913	9.3913

CFFlags =

0	3	3	3	3	3	13	3	4	NaN	NaN
0	3	3	3	3	3	13	3	13	3	4

CFPrincipal =

0	0	0	0	0	0	10	0	90	NaN	NaN
0	0	0	0	0	0	10	0	10	0	80

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

accfrac | cfdates | cftimes | cpncount | cpndaten | cpndatenq | cpdatep |
cpdatepq | cpndaysn | cpndaysp

cfconv

Cash flow convexity

Syntax

```
CFlowConvexity = cfconv(CashFlow, Yield)
```

Arguments

CashFlow	A vector of real numbers.
Yield	Periodic yield. A scalar. Enter as a decimal fraction.

Description

`CFlowConvexity = cfconv(CashFlow, Yield)` returns the convexity of a cash flow in periods.

Examples

Compute the Convexity of a Cash Flow

This example shows how to return the convexity of a cash flow, given a cash flow of nine payments of \$2.50 and a final payment \$102.50, with a periodic yield of 2.5%.

```
CashFlow = [2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 102.5];
```

```
Convex = cfconv(CashFlow, 0.025)
```

```
Convex =
```

```
90.4493
```

See Also

bndconvp | bndconvy | bnddurp | bnddury | cfdur

cfdates

Cash flow dates for fixed-income security

Syntax

```
CFlowDates = cfdates(Settle, Maturity, Period, Basis, EndMonthRule,
IssueDate, FirstCouponDate, LastCouponDate, StartDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA)

	<ul style="list-style-type: none"> • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.
StartDate	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify StartDate , the effective start date is the Settle date.

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `Maturity` contains `N` dates, then `Settle` must contain `N` dates or a single date.

Description

`CFlowDates = cfdates(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns a matrix of cash flow dates for a bond or set of bonds. `cfdates` determines all cash flow dates for a bond whether or not the coupon payment structure is normal or the first and/or last coupon period is long or short.

`CFlowDates` is an `N`-row matrix of serial date numbers, padded with `NaNs` as necessary to ensure that all rows have the same number of elements. Use the function `datestr` to convert serial date numbers to formatted date strings.

Note The cash flow flags for a portfolio of bonds were formerly available as the `cfdates` second output argument, `CFlowFlags`. You can now use `cfamounts` to get these flags. If you specify a `CFlowFlags` argument, `cfdates` displays a message directing you to use `cfamounts`.

Examples

```
CFlowDates = cfdates('14 Mar 1997', '30 Nov 1998', 2, 0, 1)
CFlowDates =
    729541    729724    729906    730089
datestr(CFlowDates)
```

```
ans =
31-May-1997
30-Nov-1997
31-May-1998
30-Nov-1998
```

Given three securities with different maturity dates and the same default arguments

```
Maturity = ['30-Sep-1997'; '31-Oct-1998'; '30-Nov-1998'];
CFlowDates = cfdates('14-Mar-1997', Maturity)
```

```
CFlowDates =
```

729480	729663	NaN	NaN
729510	729694	729875	730059
729541	729724	729906	730089

Look at the cash-flow dates for the last security.

```
datestr(CFlowDates(3,:))
```

```
ans =  
31-May-1997  
30-Nov-1997  
31-May-1998  
30-Nov-1998
```

See Also

accrfrac | cfamounts | cftimes | cpncount | cpndaten | cpndatenq | cpndatep
| cpndatepq | cpndaysn | cpndaysp | cpnpersz

cfdur

Cash-flow duration and modified duration

Syntax

```
[Duration, ModDuration] = cfdur(CashFlow, Yield)
```

Arguments

CashFlow	A vector or matrix of real numbers. When using a matrix, each column of the matrix is a separate CashFlow.
Yield	Periodic yield. A scalar or vector. Enter as a decimal fraction.

Description

`[Duration, ModDuration] = cfdur(CashFlow, Yield)` calculates the duration and modified duration of a cash flow in periods.

Examples

Compute the Duration and Modified Duration of a Cash Flow

This example shows how to calculate the duration and modified duration of a cash flow, given a cash flow of nine payments of \$2.50 and a final payment \$102.50, with a periodic yield of 2.5%.

```
CashFlow=[2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 102.5];
```

```
[Duration, ModDuration] = cfdur(CashFlow, 0.025)
```

```
Duration =
```

```
8.9709
```

ModDuration =

8.7521

See Also

bndconvp | bndconvy | bnddurp | bnddury | cfconv

cfplot

Visualize cash flows of financial instruments

Syntax

```
cfplot(CFlowDates,CFlowAmounts)
```

```
h = cfplot( ____,Name,Value)
```

```
[h,axes_handle] = cfplot( ____,Name,Value)
```

Description

`cfplot(CFlowDates,CFlowAmounts)` plots a cash flow diagram for the specified cash flow amounts (`CFlowAmounts`) and dates (`CFlowDates`). The length and orientation of each arrow correspond to the cash flow amount.

`cfplot(____,Name,Value)` plots a cash flow diagram for the specified cash flow amounts (`CFlowAmounts`), dates (`CFlowDates`), and optional name-value pair arguments.

`h = cfplot(____,Name,Value)` returns the handle to the line objects used in the cash flow diagram.

`[h,axes_handle] = cfplot(____,Name,Value)` returns the handles to the line objects and the axes using optional name-value pair arguments.

Examples

Plot Cash Flows

Define `CFlowAmounts` and `CFlowDates` using the `cfamounts` function.

```
CouponRate = [0.06; 0.05; 0.03];  
Settle = '03-Jun-1999';
```

```
Maturity = ['15-Aug-2000'; '15-Dec-2000'; '15-Jun-2000'];  
Period = [1; 2; 2]; Basis = [1; 0; 0];  
[CFlowAmounts, CFlowDates] = cfamounts(...  
CouponRate, Settle, Maturity, Period, Basis)
```

```
CFlowAmounts =
```

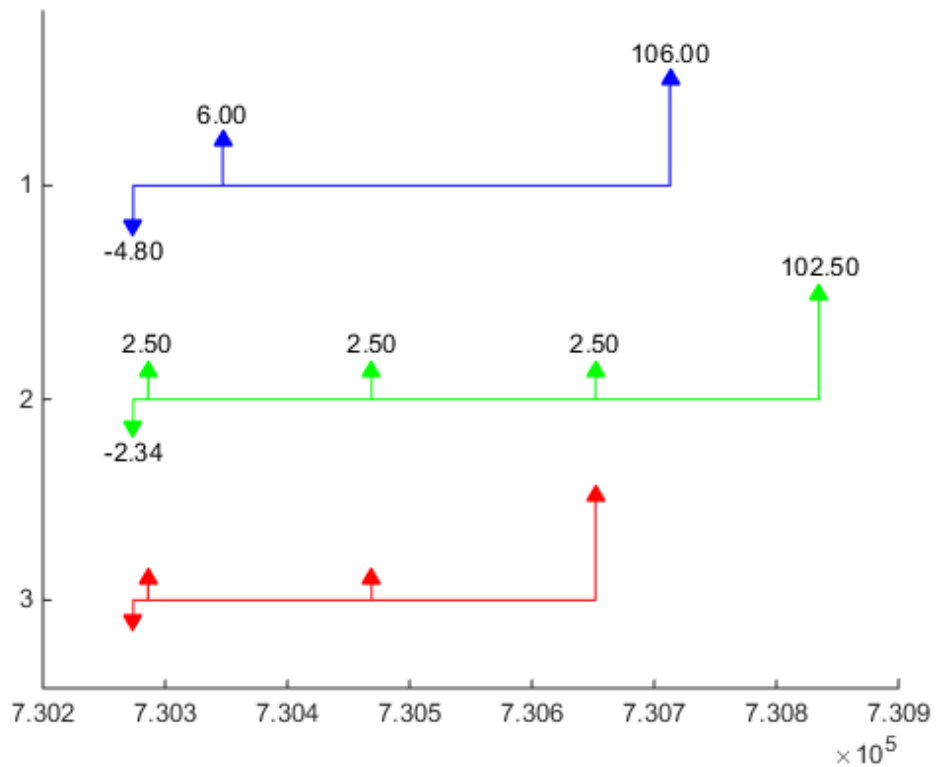
-4.8000	6.0000	106.0000	NaN	NaN
-2.3352	2.5000	2.5000	2.5000	102.5000
-1.4011	1.5000	1.5000	101.5000	NaN

```
CFlowDates =
```

730274	730347	730713	NaN	NaN
730274	730286	730469	730652	730835
730274	730286	730469	730652	NaN

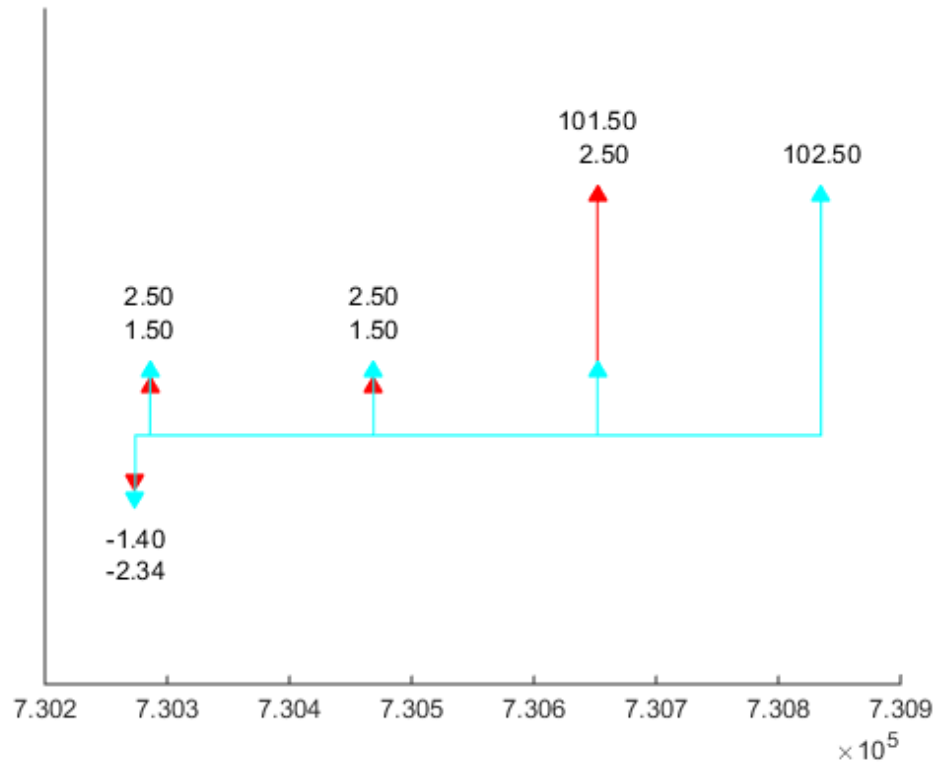
Plot all cash flows on the same axes, and label the first two.

```
cfplot(CFlowDates, CFlowAmounts, 'ShowAmnt', [1 2])
```



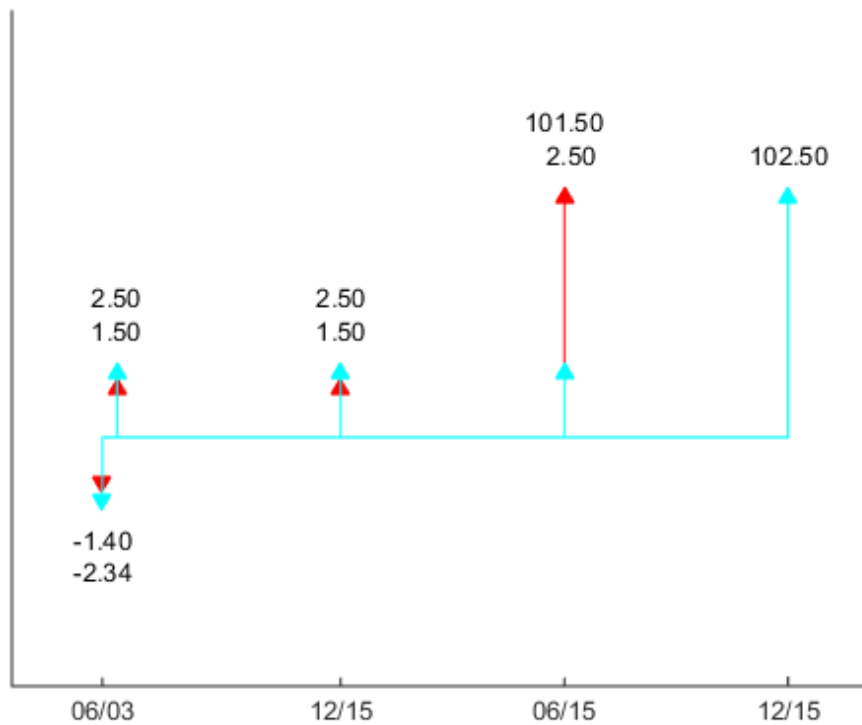
Group the second and third cash flows.

```
figure;  
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmt', 1);
```

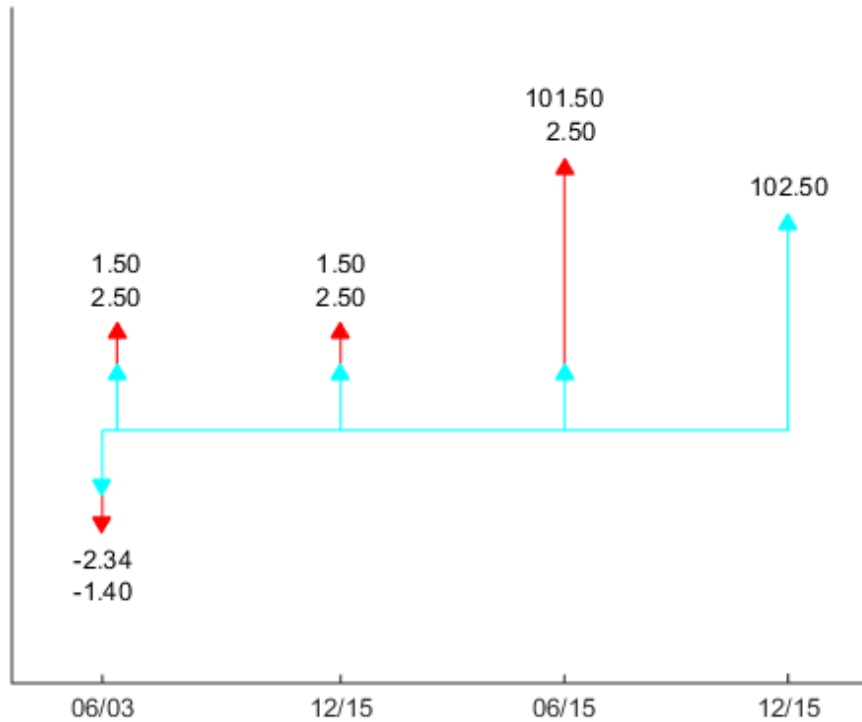
Format the date axis and place ticks on actual cash flow dates.

```
figure;
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmnt', 1, ...
'DateFormat', 6, 'DateSpacing', 100);
```



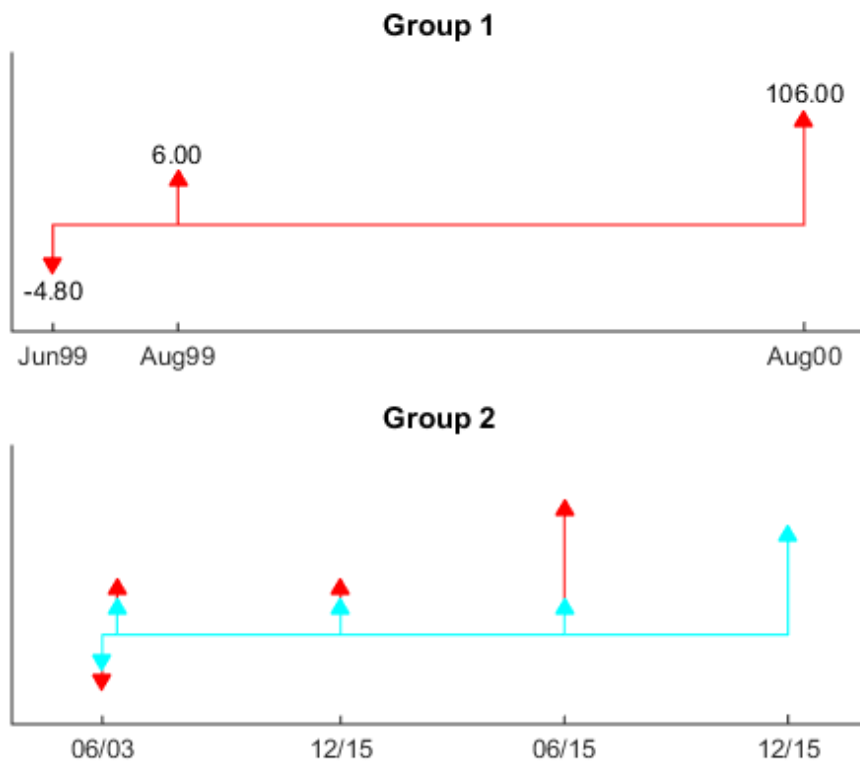
Stack the cash flow arrows occurring on the same dates.

```
figure;  
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmt', 1, ...  
'DateFormat', 6, 'DateSpacing', 100, 'Stacked', 1);
```



Form subplots of multiple groups and add titles using axes handles.

```
figure;
[h, axes_handle] = cfplot(CFlowDates, CFlowAmounts, ...
    'Groups', {[1] [2 3]}, 'ShowAmt', 1, 'Stacked', 2, ...
    'DateSpacing', [1 60 2 100], 'DateFormat', [1 12 2 6]);
title(axes_handle(1), 'Group 1', 'FontWeight', 'bold');
title(axes_handle(2), 'Group 2', 'FontWeight', 'bold');
```



Plot Cash Flows for Swap

Define the swap using the `swapbyzero` function.

```
Settle = datenum('08-Jun-2010');
RateSpec = intenvset('Rates', [.005 .0075 .01 .014 .02 .025 .03]', ...
    'StartDates',Settle, 'EndDates',{'08-Dec-2010','08-Jun-2011',...
    '08-Jun-2012','08-Jun-2013','08-Jun-2015','08-Jun-2017',...
    '08-Jun-2020'});
Maturity = datenum('15-Sep-2020');
LegRate = [.025 50];
LegType = [1 0]; % fixed/floating
LatestFloatingRate = .005;
[Price, SwapRate, AI, RecCF, RecCFDates, PayCF,PayCFDates] = ...
    swapbyzero(RateSpec, LegRate, Settle, Maturity, 'LegType',LegType,...
```

```
'LatestFloatingRate',LatestFloatingRate)
```

```
Price =
```

```
-6.7258
```

```
SwapRate =
```

```
NaN
```

```
AI =
```

```
1.4575
```

```
RecCF =
```

```
Columns 1 through 7
```

```
-1.8219    2.5000    2.5000    2.5000    2.5000    2.5000    2.5000
```

```
Columns 8 through 12
```

```
2.5000    2.5000    2.5000    2.5000    102.5000
```

```
RecCFDates =
```

```
Columns 1 through 6
```

```
734297    734396    734761    735127    735492    735857
```

```
Columns 7 through 12
```

```
736222    736588    736953    737318    737683    738049
```

```
PayCF =
```

```
Columns 1 through 7
```

```
-0.3644    0.5000    1.4048    1.9823    2.8436    3.2842    3.8218
```

Columns 8 through 12

4.1733 4.5164 4.4666 4.8068 104.6743

PayCFDates =

Columns 1 through 6

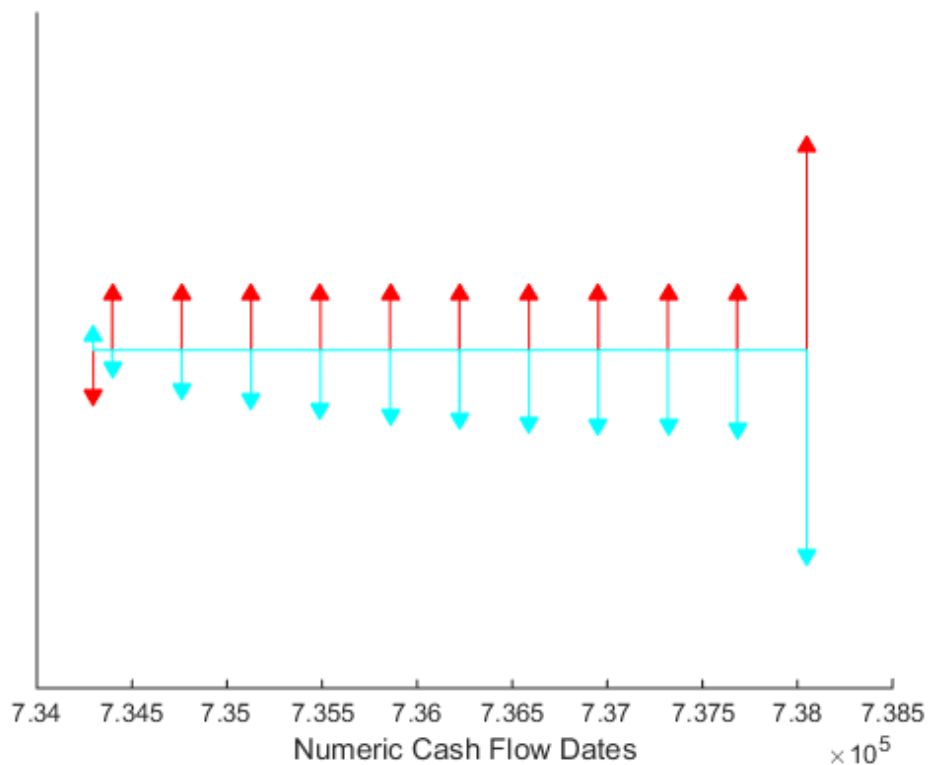
734297 734396 734761 735127 735492 735857

Columns 7 through 12

736222 736588 736953 737318 737683 738049

Define `CFlowDates` and `CFlowAmounts` for the swap and generate a cash flow plot using `cfplot`.

```
CFlowDates = [PayCFDates;RecCFDates];  
CFlowAmounts = [-PayCF;RecCF];  
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[1 2]});  
xlabel('Numeric Cash Flow Dates');
```



Input Arguments

CFlowDates — Matrix of serial date numbers for cash flows

vector

Matrix of serial date numbers for cash flows, specified as a NINST-by- (Number of cash flows) matrix of cash flow dates in date numbers, with empty entries padded with NaNs.

Each row of the `CFlowDates` matrix represents an instrument so that `CFlowDates(k, :)` is the vector of cash flow dates for the k th instrument. Rows are padded with trailing NaNs if the number of cash flows is not the same for all instruments.

`cfamounts` can be used to generate `CFlowDates`.

Data Types: `double`

CFlowAmounts — Matrix of cash flow amounts

vector

Matrix of cash flow amounts, specified as a NINST-by-(Number of cash flows) matrix of cash flow amounts, with empty entries padded with NaNs. The `CFlowAmounts` matrix must be the same size as `CFlowDates`.

`cfamounts` can be used to generate `CFlowAmounts`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`,`Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`,`Value1`, ..., `NameN`,`ValueN`.

Example: `cfplot(CFlowDates,CFlowAmounts,'Groups',{[2 3]},'ShowAmt',1,'DateFormat',6,'DateSpacing',100)`

'Groups' — Group cash flows

'off' (default) | string with value 'off' or 'individual' | cell array of vectors

Group cash flows specified using the following values:

- 'off' — Show all instruments in one set of axes, arranged from top to bottom.
- 'individual' — Generate subplots and plot each instrument in its own axis.
- GRP — Cell array of instrument groups, {`Group1`, `Group2`, ... }. This generates subplots and plots each group in each axis. When specifying {`Group1`, `Group2`, ... }, each `Group` must be mutually exclusive vectors of `INSTIndex`. Unspecified instruments are not shown in the grouped plot.

Data Types: `double`

'Stacked' — Stack arrows if cash flows are in same direction on same day

Ignored when 'Groups' is 'off', otherwise 'off' (default) | string with value 'off', 'all', or 'GRPIndex'

Stack arrows if the cash flows are in the same direction on the same day specified using the following values:

- 'off' — For all groups, all arrows originate from the horizontal line.
- 'all' — For all groups, arrows are stacked if the cash flows are in the same direction on the same day.
- 'GRPIndex' — For specified groups, arrows are stacked if the cash flows are in the same direction on the same day.

Data Types: double

'ShowAmt' — Show amount on arrows

'off' (default) | string with value 'off' or 'individual' | cell array of vectors |

Show amount on the arrows specified using the following values:

- 'off' — Hide cash flow amounts on arrows.
- 'all' — Show cash flow amounts on arrows.
- [INSTIndex or GRPIndex] — Show cash flow amounts for the specified vector of instruments (when 'Groups' is 'off') or groups.

Data Types: double

'DateSpacing' — Control for date axis tick spacing

'off' (default) | string with value 'off' or string TickDateSpace | numeric value for TickDateSpace

Control for data spacing specified by the following values:

- 'off' — The date axis ticks are spaced regularly.
- TickDateSpace — The date axis ticks are placed on actual cash flow dates. The ticks skip some cash flows if they are less than TickDateSpace apart.

Data Types: double

'DateFormat' — Date format

'off' (default) | string with value 'off' or string DateFormNum | numeric value for DateFormNum

Date format is specified by the following values:

- 'off' — The date axis tick labels will be in date numbers.
- `DateFormNum` — The date format number (2 = 'mm/dd/yy', 6 = 'mm/dd', and 10 = 'yyyy'). Additional values for `DateFormNum` are as follows:

DateFormNum	Example
2	03/01/00
3	Mar
5	03
6	03/01
7	01
8	Wed
9	W
10	2000
11	00
12	Mar00
17	Q1-00
18	Q1
19	01/03
20	01/03/00
27	Q1-2000
28	Mar2000
29	2000-03-01

Data Types: double

Output Arguments

h — Handles to line objects

vector

Handles to line objects, returned as a NINST-by-3 matrix of handles to line objects, containing [hLines, hUArrowHead, hDArrowHead] where:

- `hLines` — Horizontal and vertical lines used in the cash flow diagram
- `hUArrowHead` — "Up" arrowheads
- `hDArrowHead` — "Down" arrowheads

axes_handle — Handles to axes for plot or subplots

vector

Handles to axes for the plot or subplots, returned as a (Number of axes)-by-1 vector of handles to axes.

See Also

`cfamounts` | `cfdates` | `swapbyzero`

cfport

Portfolio form of cash flow amounts

Syntax

```
[CFBondDate, AllDates, AllTF, IndByBond] = cfport(CFlowAmounts,
CFlowDates, TFactors)
```

Arguments

CFlowAmounts	Number of bonds (NUMBONDS) by number of cash flows (NUMCFS) matrix with entries listing cash flow amounts corresponding to each date in CFlowDates.
CFlowDates	NUMBONDS-by-NUMCFS matrix with rows listing cash flow dates for each bond and padded with NaNs.
TFactors	(Optional) NUMBONDS-by-NUMCFS matrix with entries listing the time between settlement and the cash flow date measured in semiannual coupon periods.

Description

[CFBondDate, AllDates, AllTF, IndByBond] = cfport(CFlowAmounts, CFlowDates, TFactors) computes a vector of all cash flow dates of a bond portfolio, and a matrix mapping the cash flows of each bond to those dates. Use the matrix for pricing the bonds against a curve of discount factors.

CFBondDate is a NUMBONDS by number of dates (NUMDATES) matrix of cash flows indexed by bond and by date in AllDates. Each row contains a bond's cash flow values at the indices corresponding to entries in AllDates. Other indices in the row contain zeros.

AllDates is a NUMDATES-by-1 list of all dates that have any cash flow from the bond portfolio.

AllTF is a NUMDATES-by-1 list of time factors corresponding to the dates in AllDates. If TFactors is not entered, AllTF contains the number of days from the first date in AllDates.

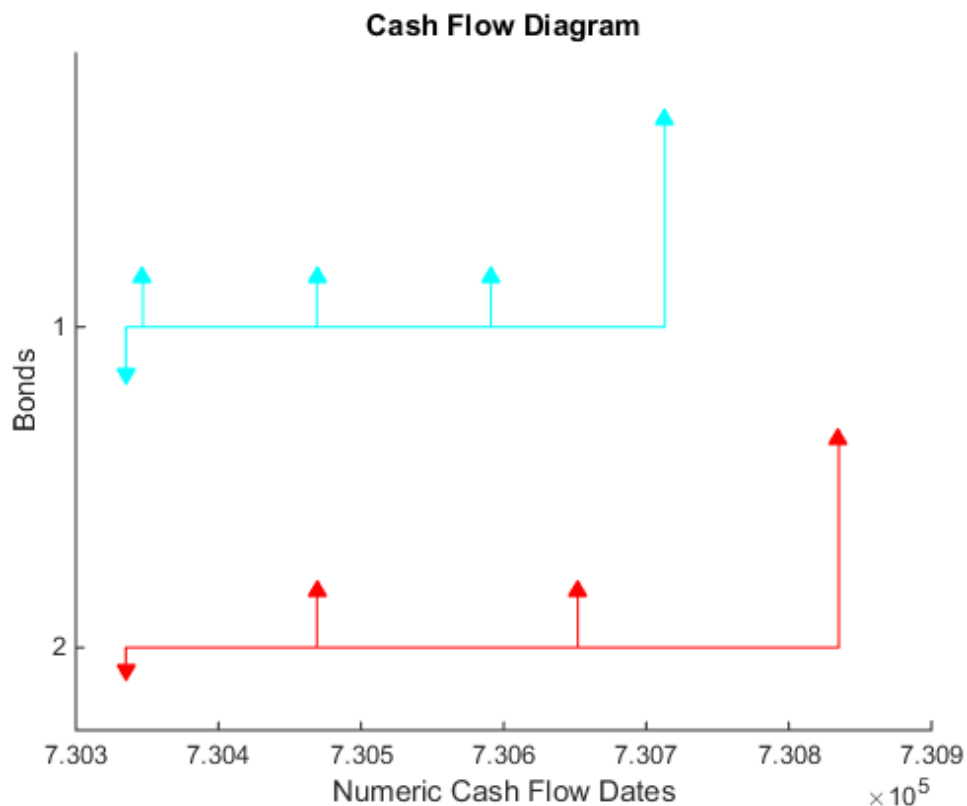
IndByBond is a NUMBONDS-by-NUMCFS matrix of indices. The i th row contains a list of indices into AllDates where the i th bond has cash flows. Since some bonds have more cash flows than others, the matrix is padded with NaNs.

Examples

Calculate the Cash Flow Amounts, Cash Flow Dates, and Time Factors for Each of Two Bonds

Use the function `cfamounts` to calculate the cash flow amounts, cash flow dates, and time factors for each of two bonds. Then use the function `cfplot` to plot the cash flow diagram.

```
Settle = '03-Aug-1999';
Maturity = ['15-Aug-2000'; '15-Dec-2000'];
CouponRate= [0.06; 0.05];
Period = [3;2];
Basis = [1;0];
[CFlowAmounts, CFlowDates, TFactors] = cfamounts(CouponRate,...
Settle, Maturity, Period, Basis);
cfplot(CFlowDates,CFlowAmounts)
xlabel('Numeric Cash Flow Dates')
ylabel('Bonds')
title('Cash Flow Diagram')
```



Call the function `cfport` to map the cash flow amounts to the cash flow dates. Each row in the resultant `CFBondDate` matrix represents a bond. Each column represents a date on which one or more of the bonds has a cash flow. A 0 means the bond did not have a cash flow on that date. The dates associated with the columns are listed in `AllDates`. For example, the first bond had a cash flow of 2.000 on 730347. The second bond had no cash flow on this date. For each bond, `IndByBond` indicates the columns of `CFBondDate`, or dates in `AllDates`, for which a bond has a cash flow.

```
[CFBondDate, AllDates, AllTF, IndByBond] = ...
cfport(CFlowAmounts, CFlowDates, TFactors)
```

```
CFBondDate =
```

```
-1.8000  2.0000  2.0000  2.0000  0 102.0000  0
-0.6694  0  2.5000  0  2.5000  0 102.5000
```

```
AllDates =
```

```
730335
730347
730469
730591
730652
730713
730835
```

```
AllTF =
```

```
0
0.0663
0.7322
1.3989
1.7322
2.0663
2.7322
```

```
IndByBond =
```

```
1 2 3 4 6
1 3 5 7 NaN
```

See Also

cfamounts | cfplot

cfprice

Compute price for cash flow given yield to maturity

Syntax

```
Price = cfprice(CFAmounts, CFDates, Yield, Settle)
Price = cfprice(CFAmounts, CFDates, Yield, Settle,
Name, Value)
```

Description

`Price = cfprice(CFAmounts, CFDates, Yield, Settle)` computes a price given yield for a cash flow.

`Price = cfprice(CFAmounts, CFDates, Yield, Settle, Name, Value)` computes a price for a cash flow given yield to maturity with additional options specified by one or more `Name, Value` pair arguments.

Input Arguments

CFlowAmounts

NINST-by-MOSTCFS matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.

CFlowDates

NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the serial date of the corresponding cash flow in `CFlowAmounts`.

Yield

NINST-by-1 vector of yields.

Settle

Settlement date is a serial date number or date string. Settlement date is the date on which the cash flows are priced.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Note: Any optional input of size N-by-1 is also acceptable as an array of size 1-by-N, or as a single value applicable to all contracts. Single values are internally expanded to an array of size N-by-1.

'Basis'

N-by-1 vector of day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see “”.

Default: 0 (actual/actual)

'CompFreq'

Compounding frequency. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

Default: actual

Output Arguments

Price

Price of cash flows.

Examples

Compute the Price for a Cash Flow Given Yield to Maturity

Use `cfprice` to compute: price for a cash flow given yield to maturity.

Define data for the yield curve.

```
Settle = datenum('01-Jul-2003');  
Yield = .05;  
CFAmounts = [30;40;30];  
CFDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'});
```

Compute the Price.

```
Price = cfprice(CFAmounts, CFDates, Yield, Settle)
```

```
Price =
```

```
28.4999  
36.1689  
25.8195
```

See Also

| cfbyzero | cfyield | cfsread

cfspread

Compute spread over yield curve for cash flow

Syntax

```
Spread = cfspread(RateSpec, Price, CFAmounts,  
CFDates, Settle)  
Spread = cfspread(RateSpec, Price, CFAmounts,  
CFDates, Settle, Name, Value)
```

Description

`Spread = cfspread(RateSpec, Price, CFAmounts, CFDates, Settle)` computes spread over a yield curve for a cash flow.

`Spread = cfspread(RateSpec, Price, CFAmounts, CFDates, Settle, Name, Value)` computes spread over a yield curve for a cash flow with additional options specified by one or more `Name, Value` pair arguments.

Input Arguments

RateSpec

Interest-rate specification for the initial risk free rate curve. See `intenvset` for information on declaring an interest-rate variable.

Price

Price of cash flows.

CFflowAmounts

NINST-by-MOSTCFS matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.

CFlowDates

NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the serial date of the corresponding cash flow in CFlowAmounts.

Settle

Settlement date is a serial date number or date string. Settlement date is the date on which the cash flows are priced.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Note: Any optional input of size N-by-1 is also acceptable as an array of size 1-by-N, or as a single value applicable to all contracts. Single values are internally expanded to an array of size N-by-1.

'Basis'

N-by-1 vector of day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see “”.

Default: 0 (actual/actual)

'CompFreq'

Compounding frequency. By default, SIA bases (0-7) and BUS/252 use a semi-annual compounding convention and ISMA bases (8-12) use an annual compounding convention.

Default: actual

Output Arguments

Spread

Spread of cash flows over a zero curve.

Examples

Compute Spread Over a Yield Curve for a Cash Flow

Use `cfsread` to compute the spread over a yield curve for a cash flow.

Define data for the yield curve.

```
Settle = datenum('01-Jul-2003');  
CurveDates = daysadd(Settle,360* [.25 .5 1 2 3 5 7 10 20],1);  
ZeroRates = [.0089 .0096 .0107 .0130 .0166 .0248 .0306 .0356 .0454]';
```

Compute the `RateSpec`.

```
RateSpec = intenvset('StartDates', Settle, 'EndDates', CurveDates, ...  
'Rates', ZeroRates)
```

```
RateSpec =
```

```
    FinObj: 'RateSpec'  
    Compounding: 2  
        Disc: [9x1 double]  
        Rates: [9x1 double]  
        EndTimes: [9x1 double]  
        StartTimes: [9x1 double]  
        EndDates: [9x1 double]  
        StartDates: 731763  
    ValuationDate: 731763  
        Basis: 0  
    EndMonthRule: 1
```

Compute the spread.

```
Price = 98;  
CFAmounts = [30;40;30];  
CFDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'});  
  
Spread = cfspread(RateSpec, Price, CFAmounts, CFDates, Settle)
```

Spread =

```
    1.0e+03 *  
  
    -8.7956  
    -4.0774  
    -3.7073
```

See Also

| cfbyzero | cfyield | cfprice

cfyield

Compute yield to maturity for cash flow given price

Syntax

```
Yield = cfyield(CFAmounts, CFDates, Price, Settle)
Yield = cfyield(CFAmounts, CFDates, Price, Settle,
Name, Value)
```

Description

`Yield = cfyield(CFAmounts, CFDates, Price, Settle)` computes yield to maturity for a cash flow given price.

`Yield = cfyield(CFAmounts, CFDates, Price, Settle, Name, Value)` computes yield to maturity for a cash flow given price with additional options specified by one or more `Name, Value` pair arguments.

Input Arguments

CFlowAmounts

NINST-by-MOSTCFS matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.

CFlowDates

NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the serial date of the corresponding cash flow in `CFlowAmounts`.

Price

Price.

Settle

Settlement date is a serial date number or date string. Settlement date is the date on which the cash flows are priced.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Note: Any optional input of size N-by-1 is also acceptable as an array of size 1-by-N, or as a single value applicable to all contracts. Single values are internally expanded to an array of size N-by-1.

'Basis'

N-by-1 vector of day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0 (actual/actual)

'CompFreq'

Compounding frequency. By default, SIA bases (0-7) and BUS/252 use a semi-annual compounding convention and ISMA bases (8-12) use an annual compounding convention.

Default: actual

Output Arguments

Yield

Yield for cash flows.

Examples

Compute the Yield to Maturity for a Cash Flow When Given a Price

Use `cfyield` to compute yield to maturity for a cash flow when given a price.

Define data for the yield curve and price.

```
Settle = datenum('01-Jul-2003');  
Price = 98;  
CFlowAmounts = [30 40 30];  
CFlowDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'})';
```

Compute the Yield.

```
Yield = cfyield(CFlowAmounts, CFlowDates, Price, Settle)
```

```
Yield =
```

```
0.0099
```

See Also

| `cfbyzero` | `cfprice` | `cfspread`

cftimes

Time factors corresponding to bond cash flow dates

Syntax

```
[TFactors] = cftimes(Settle, Maturity)
[TFactors] = cftimes(Settle, Maturity,
Period, Basis, EndMonthRule,
IssueDate, FirstCouponDate, LastCouponDate, StartDate)
[TFactors] = cftimes(Settle, Maturity,
'ParameterName', 'ParameterValue ...)
```

Description

[TFactors] = cftimes(Settle, Maturity) determines the time factors corresponding to the cash flows of a bond or set of bonds.

[TFactors] = cftimes(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate) determines the time factors corresponding to the cash flows of a bond or set of bonds, including optional inputs.

[TFactors] = cftimes(Settle, Maturity, 'ParameterName', 'ParameterValue ...) accepts optional inputs as one or more comma-separated parameter/value pairs. 'ParameterName' is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to 'ParameterName'. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

Settle

Settlement date. A vector of serial date numbers or date strings. **Settle** must be earlier than **Maturity**.

Maturity

Maturity date. A vector of serial date numbers or date strings.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

Period

Coupons per year of the bond. A vector of integers. Values are 0, 1, 2, 3, 4, 6, and 12.

Default: 2

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)
- 11 = 30/360E (ISMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

EndMonthRule

End-of-month rule. A vector. This rule applies only when **Maturity** is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. 1 = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: 1

IssueDate

Issue date for a bond.

FirstCouponDate

Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When **FirstCouponDate** and **LastCouponDate** are both specified, **FirstCouponDate** takes precedence in determining the coupon payment structure.

Default: If you do not specify a **FirstCouponDate**, the cash flow payment dates are determined from other inputs.

LastCouponDate

Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified **FirstCouponDate**, a specified **LastCouponDate** determines the coupon structure of the bond. The coupon structure of a bond is truncated at the **LastCouponDate**, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a **LastCouponDate**, the cash flow payment dates are determined from other inputs.

StartDate

Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify **StartDate**, the effective start date is the **Settle** date.

Parameter–Value Pairs

Enter the following inputs only as parameter/value pairs.

CompoundingFrequency

Compounding frequency for yield calculation. By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ISMA bases (8-12) use an annual compounding convention.

DiscountBasis

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ISMA day counts and BUS/252, the specified bases are used.

Output Arguments

TFactors

TFactors has NUMBONDS rows and the number of columns is determined by the maximum number of cash flow payment dates required to hold the bond portfolio. NaNs are padded for bonds which have less than the maximum number of cash flow payment dates.

Definitions

`cftimes` computes the time factor of a cash flow, which is the difference between the settlement date and the cash flow date, in units of semiannual coupon periods. In computing time factors, use SIA actual/actual day count conventions for all time factor calculations.

Examples

Compute the Time Factor of a Cash Flow

This example shows how to calculate a cash flow time factor.

```
Settle = '15-Mar-1997';  
Maturity = '01-Sep-1999';  
Period = 2;  
TFactors = cftimes(Settle, Maturity, Period)
```

TFactors =

0.9239 1.9239 2.9239 3.9239 4.9239

References

Krgin, Dragomir, *Handbook of Global Fixed Income Calculations*, John Wiley & Sons, 2002.

Mayle, Jan, “Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures”, *SIA*, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson, *Money Market and Bond Calculations*, McGraw-Hill, 1996.

See Also

accrfrac | cfdates | cfamounts | cpncount | cpndaten | cpndatenq | cpndatep
| cpndatepq | cpndaysn | cpndaysp | date2time

chaikosc

Chaikin oscillator

Syntax

```
chosc = chaikosc(highp, lowp, closep, tvolume)
```

```
chosc = chaikosc([highp lowp closep tvolume])
```

```
choscts = chaikosc(tsobj)
```

```
choscts = chaikosc(tsobj, ParameterName, ParameterValue, ... )
```

Arguments

highp	High price (vector)
lowp	Low price (vector)
closep	Closing price (vector)
tvolume	Volume traded (vector)
tsobj	Financial time series object

Description

The Chaikin oscillator is calculated by subtracting the 10-period exponential moving average of the Accumulation/Distribution (A/D) line from the three-period exponential moving average of the A/D line.

`chosc = chaikosc(highp, lowp, closep, tvolume)` calculates the Chaikin oscillator (vector), `chosc`, for the set of stock price and volume traded data (`tvolume`). The required inputs are the prices for the high (`highp`), low (`lowp`), and closing (`closep`) prices and the volume traded data (`tvolume`).

`chosc = chaikosc([highp lowp closep tvolume])` accepts a four-column matrix as input.

`choscts = chaikosc(tsoobj)` calculates the Chaikin Oscillator, `choscts`, from the data contained in the financial time series object `tsoobj`. `tsoobj` must at least contain data series with names `High`, `Low`, `Close`, and `Volume`. These series must represent the high, low, and closing prices, plus the volume traded. `choscts` is a financial time series object with the same dates as `tsoobj` but only one series named `ChaikOsc`.

`choscts = chaikosc(tsoobj, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

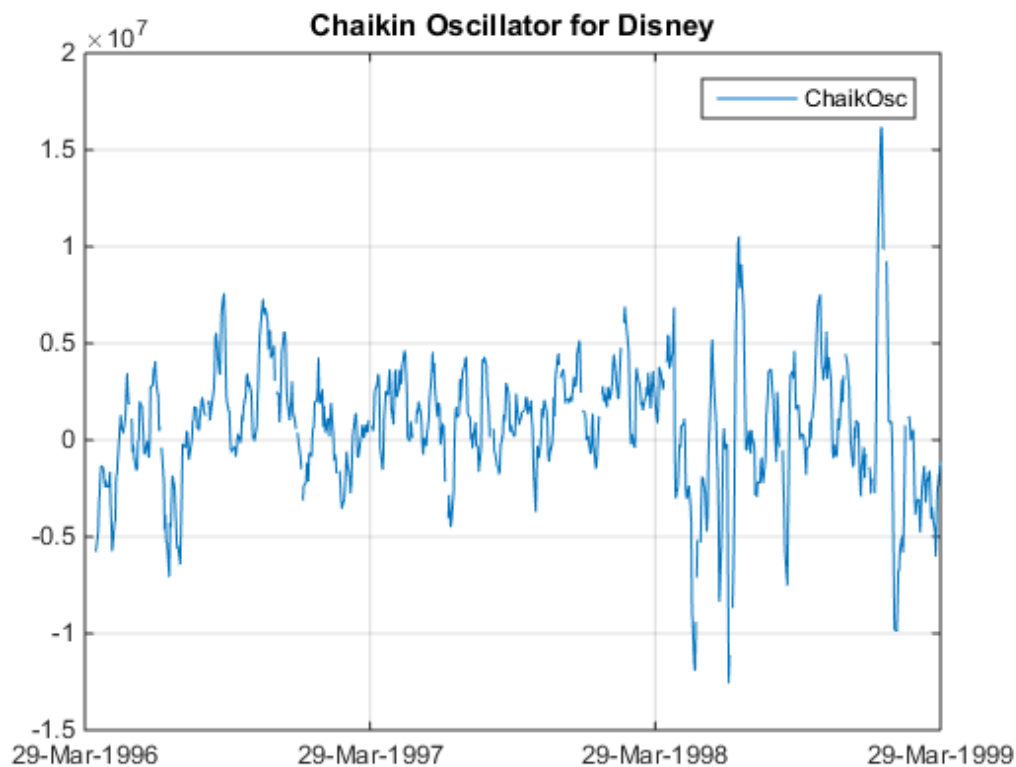
Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Chaikin Oscillator

This example shows how to compute the Chaikin oscillator for Disney stock and plot the results.

```
load disney.mat
dis_CHAIKosc = chaikosc(dis);
plot(dis_CHAIKosc)
title('Chaikin Oscillator for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 91-94.

See Also

adline

chaikvolat

Chaikin volatility

Syntax

```
chvol = chaikvolat(highp, lowp)
```

```
chvol = chaikvolat([highp lowp])
```

```
chvol = chaikvolat(high, lowp, nperdiff, manper)
```

```
chvol = chaikvolat([high lowp], nperdiff, manper)
```

```
chvts = chaikvolat(tsobj)
```

```
chvts = chaikvolat(tsobj, nperdiff, manper, ParameterName,  
ParameterValue, ...)
```

Arguments

highp	High price (vector).
lowp	Low price (vector).
nperdiff	Period difference (vector). Default = 10.
manper	Length of exponential moving average in periods (vector). Default = 10.
tsobj	Financial time series object.

Description

`chvol = chaikvolat(highp, lowp)` calculates the Chaikin volatility from the series of stock prices, `highp` and `lowp`. The vector `chvol` contains the Chaikin volatility values, calculated on a 10-period exponential moving average and 10-period difference.

`chvol = chaikvolat([highp lowp])` accepts a two-column matrix as the input.

`chvol = chaikvolat(high, lowp, nperdiff, manper)` manually sets the period difference `nperdiff` and the length of the exponential moving average `manper` in periods.

`chvol = chaikvolat([high lowp], nperdiff, manper)` accepts a two-column matrix as the first input.

`chvts = chaikvolat(tsobj)` calculates the Chaikin volatility from the financial time series object `tsobj`. The object must contain at least two series named `High` and `Low`, representing the high and low prices per period. `chvts` is a financial time series object containing the Chaikin volatility values, based on a 10-period exponential moving average and 10-period difference. `chvts` has the same dates as `tsobj` and a series called `ChaikVol`.

`chvts = chaikvolat(tsobj, nperdiff, manper, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name

Parameter values are the strings that represent the valid parameter names.

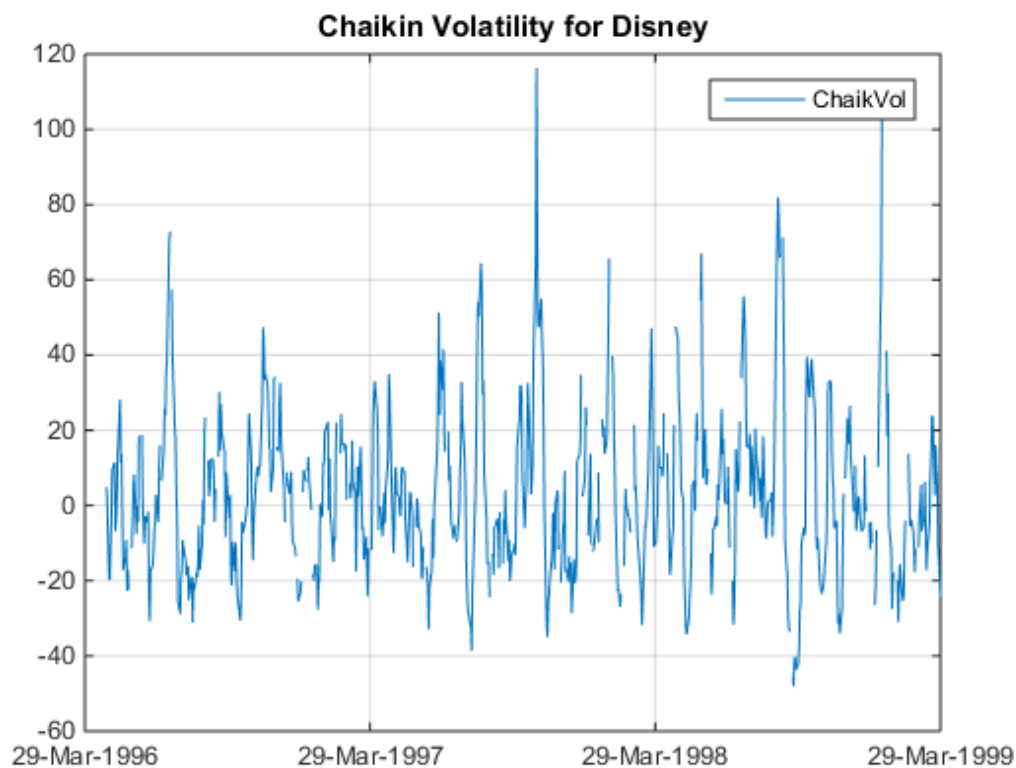
`nperdiff`, the period difference, and `manper`, the length of the exponential moving average in periods, can also be set with this form of `chaikvolat`.

Examples

Compute the Chaikin Volatility

This example shows how to compute the Chaikin volatility for Disney stock and plot the results.

```
load disney.mat
dis_CHAIKvol = chaikvolat(dis);
plot(dis_CHAIKvol)
title('Chaikin Volatility for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second Edition, McGraw-Hill, 1995, pp. 304-305.

See Also

chaikosc

chartfts

Interactive display

Syntax

```
chartfts(tsobj)
```

Description

`chartfts(tsobj)` produces a figure window that contains one or more plots. You can use the mouse to observe the data at a particular time point of the plot.

Examples

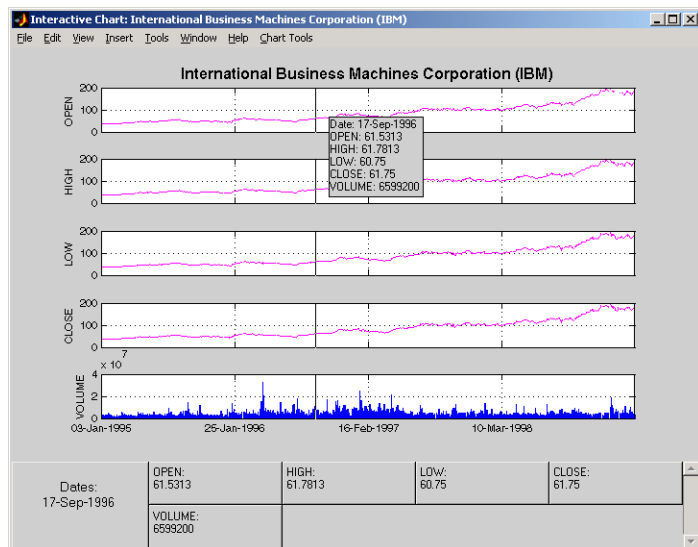
Create a financial time series object from the supplied data file `ibm9599.dat`:

```
ibmfts = ascii2fts('ibm9599.dat', 1, 3, 2);
```

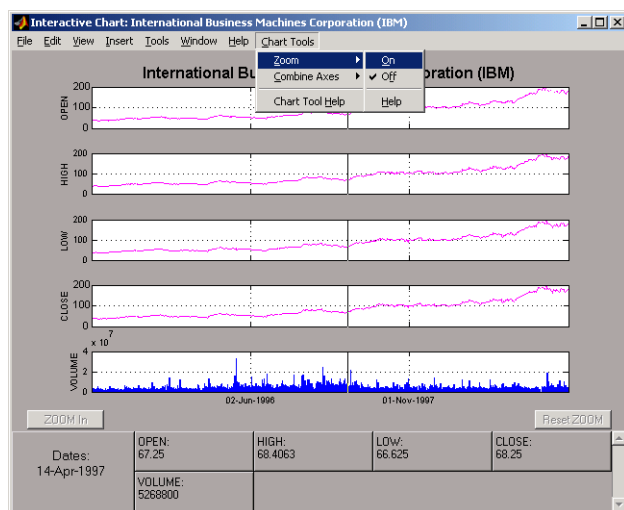
Chart the financial time series object `ibmfts`:

```
chartfts(ibmfts)
```

With the **Zoom** feature set `off`, a mouse click on the indicator line displays object data in a pop-up box.



With the **Zoom** feature set on, mouse clicks indicate the area of the chart to zoom.



You can find a tutorial on using chartfts in “Visualizing Financial Time Series Objects” on page 11-16. See “Zoom Tool” on page 11-19 for details on performing the zoom.

Also see “Combine Axes Tool” on page 11-22 for information about combining axes for specified plots.

See Also

candle | highlow | plot

checkFeasibility

Class: Portfolio

Check feasibility of input portfolios against Portfolio object

Syntax

```
status = checkFeasibility(obj,pwgt)
```

Description

`status = checkFeasibility(obj,pwgt)` checks the feasibility of input portfolios against a Portfolio object.

Tips

- You can also use dot notation to check the feasibility of input portfolios against a Portfolio object.

```
status = obj.checkFeasibility(pwgt);
```
- The constraint tolerance to assess whether a constraint is satisfied is obtained from the hidden property `obj.defaultTolCon`.

Input Arguments

obj

Portfolio object [Portfolio].

pwgt

Portfolios to be checked [NumAssets-by-NumPorts matrix].

Output Arguments

status

Row vector of NumPorts indicators that are `true` if portfolio is feasible and `false` otherwise.

Note: By definition, any portfolio set must be nonempty and bounded. If the set is empty, no portfolios can be feasible. Use `estimateBounds` to test for nonempty and bounded sets.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Determine if the Portfolio Is Feasible

Given portfolio `p`, determine if `p` is feasible.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
```

```
checkFeasibility(p, pwgt)
```

```
ans =
```

```
    1    1    1    1    1    1    1    1    1    1
```

- “Validate the Portfolio Problem” on page 4-79

See Also

estimateBounds

More About

- “Portfolio Problem Specification” on page 4-2

checkFeasibility

Class: PortfolioCVaR

Check feasibility of input portfolios against PortfolioCVaR object

Syntax

```
status = checkFeasibility(obj,pwgt)
```

Description

`status = checkFeasibility(obj,pwgt)` checks the feasibility of input portfolios against a PortfolioCVaR object.

Tips

- You can also use dot notation to check the feasibility of input portfolios against a PortfolioCVaR object.

```
status = obj.checkFeasibility(pwgt);
```

- The constraint tolerance to assess whether a constraint is satisfied is obtained from the hidden property `obj.defaultTolCon`.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

pwgt

Portfolios to be checked [NumAssets-by-NumPorts matrix].

Output Arguments

status

Row vector of NumPorts indicators that are `true` if the CVaR portfolio object is feasible and `false` otherwise.

Note: By definition, any portfolio set must be nonempty and bounded. If the set is empty, no portfolios can be feasible. Use `estimateBounds` to test for nonempty and bounded sets.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Determine if the Portfolio Is Feasible

Given portfolio `p`, determine if `p` is feasible.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
```

```
p = setScenarios(p, AssetScenarios);  
p = setDefaultConstraints(p);  
p = setProbabilityLevel(p, 0.95);
```

```
pwgt = estimateFrontier(p);
```

```
checkFeasibility(p, pwgt)
```

```
ans =
```

```
1 1 1 1 1 1 1 1 1 1
```

- “Validate the CVaR Portfolio Problem” on page 5-78

See Also

estimateBounds

More About

- “Portfolio Optimization Theory” on page 5-2

checkFeasibility

Class: PortfolioMAD

Check feasibility of input portfolios against PortfolioMAD object

Syntax

```
status = checkFeasibility(obj,pwgt)
```

Description

`status = checkFeasibility(obj,pwgt)` checks the feasibility of input portfolios against a PortfolioMAD object.

Tips

- You can also use dot notation to check the feasibility of input portfolios against a PortfolioMAD object.

```
status = obj.checkFeasibility(pwgt);
```
- The constraint tolerance to assess whether a constraint is satisfied is obtained from the hidden property `obj.defaultTolCon`.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

pwgt

Portfolios to be checked [NumAssets-by-NumPorts matrix].

Output Arguments

status

Row vector of NumPorts indicators that are `true` if the MAD portfolio object is feasible and `false` otherwise.

Note: By definition, any portfolio set must be nonempty and bounded. If the set is empty, no portfolios can be feasible. Use `estimateBounds` to test for nonempty and bounded sets.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Determine if the Portfolio Is Feasible

Given portfolio `p`, determine if `p` is feasible.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
```



```
p = setScenarios(p, AssetScenarios);  
p = setDefaultConstraints(p);
```

```
pwgt = estimateFrontier(p);
```

```
checkFeasibility(p, pwgt)
```

```
ans =
```

```
    1    1    1    1    1    1    1    1    1    1
```

- “Validate the MAD Portfolio Problem” on page 6-78

See Also

estimateBounds

More About

- “Portfolio Optimization Theory” on page 6-2

chfield

Change data series name

Syntax

```
newfts = chfield(oldfts, oldname, newname)
```

Arguments

oldfts	Name of an existing financial time series object.
oldname	Name of the existing component in oldfts. A MATLAB string or column cell array.
newname	New name for the component in oldfts. A MATLAB string or column cell array.

Description

`newfts = chfield(oldfts, oldname, newname)` changes the name of the financial time series object component from `oldname` to `newname`.

Set `newfts = oldfts` to change the name of an existing component without changing the name of the financial time series object.

To change the names of several components at once, specify the series of old and new component names in corresponding column cell arrays.

You cannot change the names of the object components `desc`, `freq`, and `dates`.

See Also

`fieldnames` | `isfield` | `rmfield`

convert2sur

Convert multivariate normal regression model to seemingly unrelated regression (SUR) model

Syntax

DesignSUR = convert2sur(Design, Group)

Arguments

Design	<p>A matrix or a cell array that depends on the number of data series <code>NUMSERIES</code>.</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>convert2sur</code> returns the <code>Design</code> matrix. • If <code>NUMSERIES > 1</code>, <code>Design</code> is a cell array with <code>NUMSAMPLES</code> cells, where each cell contains a <code>NUMSERIES</code>-by-<code>NUMPARAMS</code> matrix of known values.
Group	<p>Contains information about how data series are to be grouped, with separate parameters for each group. Specify groups either by series or by groups:</p> <ul style="list-style-type: none"> • To identify groups by series, construct an index vector that has <code>NUMSERIES</code> elements. Element $i = 1, \dots, \text{NUMSERIES}$ in the vector, and has the index $j = 1, \dots, \text{NUMGROUPS}$ of the group in which series i is a member. • To identify groups by groups, construct a cell array with <code>NUMGROUPS</code> elements. Each cell contains a vector with the indexes of the series that populate a given group. <p>In either case, the number of series is <code>NUMSERIES</code> and the number of groups is <code>NUMGROUPS</code>, with $1 \leq \text{NUMGROUPS} \leq \text{NUMSERIES}$.</p>

Description

`DesignSUR = convert2sur(Design, Group)` converts a multivariate normal regression model into a seemingly unrelated regression model with a specified grouping of the data series. `DesignSUR` is either a matrix or a cell array that depends on the value of `NUMSERIES`:

- If `NUMSERIES = 1`, `DesignSUR = Design`, which is a `NUMSAMPLES`-by-`NUMPARAMS` matrix.
- If `NUMSERIES > 1` and `NUMGROUPS` groups are to be formed, `Design` is a cell array with `NUMSAMPLES` cells, where each cell contains a `NUMSERIES`-by-`(NUMGROUPS * NUMPARAMS)` matrix of known values.

The original collection of parameters that are common to all series are replicated to form collections of parameters for each group.

Examples

Use `convert2sur` to Estimate Stock Alpha and Beta Values

This example shows a CAPM demonstration using 6 stocks and 60 months of simulated asset returns, where the model for each stock is `AssetReturn = Alpha * 1 + CashReturn + Beta * (MarketReturn - CashReturn) + Noise` and the parameters to estimate are Alpha and Beta.

Using simulated data, where the Alpha estimate(s) are displayed in the first row(s) and the Beta estimate(s) are display in the second row(s).

```
Market = (0.1 - 0.04) + 0.17*randn(60, 1);
Asset = (0.1 - 0.04) + 0.35*randn(60, 6);

Design = cell(60, 1);
for i = 1:60
    Design{i} = repmat([ 1, Market(i) ], 6, 1);
end
```

Obtain the aggregate estimates for all stocks.

```
[Param, Covar] = mvnrmls(Asset, Design);
```

```
disp({'All 6 Assets Combined'});
disp(Param);
```

```
'All 6 Assets Combined'
```

```
0.0233
```

```
0.1050
```

Estimate parameters for individual stocks using `convert2sur`

```
Group = 1:6;
DesignSUR = convert2sur(Design, Group);
[Param, Covar] = mvnrmlc(Asset, DesignSUR);
Param = reshape(Param, 2, 6);
```

```
disp({'A', 'B', 'C', 'D', 'E', 'F'});
disp(Param);
```

'A'	'B'	'C'	'D'	'E'	'F'
0.0144	0.0270	0.0046	0.0419	0.0376	0.0291
0.3264	-0.1716	0.3248	-0.0630	-0.0001	0.0637

Estimate parameters for pairs of stocks by forming groups.

```
disp({'A & B', 'C & D', 'E & F'});
Group = { [1,2 ], [3,4], [5,6] };
DesignSUR = convert2sur(Design, Group);
[Param, Covar] = mvnrmlc(Asset, DesignSUR);
```

```
Param = reshape(Param, 2, 3);
```

```
disp(Param);
```

'A & B'	'C & D'	'E & F'
0.0186	0.0190	0.0334
0.0988	0.1757	0.0293

cir

Cox-Ingersoll-Ross mean-reverting square root diffusion models

Syntax

```
CIR = cir(Speed, Level, Sigma)
```

```
CIR = cir(Speed, Level, Sigma, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Class

CIR

Description

This constructor creates and displays **CIR** objects, which derive from the **SDEMURD** (SDE with drift rate expressed in mean-reverting form) class. 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:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^2)V(t)dW_t$$

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

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Speed	<p>Speed represents S. 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). As a deterministic function of time, when Speed is called with a real-valued scalar time \mathbf{t} as its only input, Speed must produce an NVARs-by-NVARs matrix</p> <p>If you specify Speed as a function of time and state, it must generate an NVARs-by-NVARs matrix of reversion rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
--------------	---

Level	<p>Level represents L. If you specify Level as an array, it must be an NVARs-by-1 column vector of reversion levels. As a deterministic function of time, when Level is called with a real-valued scalar time t as its only input, Level must produce an NVARs-by-1 matrix.</p> <p>If you specify Level as a function of time and state, it must generate an NVARs-by-1 column vector of reversion levels when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V. 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. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARs-by-NBROWNS matrix.</p> <p>If you specify Sigma as a function of time and state, it must generate an NVARs-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.

Note: Although the constructor does not enforce restrictions on the signs of these input arguments, each argument is usually specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, cir applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, cir applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, cir applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>

Simulation	A user-defined simulation function or SDE simulation method. If you do not specify a value for <code>Simulation</code> , the default method is simulation by Euler approximation (<code>simByEuler</code>).
------------	---

Output Arguments

CIR	<p>Object of class <code>CIR</code> with the following displayed parameters:</p> <ul style="list-style-type: none"> • <code>StartTime</code>: Initial observation time • <code>StartState</code>: Initial state at time <code>StartTime</code> • <code>Correlation</code>: Access function for the <code>Correlation</code> input argument, callable as a function of time • <code>Drift</code>: Composite drift-rate function, callable as a function of time and state • <code>Diffusion</code>: Composite diffusion-rate function, callable as a function of time and state • <code>Simulation</code>: A simulation function or method • <code>Speed</code>: Access function for the input argument <code>Speed</code>, callable as a function of time and state • <code>Level</code>: Access function for the input argument <code>Level</code>, callable as a function of time and state • <code>Sigma</code>: Access function for the input argument <code>Sigma</code>, callable as a function of time and state
-----	--

Examples

“Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models” on page 17-28

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | diffusion | sdeddo

convertto

Convert to specified frequency

Syntax

```
newfts = convertto(oldfts, newfreq)
```

```
newfts = convertto(oldfts, newfreq, 'param1', 'value1', 'param2',  
'value2', ... )
```

Arguments

oldfts	Name of an existing financial time series object.
newfreq	1, DAILY, Daily, daily, D, d
	2, WEEKLY, Weekly, weekly, W, w
	3, MONTHLY, Monthly, monthly, M, m
	4, QUARTERLY, Quarterly, quarterly, Q, q
	5, SEMIANNUAL, Semiannual, semiannual, S, s
	6, ANNUAL, Annual, annual, A, a

Description

`convertto` converts a financial time series of any frequency to one of a specified frequency.

`newfts = convertto(oldfts, newfreq)` converts the object `oldfts` to the new time series object `newfts` with the frequency `newfreq`.

Refer to the documentation for each frequency conversion function to determine the valid parameter/value pairs.

See Also

toannual | todaily | tomonthly | toquarterly | tosemi | toweekly

corrcoef

Correlation coefficients

Syntax

`r = corrcoef(X)`

`r = corrcoef(X,Y)`

Arguments

X	Matrix where each row is an observation and each column is a variable.
Y	Matrix where each row is an observation and each column is a variable.

Description

`corrcoef` for financial time series objects is based on the MATLAB `corrcoef` function. See `corrcoef` in the MATLAB documentation.

`r=corrcoef(X)` calculates a matrix `r` of correlation coefficients for an array `X`, in which each row is an observation and each column is a variable.

`r=corrcoef(X,Y)`, where `X` and `Y` are column vectors, is the same as `r=corrcoef([X Y])`. `corrcoef` converts `X` and `Y` to column vectors if they are not; that is, `r = corrcoef(X,Y)` is equivalent to `r=corrcoef([X(:) Y(:)])` in that case.

If `c` is the covariance matrix, `c= cov(X)`, then `corrcoef(X)` is the matrix whose (i,j) 'th element is $c_{i,j}/\sqrt{c_{i,i}*c_{j,j}}$.

`[r,p]=corrcoef(...)` also returns `p`, a matrix of `p`-values for testing the hypothesis of no correlation. Each `p`-value is the probability of getting a correlation as large as the

observed value by random chance, when the true correlation is zero. If $p(i, j)$ is less than 0.05, then the correlation $r(i, j)$ is significant.

`[r,p,rlo,rup]=corrcoef(...)` also returns matrices `rlo` and `rup`, of the same size as `r`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(..., 'PARAM1', VAL1, 'PARAM2', VAL2, ...)` specifies additional parameters and their values. Valid parameters are:

- `'alpha'` — A number between 0 and 1 to specify a confidence level of $100*(1-ALPHA)\%$. Default is 0.05 for 95% confidence intervals.
- `'rows'` — Either `'all'` (default) to use all rows, `'complete'` to use rows with no NaN values, or `'pairwise'` to compute $r(i, j)$ using rows with no NaN values in column i or j .

The p-value is computed by transforming the correlation to create a t-statistic having $N - 2$ degrees of freedom, where N is the number of rows of X . The confidence bounds are based on an asymptotic normal distribution of $0.5*\log((1+r)/(1-r))$, with an approximate variance equal to $1/(N - 3)$. These bounds are accurate for large samples when X has a multivariate normal distribution. The `'pairwise'` option can produce an `r` matrix that is not positive definite.

Examples

Compute Correlation Coefficients

This example shows how to generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4);           % uncorrelated data
x(:,4) = sum(x,2);        % introduce correlation
f = fints('today:today+29', x); % create a fints object using x
[r,p] = corrcoef(x);      % compute sample correlation and p-values
[i,j] = find(p<0.05);    % find significant correlations
[i,j]                    % display their (row,col) indices
```

```
ans =
```

```
4      1
```


3	2
2	3
1	4

Class support for inputs X,Y: float: double and single.

See Also

cov | std | var

corr2cov

Convert standard deviation and correlation to covariance

Syntax

```
ExpCovariance = corr2cov(ExpSigma, ExpCorrC)
```

Arguments

ExpSigma	Vector of length n with the standard deviations of each process. n is the number of random processes.
ExpCorrC	(Optional) n -by- n correlation coefficient matrix. If ExpCorrC is not specified, the processes are assumed to be uncorrelated, and the identity matrix is used.

Description

corr2cov converts standard deviation and correlation to covariance.

ExpCovariance is an n -by- n covariance matrix, where n is the number of processes.

```
ExpCov(i,j) = ExpCorrC(i,j)*ExpSigma(i)*ExpSigma(j)
```

Examples

Convert Standard Deviation and Correlation to Covariance

This example shows how to convert standard deviation and correlation to covariance.

```
ExpSigma = [0.5  2.0];
```

```
ExpCorrC = [1.0 -0.5  
           -0.5  1.0];
```

```
ExpCovariance = corr2cov(ExpSigma, ExpCorrC)
```

ExpCovariance =

```
    0.2500    -0.5000
   -0.5000     4.0000
```

See Also

[corrcoef](#) | [cov](#) | [cov2corr](#) | [ewstats](#) | [std](#)

COV

Covariance matrix

Syntax

`cov(X)`

`cov(X,Y)`

Arguments

X	Financial times series object.
Y	Financial times series object.

Description

`cov` for financial time series objects is based on the MATLAB `cov` function. See `cov` in the MATLAB documentation.

If `X` is a financial time series object with one series, `cov(X)` returns the variance. For a financial time series object containing multiple series, where each row is an observation, and each series a variable, `cov(X)` is the covariance matrix.

`diag(cov(X))` is a vector of variances for each series and `sqrt(diag(cov(X)))` is a vector of standard deviations.

`cov(X, Y)`, where `X` and `Y` are financial time series objects with the same number of elements, is equivalent to `cov([X(:) Y(:)])`.

`cov(X)` or `cov(X, Y)` normalizes by $(N - 1)$ if $N > 1$, where N is the number of observations. This makes `cov(X)` the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For $N = 1$, `cov` normalizes by N .

`cov(X, 1)` or `cov(X, Y, 1)` normalizes by N and produces the second moment matrix of the observations about their mean. `cov(X, Y, 0)` is the same as `cov(X, Y)` and

$\text{cov}(X, 0)$ is the same as $\text{cov}(X)$. The mean is removed from each column before calculating the result.

Examples

Create a Covariance Matrix

This example shows how to create a covariance matrix for the following dates.

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'};  
A = [-1 1 2 ; -2 3 1 ; 4 0 3];  
f = fints(dates, A);
```

```
c = cov(f)
```

```
c =
```

```
10.3333    -4.1667     3.0000  
-4.1667     2.3333    -1.5000  
 3.0000    -1.5000     1.0000
```

See Also

[corrcoef](#) | [mean](#) | [std](#) | [var](#)

cov2corr

Convert covariance to standard deviation and correlation coefficient

Syntax

```
[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)
```

Arguments

ExpCovariance	<i>n</i> -by- <i>n</i> covariance matrix; for example, from <code>COV</code> or <code>ewstats</code> . <i>n</i> is the number of random processes.
---------------	--

Description

`[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)` converts covariance to standard deviations and correlation coefficients.

`ExpSigma` is a 1-by-*n* vector with the standard deviation of each process.

`ExpCorrC` is an *n*-by-*n* matrix of correlation coefficients.

```
ExpSigma(i) = sqrt(ExpCovariance(i,i))  
ExpCorrC(i,j) = ExpCovariance(i,j)/(ExpSigma(i)*ExpSigma(j))
```

Examples

Convert Covariance to Standard Deviations and Correlation Coefficients

This example shows how to convert a covariance matrix to standard deviations and correlation coefficients.

```
ExpCovariance = [0.25 -0.5  
                -0.5  4.0];
```

```
[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)
```

ExpSigma =

0.5000 2.0000

ExpCorrC =

1.0000 -0.5000
-0.5000 1.0000

See Also

[corr2cov](#) | [cov](#) | [corrcoef](#) | [ewstats](#) | [std](#)

cpncount

Coupon payments remaining until maturity

Syntax

```
NumCouponsRemaining = cpncount(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
StartDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)

	<ul style="list-style-type: none"> • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.
StartDate	(Future implementation)

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Description

`NumCouponsRemaining = cpncount(Settle, Maturity, Period, Basis, EndMonthRule)` returns the whole number of coupon payments between the settlement and maturity dates for a coupon bond or set of bonds. Coupons falling on or before settlement are not counted, except for the maturity payment which is always counted.

Examples

Find Coupon Payments Remaining Until Maturity

This example shows how to find the coupon payments remaining until maturity.

```
NumCouponsRemaining = cpncount('14 Mar 1997', '30 Nov 2000', ...  
2, 0, 0)
```

```
NumCouponsRemaining =
```

```
8
```

Find Coupon Payments Remaining Until Maturity for Different Maturity Dates

This example shows how to find the coupon payments remaining until maturity, given three coupon bonds with different maturity dates and the same default arguments.

```
Maturity = ['30 Sep 2000'; '31 Oct 2001'; '30 Nov 2002'];  
NumCouponsRemaining = cpncount('14 Sep 1997', Maturity)
```

```
NumCouponsRemaining =
```

```
7  
9  
11
```

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz`

cpn Daten

Next coupon date for fixed-income security

Syntax

```
NextCouponDate = cpn Daten(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA)

	<ul style="list-style-type: none"> • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Description

`NextCouponDate = cpndaten(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` returns the

next coupon date after the settlement date. This function finds the next coupon date whether or not the coupon structure is synchronized with the maturity date.

`NextCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

```
NextCouponDate = cpndaten('14 Mar 1997', '30 Nov 2000', 2, 0, 0);  
datestr(NextCouponDate)
```

```
ans =
```

```
30-May-1997
```

```
NextCouponDate = cpndaten('14 Mar 1997', '30 Nov 2000', 2, 0, 1);  
datestr(NextCouponDate)
```

```
ans =
```

```
31-May-1997
```

```
Maturity = ['30 Sep 2000'; '31 Oct 2000'; '30 Nov 2000'];  
NextCouponDate = cpndaten('14 Mar 1997', Maturity);  
datestr(NextCouponDate)
```

```
ans =
```

```
31-Mar-1997
```

```
30-Apr-1997
```

```
31-May-1997
```

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndatenq` | `cpndatep`
| `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz`

cpndatenq

Next quasi-coupon date for fixed-income security

Syntax

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA)

	<ul style="list-style-type: none"> • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices. Fill unspecified entries in input vectors with the value NaN. Dates can be serial date numbers or date strings.

Description

`NextQuasiCouponDate = cpndatenq(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` determines

the next quasi coupon date for a portfolio of NUMBONDS fixed income securities whether or not the first or last coupon is normal, short, or long. For zero coupon bonds `cpndatenq` returns quasi coupon dates as if the bond had a semiannual coupon structure. Successive quasi coupon dates determine the length of the standard coupon period for the fixed income security of interest and do not necessarily coincide with actual coupon payment dates.

Outputs are NUMBONDS-by-1 vectors.

If `Settle` is a coupon date, this function never returns the settlement date. It returns the quasi coupon date strictly after settlement.

`NextQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

Determine the Next Quasi Coupon Date for a Portfolio of Fixed-Income Securities

Given a pair of bonds with the following characteristics:

```
Settle = char('30-May-1997','10-Dec-1997');  
Maturity = char('30-Nov-2002','10-Jun-2004');
```

Compute `NextCouponDate` for this pair of bonds.

```
NextCouponDate = cpndaten(Settle, Maturity);  
datestr(NextCouponDate)
```

```
ans =
```

```
31-May-1997  
10-Jun-1998
```

Compute the next quasi coupon dates for these two bonds.

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity);  
datestr(NextQuasiCouponDate)
```



```
ans =
31-May-1997
10-Jun-1998
```

Because no `FirstCouponDate` has been specified, the results are identical.

Now supply an explicit `FirstCouponDate` for each bond.

```
FirstCouponDate = char('30-Nov-1997','10-Dec-1998');
```

Compute the next coupon dates.

```
NextCouponDate = cpndaten(Settle, Maturity, 2, 0, 1, [],...
FirstCouponDate);
datestr(NextCouponDate)
```

```
ans =
30-Nov-1997
10-Dec-1998
```

The next coupon dates are identical to the specified first coupon dates.

Now recompute the next quasi coupon dates.

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity, 2, 0, 1, [],...
FirstCouponDate);
datestr(NextQuasiCouponDate)
```

```
ans =
31-May-1997
10-Jun-1998
```

These results illustrate the distinction between actual coupon payment dates and quasi coupon dates. `FirstCouponDate` (and `LastCouponDate`, as well), when specified, is associated with an actual coupon payment and also serves as the synchronization date for determining all quasi coupon dates. Since each bond in this example pays semiannual coupons, and the first coupon date occurs more than six months after settlement, each

will have an intermediate quasi coupon date before the actual first coupon payment occurs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatep | cpndatepq | cpndaysn | cpndaysp | cnpersz

cpndatep

Previous coupon date for fixed-income security

Syntax

```
PreviousCouponDate = cpndatep(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA)

	<ul style="list-style-type: none"> • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Description

`PreviousCouponDate = cpndatep(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` returns the

previous coupon date on or before settlement for a portfolio of bonds. This function finds the previous coupon date whether or not the coupon structure is synchronized with the maturity date.

For zero coupon bonds the previous coupon date is the issue date, if available. However, if the issue date is not supplied, the previous coupon date for zero coupon bonds is the previous quasi coupon date calculated as if the frequency is semiannual.

PreviousCouponDate is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

```
PreviousCouponDate = cpndatep('14 Mar 1997', '30 Jun 2000',...
2, 0, 0);
datestr(PreviousCouponDate)
```

ans =

30-Dec-1996

```
PreviousCouponDate = cpndatep('14 Mar 1997', '30 Jun 2000',...
2, 0, 1);
datestr(PreviousCouponDate)
```

ans =

31-Dec-1996

```
Maturity = ['30 Apr 2000'; '31 May 2000'; '30 Jun 2000'];
PreviousCouponDate = cpndatep('14 Mar 1997', Maturity);
datestr(PreviousCouponDate)
```

ans =

31-Oct-1996

30-Nov-1996

31-Dec-1996

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndaten` | `cpndateng`
| `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz`

cpndatepq

Previous quasi coupon date for fixed income security

Syntax

PreviousQuasiCouponDate = cpndatepq(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA)

	<ul style="list-style-type: none"> • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices. Fill unspecified entries in input vectors with the value NaN. Dates can be serial date numbers or date strings.

Description

`PreviousQuasiCouponDate = cpndatepq(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` determines

the previous quasi coupon date on or before settlement for a set of NUMBONDS fixed income securities. This function finds the previous quasi coupon date for a bond with a coupon structure in which the first or last period is either normal, short, or long (whether or not the coupon structure is synchronized to maturity). For zero coupon bonds this function returns quasi coupon dates as if the bond had a semiannual coupon structure.

The term “previous quasi coupon date” refers to the previous coupon date for a bond calculated as if no issue date were specified. Although the issue date is not actually a coupon date, when issue date is specified, the previous actual coupon date for a bond is normally calculated as being either the previous coupon date or the issue date, whichever is greater. This function always returns the previous quasi coupon date regardless of issue date. If the settlement date is a coupon date, this function returns the settlement date.

PreviousQuasiCouponDate is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

Determine the Previous Quasi Coupon Date for a Portfolio of Fixed-Income Securities

Given a pair of bonds with the following characteristics:

```
Settle = char('30-May-1997','10-Dec-1997');  
Maturity = char('30-Nov-2002','10-Jun-2004');
```

With no `FirstCouponDate` explicitly supplied, compute the `PreviousCouponDate` for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity);  
datestr(PreviousCouponDate)
```

```
ans =
```

```
30-Nov-1996  
10-Dec-1997
```

Note that since the settlement date for the second bond is also a coupon date, `cpndatep` returns this date as the previous coupon date.

Now establish a `FirstCouponDate` and `IssueDate` for this pair of bonds.


```
FirstCouponDate = char('30-Nov-1997', '10-Dec-1998');
IssueDate = char('30-May-1996', '10-Dec-1996');
```

Recompute the PreviousCouponDate for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity, 2, 0, 1, ...
IssueDate, FirstCouponDate);
datestr(PreviousCouponDate)
```

ans =

```
30-May-1996
10-Dec-1996
```

Since both of these bonds settled before the first coupon had been paid, cpndatep returns the IssueDate as the PreviousCouponDate.

Using the same data, compute PreviousQuasiCouponDate.

```
PreviousQuasiCouponDate = cpndatepq(Settle, Maturity, 2, 0, 1, ...
IssueDate, FirstCouponDate);
datestr(PreviousQuasiCouponDate)
```

ans =

```
30-Nov-1996
10-Dec-1997
```

For the first bond the settlement date is not a normal coupon date. The PreviousQuasiCouponDate is the coupon date before or on the settlement date. Since the coupon structure is synchronized to FirstCouponDate, the previous quasi coupon date is 30-Nov-1996. PreviousQuasiCouponDate disregards IssueDate and FirstCouponDate in this case. For the second bond the settlement date (10-Dec-1997) occurs on a date when a coupon would normally be paid in the absence of an explicit FirstCouponDate. cpndatepq returns this date as PreviousQuasiCouponDate.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndateng
| cpndatep | cpndaysn | cpndaysp | cpnpersz

cpndaysn

Number of days to next coupon date

Syntax

```
NumDaysNext = cpndaysn(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
StartDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)

	<ul style="list-style-type: none"> • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.
StartDate	(Future implementation)

Required arguments must be number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Description

NumDaysNext = cpndaysn(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate) returns the number of days from the settlement date to the next coupon date for a bond or set of bonds. For zero coupon bonds coupon dates are computed as if the bonds have a semiannual coupon structure.

Examples

```
NumDaysNext = cpndaysn('14 Sep 2000', '30 Jun 2001', 2, 0, 0)
```

```
NumDaysNext =
```

```
107
```

```
NumDaysNext = cpndaysn('14 Sep 2000', '30 Jun 2001', 2, 0, 1)
```

```
NumDaysNext =
```

```
108
```

```
Maturity = ['30 Apr 2001'; '31 May 2001'; '30 Jun 2001'];  
NumDaysNext = cpndaysn('14 Sep 2000', Maturity)
```

```
NumDaysNext =
```

```
47
```

```
77
```

```
108
```

See Also

accrfrac | cfamounts | cftimes | cfdates | cpncount | cpndaten | cpndateng
| cpndatep | cpndatepq | cpndaysp | cpnpersz

cpndaysp

Number of days since previous coupon date

Syntax

```
NumDaysPrevious = cpndaysp(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
StartDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)

	<ul style="list-style-type: none"> • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.
StartDate	(Future implementation)

Required arguments must be a number of bonds (**NUMBONDS**)-by-1 or 1-by-**NUMBONDS** conforming vectors or scalars. Optional arguments must be either **NUMBONDS**-by-1 or 1-by-**NUMBONDS** conforming vectors, scalars, or empty matrices.

Description

NumDaysPrevious = cpndaysp(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate) returns the number of days between the previous coupon date and the settlement date for a bond or set of bonds. When the coupon frequency is 0 (a zero coupon bond), the previous coupon date is calculated as if the frequency were semiannual.

Examples

```
NumDaysPrevious = cpndaysp('14 Mar 2000', '30 Jun 2001', 2, 0, 0)
```

```
NumDaysPrevious =
```

```
75
```

```
NumDaysPrevious = cpndaysp('14 Mar 2000', '30 Jun 2001', 2, 0, 1)
```

```
NumDaysPrevious =
```

```
74
```

```
Maturity = ['30 Apr 2001'; '31 May 2001'; '30 Jun 2001'];
NumDaysPrevious = cpndaysp('14 Mar 2000', Maturity)
```

```
NumDaysPrevious =
```

```
135
```

```
105
```

```
74
```

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndateng
| cpndatep | cpndatepq | cpndaysn | cnpersz

cpnpersz

Number of days in coupon period

Syntax

```
NumDaysPeriod = cpnpersz(Settle, Maturity, Period, Basis,
EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
StartDate)
```

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Maturity date. A vector of serial date numbers or date strings.
Period	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)

	<ul style="list-style-type: none"> • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment; used when bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate , the cash flow payment dates are determined from other inputs.
LastCouponDate	(Optional) Last coupon date of a bond before the maturity date; used when bond has an irregular last coupon period. In the absence of a specified FirstCouponDate , a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate , regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a LastCouponDate , the cash flow payment dates are determined from other inputs.
StartDate	(Future implementation)

Required arguments must be a number of bonds (NUMBONDS)-by-1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Description

`NumDaysPeriod = cnpersz(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns the number of days in the coupon period containing the settlement date. For zero coupon bonds coupon dates are computed as if the bonds have a semiannual coupon structure.

Examples

```
NumDaysPeriod = cnpersz('14 Sep 2000', '30 Jun 2001', 2, 0, 0)
```

```
NumDaysPeriod =
```

```
183
```

```
NumDaysPeriod = cnpersz('14 Sep 2000', '30 Jun 2001', 2, 0, 1)
```

```
NumDaysPeriod =
```

```
184
```

```
Maturity = ['30 Apr 2001'; '31 May 2001'; '30 Jun 2001'];  
NumDaysPeriod = cnpersz('14 Sep 2000', Maturity)
```

```
NumDaysPeriod =
```

```
184
```

```
183
```

```
184
```

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp`

createholidays

Create trading calendars

Syntax

```
createholidays(Filename, Codefile, InfoFile, TargetDir,  
IncludeWkds, Wprompt, NoGUI)
```

Arguments

Filename	The data file name.
Codefile	The code file name.
InfoFile	The info file name.
TargetDir	The target folder where to write the new <code>holidays.m</code> files.
IncludeWkds	Option to include weekends in the holiday list. Values are: <ul style="list-style-type: none">• 0 – Do not include weekends in the holiday list.• 1 – Include weekends in the holiday list.
Wprompt	Option to prompt for the file location for each <code>holiday.m</code> file that is created. Values are: <ul style="list-style-type: none">• 0 – Do not prompt for the file location.• 1 – Prompt for the file location.
NoGUI	Run <code>createholidays</code> without displaying the Trading Calendars graphical user interface. Values are: <ul style="list-style-type: none">• 0 – Display the GUI.• 1 – Do not display the GUI.

Description

`createholidays(Filename, Codefile, InfoFile, TargetDir, IncludeWkds, Wprompt, NoGUI)` programmatically generates the market-specific `holidays.m` files without displaying the interface.

Note: To use `createholidays`, you must obtain data, codes, and info files from <http://www.FinancialCalendar.com> trading calendars.

Examples

```
createholidays('FinancialCalendar\My_datafile.csv',...  
'FinancialCalendar\My_codesfile.csv',...  
'FinancialCalendar\My_infofile.csv', 'c:\work', 1, 1, 1)
```

will create `holidays*.m` files from `My_datafile.csv` in the folder `c:\work`. Weekends will be included in the holidays list based on the input flag `INCLUDEWDKS = 1`.

See Also

`holidays`

cumsum

Cumulative sum

Syntax

```
newfts = cumsum(oldfts)
```

Description

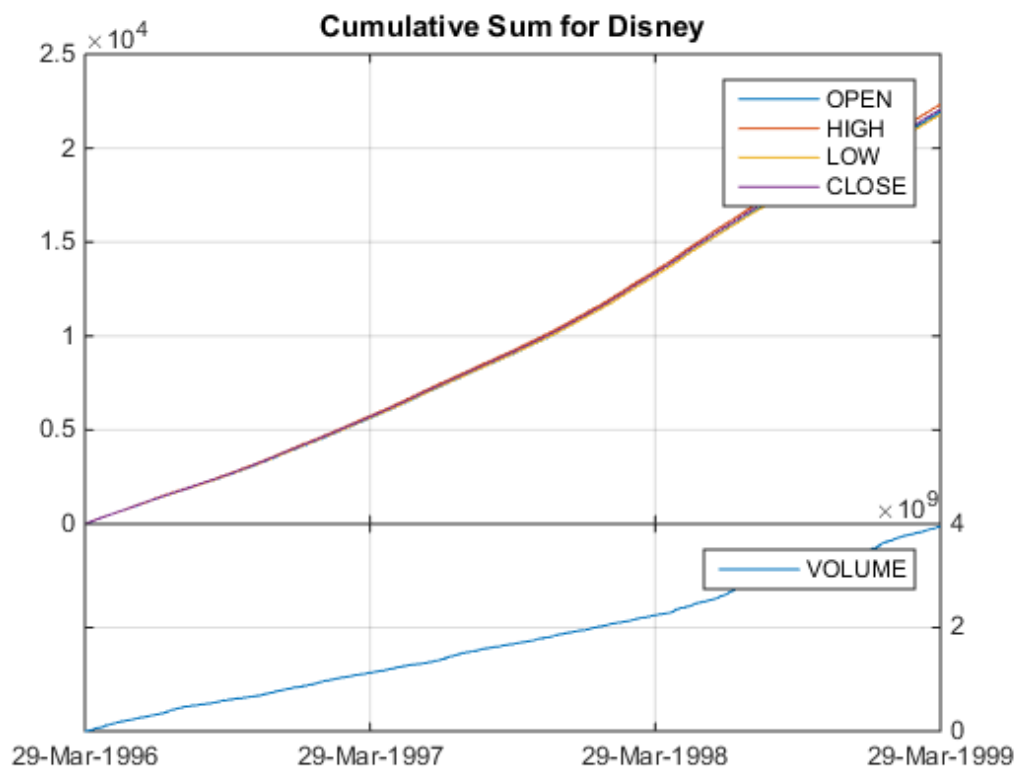
`newfts = cumsum(oldfts)` calculates the cumulative sum of each individual time series data series in the financial time series object `oldfts` and returns the result in another financial time series object `newfts`. `newfts` contains the same data series names as `oldfts`.

Examples

Compute the Cumulative Sum

This example shows how to compute the cumulative sum for Disney stock and plot the results.

```
load disney.mat
cs_dis = cumsum(fillts(dis));
plot(cs_dis)
title('Cumulative Sum for Disney')
```



See Also
cumsum

cur2frac

Decimal currency values to fractional values

Syntax

```
Fraction = cur2frac(Decimal, Denominator)
```

Description

`Fraction = cur2frac(Decimal, Denominator)` converts decimal currency values to fractional values. `Fraction` is returned as a string.

Examples

Convert Decimal Currency Values to Fractional Values

This example shows how to convert decimal currency values to fractional values.

```
Fraction = cur2frac(12.125, 8)
```

```
Fraction =
```

```
12.1
```

See Also

cur2str | frac2cur

cur2str

Bank-formatted text

Syntax

```
String = cur2str(Value, Digits)
```

Description

`String = cur2str(Value, Digits)` returns the given value in bank format. By default, `Digits = 2`. A negative `Digits` rounds the value to the left of the decimal point. `String` is returned as a string with a leading dollar sign (\$). Negative numbers are displayed in parentheses.

Examples

Return Bank-Formatted Text

This example shows how to return a given value in bank format.

```
String = cur2str(-8264, 2)
```

```
String =  
($8264.00)
```

See Also

[cur2frac](#) | [frac2cur](#)

date2time

Time and frequency from dates

Syntax

[TFactors, F] = date2time(Settle, Maturity, Compounding, Basis, EndMonthRule)

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings.
Maturity	A vector of serial maturity dates.
Compounding	<p>Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors:</p> <ul style="list-style-type: none"> • Compounding = 1, 2, 3, 4, 6, 12 • Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units, for example, $T = F$ is one year. • Compounding = 365 • Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis. • Compounding = -1 • Disc = $\exp(-T*Z)$, where T is time in years.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360

	<ul style="list-style-type: none"> • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Description

[TFactors, F] = `date2time`(Settle, Maturity, Compounding, Basis, EndMonthRule) computes time factors appropriate to compounded rate quotes between the settlement and maturity dates.

TFactors is a vector of time factors.

F is a scalar of related compounding frequencies.

`date2time` is the inverse of `time2date`.

Definitions

The difference between `yearfrac` and `date2time` is that `date2time` counts full periods as a whole integer, even if the number of actual days in the periods are different. `yearfrac` does not count full periods. For example,

```
yearfrac('1/1/2000', '1/1/2001', 9)
```

```
ans =
```

```
1.0167
```

`yearfrac` for Basis 9 (ACT/360 ISMA) will calculate $366/360 = 1.0167$. So, even if the dates have the same month and date, with a difference of 1 in the year, the returned value may not be exactly 1. On the other hand, `date2time` will calculate one full year period:

```
date2time('1/1/2000', '1/1/2001', 1, 9)
```

```
ans =
```

```
1
```

See Also

`cftimes` | `disc2rate` | `rate2disc` | `time2date` | `yearfrac`

dateaxis

Convert serial-date axis labels to calendar-date axis labels

Syntax

```
dateaxis(Tickaxis, DateForm, StartDate)
```

Arguments

Tickaxis	(Optional) Determines which axis tick labels— <i>x</i> , <i>y</i> , or <i>z</i> —to replace. Enter as a string. Default = 'x'.
DateForm	(Optional) Specifies which date format to use. Enter as an integer from 0 to 17. If no DateForm argument is entered, this function determines the date format based on the span of the axis limits. For example, if the difference between the axis minimum and maximum is less than 15, the tick labels are converted to three-letter day-of-the-week abbreviations (DateForm = 8). See DateForm format descriptions below.
StartDate	(Optional) Assigns the date to the first axis tick value. Enter as a string. The tick values are treated as serial date numbers. The default StartDate is the lower axis limit converted to the appropriate date number. For example, a tick value of 1 is converted to the date 01-Jan-0000. Entering StartDate as '06-apr-1999' assigns the date April 6, 1999 to the first tick value and the axis tick labels are set accordingly.

Description

`dateaxis(Tickaxis, DateForm, StartDate)` replaces axis tick labels with date labels on a graphic figure.

See the MATLAB `set` command for information on modifying the axis tick values and other axis parameters.

DateForm	Format	Description
0	01-Mar-1999 15:45:17	day-month-year hour:minute:second
1	01-mar-1999	day-month-year
2	03/01/99	month/day/year
3	Mar	month, three letters
4	M	month, single letter
5	3	month
6	03/01	month/day
7	1	day of month
8	Wed	day of week, three letters
9	W	day of week, single letter
10	1999	year, four digits
11	99	year, two digits
12	Mar99	month year
13	15:45:17	hour:minute:second
14	03:45:17 PM	hour:minute:second AM or PM
15	15:45	hour:minute
16	03:45 PM	hour:minute AM or PM
17	95/03/01	year month day

Examples

`dateaxis('x')` or `dateaxis`

converts the *x*-axis labels to an automatically determined date format.

`dateaxis('y', 6)`

converts the *y*-axis labels to the month/day format.

`dateaxis('x', 2, '03/03/1999')`

converts the *x*-axis labels to the month/day/year format. The minimum *x*-tick value is treated as March 3, 1999.

See Also

bolling | candle | datenum | datestr | highlow | movavg | pointfig

datedisp

Display date entries

Syntax

```
datedisp(NumMat, DateForm)
```

```
CharMat = datedisp(NumMat, DateForm)
```

Arguments

NumMat	Numeric matrix to display.
DateForm	(Optional) Date format. See <code>datestr</code> for available and default format flags.

Description

`datedisp(NumMat, DateForm)` displays a matrix with the serial dates formatted as date strings, using a matrix with mixed numeric entries and serial date number entries. Integers between `datenum('01-Jan-1900')` and `datenum('01-Jan-2200')` are assumed to be serial date numbers, while all other values are treated as numeric entries.

`CharMat` is a character array representing `NumMat`. If no output variable is assigned, the function prints the array to the display.

Examples

Display a Matrix with the Serial Dates Formatted as Date Strings

This example shows how to display a matrix with the serial dates formatted as date strings.

```
NumMat = [730730, 0.03, 1200 730100;  
          730731, 0.05, 1000 NaN];
```

`datedisp(NumMat)`

```
01-Sep-2000  0.03  1200  11-Dec-1998
02-Sep-2000  0.05  1000      NaN
```

See Also

`datenum` | `datestr`

datefind

Indices of date numbers in matrix

Syntax

```
Indices = datefind(Subset, Superset, Tolerance)
```

Arguments

Subset	Subset matrix of date numbers used to find matching date numbers in Superset . These date numbers must be a nonrepeating subset of those in Superset .
Superset	Superset matrix of nonrepeating date numbers whose elements are sought.
Tolerance	(Optional) Tolerance (+ / -) for matching the date numbers in Superset . A positive integer. Default = 0.

Description

`Indices = datefind(Subset, Superset, Tolerance)` returns a vector of indices to the date numbers in **Superset** that are present in **Subset**, plus or minus the **Tolerance**. If no date numbers match, `Indices = []`.

Although this function was designed for use with sequential date numbers, you can use it with any nonrepeating integers.

Examples

Return a Vector of Indices to Date Numbers

This example shows how to return a vector of indices to date numbers.

```
Superset = datenum(1999, 7, 1:31);
```

```
Subset = [datenum(1999, 7, 10); datenum(1999, 7, 20)];  
Indices = datefind(Subset, Superset, 1)
```

```
Indices =
```

```
     9  
    10  
    11  
    19  
    20  
    21
```

See Also

datenum

datemnth

Date of day in future or past month

Syntax

TargetDate = datemnth(StartDate, NumberMonths, DayFlag, Basis, EndMonthRule)

Arguments

StartDate	Enter as serial date numbers or date strings.
NumberMonths	Vector containing number of months in future (positive) or past (negative). Values must be in integer form.
DayFlag	<p>(Optional) Vector containing values that specify how the actual day number for the target date in future or past month is determined. 0 (default) = day number should be the day in the future or past month corresponding to the actual day number of the start date. 1 = day number should be the first day of the future or past month. 2 = day number should be the last day of the future or past month.</p> <p>This flag has no effect if EndMonthRule is set to 1.</p>
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

	<ul style="list-style-type: none"> • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	<p>(Optional) End-of-month rule. A vector. 1 = rule in effect, meaning that if you are beginning on the last day of a month, and the month has 30 or fewer days, you will end on the last actual day of the future or past month regardless of whether that month has 28, 29, 30 or 31 days)</p> <p>0 = rule off (default), meaning that the rule is not in effect.</p>

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if **StartDate** is an *n*-row character array of date strings, then **NumberMonths** must be an *n*-by-1 vector of integers or a single integer. **TargetDate** is then an *n*-by-1 vector of date numbers.

Description

`TargetDate = datemnth(StartDate, NumberMonths, DayFlag, Basis, EndMonthRule)` returns the serial date number of the target date in the future or past.

Use `datestr` to convert serial date numbers to formatted date strings.

Examples

```
Day = datemnth('3 jun 2001', 6, 0, 0, 0)
```

```
Day =
    731188
```

```
datestr(Day)
```

```
ans =
```

03-Dec-2001

```
Day = datemnth('3 jun 2001', 6, 1, 0, 1); datestr(Day)
```

```
ans =  
01-Dec-2001
```

```
Day = datemnth('31 jan 2001', 5, 0, 0, 0); datestr(Day)
```

```
ans =  
30-Jun-2001
```

```
Day = datemnth('31 jan 2001', 5, 1, 0, 0); datestr(Day)
```

```
ans =  
01-Jun-2001
```

```
Day = datemnth('31 jan 2001', 5, 1, 0, 1); datestr(Day)
```

```
ans =  
30-Jun-2001
```

```
Day = datemnth('31 jan 2001', 5, 2, 0, 1); datestr(Day)
```

```
ans =  
30-Jun-2001
```

```
Months = [1; 3; 5; 7; 9];
```

```
Day = datemnth('31 jan 2001', Months); datestr(Day)
```

```
ans =  
28-Feb-2001  
30-Apr-2001  
30-Jun-2001  
31-Aug-2001  
31-Oct-2001
```

See Also

[datestr](#) | [datevec](#) | [days360](#) | [days365](#) | [daysact](#) | [daysdif](#) | [wrkdydif](#)

datewrkdy

Date of future or past workday

Syntax

```
EndDate = datewrkdy(StartDate, NumberWorkDays, NumberHolidays)
```

Arguments

StartDate	Start date vector. Enter as serial date numbers or date strings.
NumberWorkDays	Vector containing number of work or business days in future (positive) or past (negative), including the starting date.
NumberHolidays	Vector containing values for the number of holidays within NumberWorkDays. NumberHolidays and NumberWorkDays must have the same sign.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an `n`-row character array of date strings, then `NumberWorkDays` must be an `n`-by-1 vector of integers or a single integer. `EndDate` is then an `n`-by-1 vector of date numbers.

Description

`EndDate = datewrkdy(StartDate, NumberWorkDays, NumberHolidays)` returns the serial number of the date a given number of workdays before or after the start date.

Use `datestr` to convert serial date numbers to formatted date strings.

Examples

```
Workday = datewrkdy('12-dec-2000', 16, 2);  
datestr(Workday)
```

```
ans =  
04-Jan-2001
```

```
NumDays = [16; 20; 44];  
Workdays = datewrkdy('12-dec-2000', NumDays, 2);  
datestr(Workdays)
```

```
ans =  
4-Jan-2001  
10-Jan-2001  
13-Feb-2001
```

See Also

[busdate](#) | [holidays](#) | [isbusday](#) | [wrkdydif](#)

day

Day of month

Syntax

```
DayMonth = day(Date)
```

```
DayMonth = day(Date, F)
```

Description

`DayMonth = day(Date)` returns the day of the month given a serial date number or date string.

`DayMonth = day(Date, F)` returns the day of the of the month, given a serial date number or date string, in a specified date format.

Examples

```
DayMonth = day(730544)
```

or

```
DayMonth = day('2/28/00')
```

```
DayMonth =
```

```
28.00
```

You can also use the `F` argument to designate a country-specific date format:

```
DayMonth = day('28/02/00', 'dd/mm/yyyy')
```

```
DayMonth =
```

```
28.00
```


See Also

datevec | eomday | month | year

days252bus

Number of business days between dates

Syntax

```
NumberDays = days252bus(StartDate, EndDate)
```

```
NumberDays = days252bus(StartDate, EndDate, HolidayVector)
```

Arguments

StartDate	N-by-1 or 1-by-N vector or scalar value, in either serial date number or date string form, representing the start date.
EndDate	N-by-1 or 1-by-N vector or scalar value, in either serial date number or date string form, representing the end date.
HolidayVector	(Optional) N-by-1 or 1-by-N vector, in either serial date number or date string form, representing holidays.

Description

`NumberDays = days252bus(StartDate, EndDate, HolidayVector)` computes the number of business days (i.e. non-holiday or non-weekend) between the two input dates. Note that a holiday vector may be optionally specified; if it is not, then the `holidays.m` file is used to determine the holidays.

`days252bus` returns `NumberDays`, a N-by-1 or 1-by-N vector or scalar value for the number of days between two dates

Examples

Computes the Number of Business Days Between Two Input Dates

This example shows how to compute the number of business days (i.e. non-holiday or non-weekend) between two dates using the `days252bus` convention.

```
NumberDays = days252bus('1/1/2009', '8/1/2009')
```

```
NumberDays =
```

```
145
```

See Also

days360psa | daysact | daysdif | days365

days360

Days between dates based on 360-day year

Syntax

```
NumDays = days360(StartDate, EndDate)
```

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all. For example, if **StartDate** is an *n*-row character array of date strings, then **EndDate** must be an *n*-by-1 vector of integers or a single integer. **NumDays** is then an *n*-by-1 vector of date numbers.

Description

NumDays = `days360(StartDate, EndDate)` returns the number of days between **StartDate** and **EndDate** based on a 360-day year (that is, all months contain 30 days). If **EndDate** is earlier than **StartDate**, **NumDays** is negative.

Examples

```
NumDays = days360('15-jan-2000', '15-mar-2000')
```

```
NumDays =
```

```
60
```

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];  
NumDays = days360('15-jan-2000', MoreDays)
```

```
NumDays =
```

60
90
150

References

Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures*, Vol. 2, Spring 1995.

See Also

days365 | daysact | daysdif | wrkdydif | yearfrac

days360e

Days between dates based on 360-day year (European)

Syntax

```
NumDays = days360e(StartDate, EndDate)
```

Arguments

StartDate	Row vector, column vector, or scalar value in serial date number or date string format.
EndDate	Row vector, column vector, or scalar value in serial date number or date string format.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all.

Description

`NumDays = days360e(StartDate, EndDate)` returns a vector or scalar value representing the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

This day count convention is used primarily in Europe. Under this convention all months contain 30 days.

Examples

Example 1. Use this convention to find the number of days in the month of January.

```
StartDate = '1-Jan-2002';  
EndDate = '1-Feb-2002';
```

```
NumDays = days360e(StartDate, EndDate)
```

```
NumDays =
```

```
30
```

Example 2. Use this convention to find the number of days in February during a leap year.

```
StartDate = '1-Feb-2000';
```

```
EndDate = '1-Mar-2000';
```

```
NumDays = days360e(StartDate, EndDate)
```

```
NumDays =
```

```
30
```

Example 3. Use this convention to find the number of days in February of a non-leap year.

```
StartDate = '1-Feb-2002';
```

```
EndDate = '1-Mar-2002';
```

```
NumDays = days360e(StartDate, EndDate)
```

```
NumDays =
```

```
30
```

See Also

days360 | days360isda | days360psa

days360isda

Days between dates based on 360-day year (International Swap Dealer Association (ISDA) compliant)

Syntax

```
NumDays = days360isda(StartDate, EndDate)
```

Arguments

StartDate	Row vector, column vector, or scalar value in serial date number or date string format.
EndDate	Row vector, column vector, or scalar value in serial date number or date string format.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all.

Description

`NumDays = days360isda(StartDate, EndDate)` returns a vector or scalar value representing the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

Under this convention all months contain 30 days.

Examples

Example 1. Use this convention to find the number of days in the month of January.

```
StartDate = '1-Jan-2002';  
EndDate = '1-Feb-2002';
```



```
NumDays = days360isda(StartDate, EndDate)
```

```
NumDays =
```

```
30
```

Example 2. Use this convention to find the number of days in February during a leap year.

```
StartDate = '1-Feb-2000';
```

```
EndDate = '1-Mar-2000';
```

```
NumDays = days360isda(StartDate, EndDate)
```

```
NumDays =
```

```
30
```

Example 3. Use this convention to find the number of days in February of a non leap year.

```
StartDate = '1-Feb-2002';
```

```
EndDate = '1-Mar-2002';
```

```
NumDays = days360isda(StartDate, EndDate)
```

```
NumDays =
```

```
30
```

See Also

days360 | days360e | days360psa

days360psa

Days between dates based on 360-day year (Public Securities Association (PSA) compliant)

Syntax

```
NumDays = days360psa(StartDate, EndDate)
```

Arguments

StartDate	Row vector, column vector, or scalar value in serial date number or date string format.
EndDate	Row vector, column vector, or scalar value in serial date number or date string format.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all.

Description

`NumDays = days360psa(StartDate, EndDate)` returns a vector or scalar value representing the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

Under this payment convention all months contain 30 days. In both leap and non-leap years, if the `StartDate` is the last day of February, this day is considered to be day 30 of the month.

Examples

Example 1. Use this convention to find the number of days in between the last day of February and the first day of March during a leap year.

```
StartDate = '29-Feb-2000';  
EndDate = '1-Mar-2000';  
NumDays = days360psa(StartDate, EndDate)
```

```
NumDays =
```

```
1
```

Example 2. Use this convention to find the number of days in between the last day of February and the first day of March during a non-leap year.

```
StartDate = '28-Feb-2002';  
EndDate = '1-Mar-2002';  
NumDays = days360psa(StartDate, EndDate)
```

```
NumDays =
```

```
1
```

As expected, the number of days in both cases is the same. The convention always assumes that the last day of February is the 30th day.

See Also

days360 | days360e | days360isda

days365

Days between dates based on 365-day year

Syntax

```
NumDays = days365(StartDate, EndDate)
```

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all. For example, if **StartDate** is an *n*-row character array of date strings, then **EndDate** must be an *n*-by-1 vector of integers or a single integer. **NumDays** is then an *n*-by-1 vector of date numbers.

Description

`NumDays = days365(StartDate, EndDate)` returns the number of days between dates **StartDate** and **EndDate** based on a 365-day year. (All months contain their actual number of days. February always contains 28 days.) If **EndDate** is earlier than **StartDate**, **NumDays** is negative. Enter dates as serial date numbers or date strings.

Examples

```
NumDays = days365('15-jan-2000', '15-mar-2000')
```

```
NumDays =
```

```
59
```

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];  
NumDays = days365('15-jan-2000', MoreDays)
```

NumDays =

59
90
151

See Also

days360 | daysact | daysdif | wrkdydif | yearfrac

daysact

Actual number of days between dates

Syntax

```
NumDays = daysact(StartDate, EndDate)
```

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	(Optional) Enter as serial date numbers or date strings.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all. For example, if **StartDate** is an n -row character array of date strings, then **EndDate** must be an n -row character array of date strings or a single date. **NumDays** is then an n -by-1 vector of numbers.

Description

`NumDays = daysact(StartDate, EndDate)` returns the actual number of days between two dates. Enter dates as serial date numbers or date strings. **NumDays** is negative if **EndDate** is earlier than **StartDate**.

`NumDays = daysact(StartDate)` returns the actual number of days between the MATLAB base date and **StartDate**. In MATLAB software, the base date 1 is 1-Jan-0000 A.D. See `datenum` for a similar function.

Examples

```
NumDays = daysact('7-sep-2002', '25-dec-2002')
```

```
NumDays =  
    109
```

```
NumDays = daysact('9/7/2002')
```

```
NumDays =  
    731466
```

```
MoreDays = ['09/07/2002'; '10/22/2002'; '11/05/2002'];  
NumDays = daysact(MoreDays, '12/25/2002')
```

```
NumDays =  
    109  
     64  
     50
```

See Also

[datenum](#) | [datevec](#) | [days360](#) | [days365](#) | [daysdif](#)

daysadd

Date away from starting date for any day-count basis

Syntax

NumDays = daysadd(StartDate, NumDays, Basis)

Arguments

StartDate	Start date. Enter as serial date numbers or date strings.
NumDays	Integer number of days from start date. Enter a negative integer for dates before start date.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>

Note When using the 30/360 day-count basis, it is not always possible to find the exact date NumDays number of days away because of a known discontinuity in the method of counting days. A warning is displayed if this occurs.

Description

NumDays = daysadd(StartDate, NumDays, Basis) returns a date NumDays number of days away from StartDate, using the given day-count basis.

Examples

```
NewDate = daysadd('01-Feb-2004', 31)
```

```
NewDate =
```

```
732009
```

```
datestr(NewDate)
```

```
ans =
```

```
03-Mar-2004
```

```
NewDate = daysadd('01-Feb-2004', 31, 1)
```

```
NewDate =
```

```
732008
```

```
datestr(NewDate)
```

```
ans =
```

```
02-Mar-2004
```

References

Stigum, Marcia L. and Franklin Robinson, *Money Market and Bond Calculations*, Richard D. Irwin, 1996, ISBN 1-55623-476-7

See Also
daysdif

daysdif

Days between dates for any day-count basis

Syntax

NumDays = daysdif(StartDate, EndDate, Basis)

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>

Any input argument can contain multiple values, but if so, the other inputs must contain the same number of values or a single value that applies to all. For example, if

`StartDate` is an n-row character array of date strings, then `EndDate` must be an n-row character array of date strings or a single date. `NumDays` is then an n-by-1 vector of numbers.

Description

`NumDays = daysdif(StartDate, EndDate, Basis)` returns the number of days between dates `StartDate` and `EndDate` using the given day-count basis. Enter dates as serial date numbers or date strings. Enter dates as serial date numbers or date strings. The first date (`StartDate`) is not included when determining the number of days between first and last date.

This function is a helper function for the bond pricing and yield functions. It is designed to make the code more readable and to eliminate redundant calls within `if` statements.

Examples

```
NumDays = daysdif('3/1/99', '3/1/00', 1)
```

```
NumDays =  
    360
```

```
MoreDays = ['3/1/2001'; '3/1/2002'; '3/1/2003'];  
NumDays = daysdif('3/1/98', MoreDays)
```

```
NumDays =  
    1096  
    1461  
    1826
```

References

Stigum, Marcia L. and Franklin Robinson, *Money Market and Bond Calculations*, Richard D. Irwin, 1996, ISBN 1-55623-476-7.

See Also

`datenum` | `days360` | `days365` | `daysact` | `daysadd` | `wrkdydif` | `yearfrac`

dec2thirtytwo

Decimal to thirty-second quotation

Syntax

```
[OutNumber, Fractions] = dec2thirtytwo(InNumber, Accuracy)
```

Arguments

InNumber	Input number as a decimal fraction.
Accuracy	(Optional) Rounding. Default = 1, round down to nearest thirty second. Other values are 2 (nearest half), 4 (nearest quarter) and 10 (nearest decile).

Description

[OutNumber, Fractions] = dec2thirtytwo(InNumber, Accuracy) changes a decimal price quotation for a bond or bond future to a fraction with a denominator of 32.

OutNumber is InNumber rounded downward to the closest integer.

Fractions is the fractional part in units of thirty-second with accuracy as prescribed by the input Accuracy.

Examples

Convert Decimal to Thirty-Second Quotation

This example shows two bonds that are quoted with decimal prices of 101.78 and 102.96. These prices are converted to fractions with a denominator of 32.

```
InNumber = [101.78; 102.96];
[OutNumber, Fractions] = dec2thirtytwo(InNumber)
```

OutNumber =

101
102

Fractions =

25
31

See Also

thirtytwo2dec

deprexdb

Fixed declining-balance depreciation schedule

Syntax

Depreciation = deprexdb(Cost, Salvage, Life, Period, Month)

Arguments

Cost	Scalar for the initial value of the asset.
Salvage	Scalar for the salvage value of the asset.
Life	Scalar value for the life of the asset in years.
Period	Scalar integer for the number of years to calculate.
Month	(Optional) Scalar value for the number of months in the first year of asset life. Default = 12.

Description

Depreciation = deprexdb(Cost, Salvage, Life, Period, Month) calculates the fixed declining-balance depreciation for each period.

Examples

Compute the Fixed Declining-Balance Depreciation

This example shows how to calculate the depreciation for the first five years for a car is purchased for \$11,000, with a salvage value of \$1500, and a lifetime of eight years.

```
Depreciation = deprexdb(11000, 1500, 8, 5)
```

```
Depreciation =
```

1.0e+03 *

2.4251 1.8904 1.4737 1.1488 0.8955

See Also

depgendb | deprdv | depsoyd | depstln

depgendb

General declining-balance depreciation schedule

Syntax

```
Depreciation = depgendb(Cost, Salvage, Life, Factor)
```

Arguments

Cost	Cost of the asset.
Salvage	Estimated salvage value of the asset.
Life	Number of periods over which the asset is depreciated.
Factor	Depreciation factor. Factor = 2 uses the double-declining-balance method.

Description

`Depreciation = depgendb(Cost, Salvage, Life, Factor)` calculates the declining-balance depreciation for each period.

Examples

Calculate the Declining-Balance Depreciation

A car is purchased for \$10,000 and is to be depreciated over five years. The estimated salvage value is \$1000. Using the double-declining-balance method, the function calculates the depreciation for each year and returns the remaining depreciable value at the end of the life of the car.

Define the depreciation.

```
Life = 5;  
Salvage = 0;
```

```
Cost = 10000;  
Factor=2;
```

Use `depreddb` to calculate the depreciation.

```
Depreciation = depreddb(10000, 1000, 5, 2)
```

```
Depreciation =
```

```
1.0e+03 *  
4.0000    2.4000    1.4400    0.8640    0.2960
```

The large value returned at the final year is the sum of the depreciation over the life time and is equal to the difference between the `COST` and `Salvage`. The value of the asset in the final year is computed as $(\text{Cost} - \text{Salvage}) - \text{Sum_Depreciation_Upto_Finanl_Year}$.

See Also

`depreddb` | `deprdv` | `depsyod` | `depstln`

deprdv

Remaining depreciable value

Syntax

Value = deprdv(Cost, Salvage, Accum)

Arguments

Cost	Cost of the asset.
Salvage	Salvage value of the asset.
Accum	Accumulated depreciation of the asset for prior periods.

Description

Value = deprdv(Cost, Salvage, Accum) returns the remaining depreciable value for an asset.

Examples

Compute the Remaining Depreciable Value for an Asset

This example shows how to find the remaining depreciable value after six years for the cost of an asset for \$13,000 with a life of 10 years. The salvage value is \$1000.

```
Accum = depstln(13000, 1000, 10) * 6  
Value = deprdv(13000, 1000, 7200)
```

```
Accum =
```

```
7200
```

Value =

4800

See Also

depfixdb | depgendb | depsoyd | depstln

depsyd

Sum of years' digits depreciation

Syntax

Sum = depsoyd(Cost, Salvage, Life)

Arguments

Cost	Cost of the asset.
Salvage	Salvage value of the asset.
Life	Depreciable life of the asset in years.

Description

Sum = depsoyd(Cost, Salvage, Life) calculates the depreciation for an asset using the sum of years' digits method. Sum is a 1-by-Life vector of depreciation values with each element corresponding to a year of the asset's life.

Examples

Compute the Depreciation for an Asset Using the Sum of Years' Digits Method

This example shows how to calculate the depreciation for an asset using the sum of years' digits method where the asset is \$13,000 with a life of 10 years. The salvage value of the asset is \$1000.

```
Sum = depsoyd(13000, 1000, 10)'
```

```
Sum =
```

```
1.0e+03 *
```

2.1818
1.9636
1.7455
1.5273
1.3091
1.0909
0.8727
0.6545
0.4364
0.2182

See Also

depfixdb | deprdv | depgendb | depstln

depstln

Straight-line depreciation schedule

Syntax

Depreciation = depstln(Cost, Salvage, Life)

Arguments

Cost	Cost of the asset.
Salvage	Salvage value of the asset.
Life	Depreciable life of the asset in years.

Description

Depreciation = depstln(Cost, Salvage, Life) calculates straight-line depreciation for an asset.

Examples

Compute the Straight-Line Depreciation for an Asset

This example shows how to calculate the straight-line depreciation for an asset that costs \$13,000 with a life of 10 years. The salvage value of the asset is \$1000.

Depreciation = depstln(13000, 1000, 10)

Depreciation =

1200

See Also

depfixdb | deprdv | depgendb | depsoyd

diff

Differencing

Syntax

```
newfts = diff(oldfts)
```

Description

`diff` computes the differences of the data series in a financial time series object. It returns another time series object containing the difference.

`newfts = diff(oldfts)` computes the difference of all the data in the data series of the object `oldfts` and returns the result in the object `newfts`. `newfts` is a financial time series object containing the same data series (names) as the input `oldfts`.

See Also

`diff`

diffusion

Construct diffusion-rate model components

Syntax

```
DiffusionRate = diffusion(Alpha, Sigma)
```

Class

Diffusion

Description

The `diffusion` constructor specifies 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:

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

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 an NVARs-by-NBROWNS matrix-valued volatility rate function Σ .

The diffusion-rate specification is flexible, and 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

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Alpha	<p>Alpha determines the format of the parameter D. If you specify Alpha as an array, it must be an NVARs-by-1 column vector of exponents. As a deterministic function of time, when Alpha is called with a real-valued scalar time \mathbf{t} as its only input, Alpha must produce an NVARs-by-1 column vector. If you specify Alpha as a function of time and state, it must return an NVARs-by-1 column vector of exponents when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
-------	--

Sigma	<p>Sigma represents the parameter V.</p> <p>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. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARS-by-NBROWNS matrix. If you specify Sigma as a function of time and state, it must return an NVARS-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t.
--------------	--

Note: Although the constructor enforces no restrictions on the signs of these volatility parameters, each parameter is usually specified as a positive value.

Output Arguments

DiffusionRate	<p>Object of class diffusion that encapsulates the composite diffusion-rate specification, with the following displayed parameters:</p> <ul style="list-style-type: none"> • 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.
----------------------	---

Examples

“Overview” on page 17-18

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | sdeddo

disc2zero

Zero curve given discount curve

Syntax

```
[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates, Settle, Compounding, Basis)
```

Arguments

DiscRates	Column vector of discount factors, as decimal fractions. In aggregate, the factors in DiscRates constitute a discount curve for the investment horizon represented by CurveDates .
CurveDates	Column vector of maturity dates (as serial date numbers) that correspond to the discount factors in DiscRates .
Settle	Serial date number that is the common settlement date for the discount rates in DiscRates .
Compounding	(Optional) Output compounding. A scalar that sets the compounding frequency per year for annualizing the output zero rates. Allowed values are:
	1 Annual compounding
	2 Semiannual compounding (default)
	3 Compounding three times per year
	4 Quarterly compounding
	6 Bimonthly compounding
	12 Monthly compounding
	365 Daily compounding
	-1 Continuous compounding
Basis	(Optional) Day-count basis for annualizing the output zero rates. <ul style="list-style-type: none"> 0 = actual/actual (default)

	<ul style="list-style-type: none"> • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
--	--

Description

[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates, Settle, Compounding, Basis) returns a zero curve given a discount curve and its maturity dates.

ZeroRates	Column vector of decimal fractions. In aggregate, the rates in ZeroRates constitute a zero curve for the investment horizon represented by CurveDates . The zero rates are the yields to maturity on theoretical zero-coupon bonds.
CurveDates	Column vector of maturity dates (as serial date numbers) that correspond to the zero rates. This vector is the same as the input vector CurveDates .

Examples

Determine the Zero Curve Given a Discount Curve and Maturity Dates

Given the following discount factors `DiscRates` over a set of maturity dates `CurveDates`, and a settlement date `Settle`:

```
DiscRates = [0.9996  
            0.9947  
            0.9896  
            0.9866  
            0.9826  
            0.9786  
            0.9745  
            0.9665  
            0.9552  
            0.9466];
```

```
CurveDates = [datetime('06-Nov-2000')  
             datetime('11-Dec-2000')  
             datetime('15-Jan-2001')  
             datetime('05-Feb-2001')  
             datetime('04-Mar-2001')  
             datetime('02-Apr-2001')  
             datetime('30-Apr-2001')  
             datetime('25-Jun-2001')  
             datetime('04-Sep-2001')  
             datetime('12-Nov-2001')];
```

```
Settle = datetime('03-Nov-2000');
```

Set daily compounding for the output zero curve, on an actual/365 basis.

```
Compounding = 365;  
Basis = 3;
```

Execute the function `disc2zero` which returns the zero curve `ZeroRates` at the maturity dates `CurveDates`.

```
[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates, ...  
Settle, Compounding, Basis)
```

```
ZeroRates =
```



```
0.0487
0.0510
0.0523
0.0524
0.0530
0.0526
0.0530
0.0532
0.0549
0.0536
```

```
CurveDates =
```

```
730796
730831
730866
730887
730914
730943
730971
731027
731098
731167
```

For readability, **DiscRates** and **ZeroRates** are shown here only to the basis point. However, MATLAB® software computed them at full precision. If you enter **DiscRates** as shown, **ZeroRates** may differ due to rounding.

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

zero2disc

discrate

Bank discount rate of money market security

Syntax

`DiscRate = discrate(Settle, Maturity, Face, Price, Basis)`

Arguments

Settle	Enter as serial date numbers or date strings. Settle must be earlier than Maturity .
Maturity	Enter as serial date numbers or date strings.
Face	Redemption (par, face) value.
Price	Price of the security.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252

For more information, see **basis**.

Description

`DiscRate = discrate(Settle, Maturity, Face, Price, Basis)` finds the bank discount rate of a security. The bank discount rate normalizes by the face value of the security (for example, U. S. Treasury Bills) and understates the true yield earned by investors.

Examples

Compute the Bank Discount Rate of a Security

This example shows how to find the bank discount rate of a security.

```
DiscRate = discrate('12-jan-2000', '25-jun-2000', 100, 97.74, 0)
```

```
DiscRate =  
    0.0501
```

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 1.

See Also

`acrudisc` | `fvdisc` | `prdisc` | `ylddisc`

drift

Construct drift-rate model components

Syntax

```
DriftRate = drift(A, B)
```

Class

Drift

Description

This constructor specifies 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:

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

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 model parameters.

The drift-rate specification is flexible, and 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

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

A	<p>This argument represents the parameter A. If you specify A as an array, it must be an NVARs-by-1 column vector. As a deterministic function of time, when A is called with a real-valued scalar time \mathbf{t} as its only input, A must produce an NVARs-by-1 column vector. If you specify A as a function of time and state, it must return an NVARs-by-1 column vector when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
---	--

B	<p>This argument represents the parameter B. If you specify B as an array, it must be an NVARs-by-NVARs 2-dimensional matrix. As a deterministic function of time, when B is called with a real-valued scalar time t as its only input, B must produce an NVARs-by-NVARs matrix. If you specify B as a function of time and state, it must return an NVARs-by-NVARs column vector when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
----------	---

Output Arguments

DriftRate	<p>Object of class <code>drift</code> that encapsulates the composite drift-rate specification, with the following displayed parameters:</p> <ul style="list-style-type: none"> • 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.
------------------	---

Examples

“Overview” on page 17-18

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

diffusion | sdeddo

ecmlsrmlle

Least-squares regression with missing data

Syntax

```
[Parameters, Covariance, Resid, Info] = ecmlsrmlle(Data, Design,
MaxIterations, TolParam, TolObj, Param0, Covar0, CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are represented as NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use <code>mvnrmlle</code> .)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
MaxIterations	(Optional) Maximum number of iterations for the estimation algorithm. Default value is 100.
TolParam	(Optional) Convergence tolerance for estimation algorithm based on changes in model parameter estimates. Default value is <code>sqrt(eps)</code> which is about 1.0e-8 for double

	precision. The convergence test for changes in model parameters is
	$\ Param_k - Param_{k-1}\ < TolParam \times (1 + \ Param_k\)$
	where Param represents the output Parameters , and iteration $k = 2, 3, \dots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both $TolParam \leq 0$ and $TolObj \leq 0$, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.
TolObj	<p>(Optional) Convergence tolerance for estimation algorithm based on changes in the objective function. Default value is eps # 3/4 which is about 1.0e-12 for double precision. The convergence test for changes in the objective function is</p> $ Obj_k - Obj_{k-1} < TolObj \times (1 + Obj_k)$ <p>for iteration $k = 2, 3, \dots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both $TolParam \leq 0$ and $TolObj \leq 0$, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.</p>
Param0	(Optional) NUMPARAMS -by-1 column vector that contains a user-supplied initial estimate for the parameters of the regression model. Default is a zero vector.
Covar0	<p>(Optional) NUMSERIES-by-NUMSERIES matrix that contains a user-supplied initial or known estimate for the covariance matrix of the regression residuals. Default is an identity matrix.</p> <p>For covariance-weighted least-squares calculations, this matrix corresponds with weights for each series in the regression. The matrix also serves as an initial guess for the residual covariance in the expectation conditional maximization (ECM) algorithm.</p>

CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • 'full' - Default method. Compute the full covariance matrix. • 'diagonal' - Force the covariance matrix to be a diagonal matrix.
-------------	--

Description

[Parameters, Covariance, Resid, Info] = `ecmlsrmle`(Data, Design, MaxIterations, TolParam, TolObj, Param0, Covar0, CovarFormat) estimates a least-squares regression model with missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

for samples $k = 1, \dots, \text{NUMSAMPLES}$.

`ecmlsrmle` estimates a `NUMPARAMS`-by-1 column vector of model parameters called `Parameters`, and a `NUMSERIES`-by-`NUMSERIES` matrix of covariance parameters called `Covariance`.

`ecmlsrmle`(Data, Design) with no output arguments plots the log-likelihood function for each iteration of the algorithm.

To summarize the outputs of `ecmlsrmle`:

- `Parameters` is a `NUMPARAMS`-by-1 column vector of estimates for the parameters of the regression model.
- `Covariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the regression model's residuals. For least-squares models, this estimate may not be a maximum likelihood estimate except under special circumstances.
- `Resid` is a `NUMSAMPLES`-by-`NUMSERIES` matrix of residuals from the regression.

Another output, `Info`, is a structure that contains additional information from the regression. The structure has these fields:

- **Info.Obj** – A variable-extent column vector, with no more than **MaxIterations** elements, that contains each value of the objective function at each iteration of the estimation algorithm. The last value in this vector, **Obj (end)**, is the terminal estimate of the objective function. If you do least-squares, the objective function is the least-squares objective function.
- **Info.PrevParameters** – **NUMPARAMS**-by-1 column vector of estimates for the model parameters from the iteration just prior to the terminal iteration.
- **Info.PrevCovariance** – **NUMSERIES**-by-**NUMSERIES** matrix of estimates for the covariance parameters from the iteration just prior to the terminal iteration.

Notes

If doing covariance-weighted least-squares, **Covar0** should usually be a diagonal matrix. Series with greater influence should have smaller diagonal elements in **Covar0** and series with lesser influence should have larger diagonal elements. Note that if doing CWLS, **Covar0** need not be a diagonal matrix even if **CovarFormat** = 'diagonal'.

You can configure **Design** as a matrix if **NUMSERIES** = 1 or as a cell array if **NUMSERIES** ≥ 1.

- If **Design** is a cell array and **NUMSERIES** = 1, each cell contains a **NUMPARAMS** row vector.
- If **Design** is a cell array and **NUMSERIES** > 1, each cell contains a **NUMSERIES**-by-**NUMPARAMS** matrix.

These points concern how **Design** handles missing data:

- Although **Design** should not have NaN values, ignored samples due to NaN values in **Data** are also ignored in the corresponding **Design** array.
- If **Design** is a 1-by-1 cell array, which has a single **Design** matrix for each sample, no NaN values are permitted in the array. A model with this structure must have **NUMSERIES** ≥ **NUMPARAMS** with **rank(Design{1})** = **NUMPARAMS**.
- **ecmlsrml** is more strict than **mvnrml** about the presence of NaN values in the **Design** array.

Use the estimates in the optional output structure **Info** for diagnostic purposes.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

References

Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

Xiao-Li Meng and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

Joe Sexton and Anders Rygh Swensen, “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

A. P. Dempster, N.M. Laird, and D. B. Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, 1977, pp. 1-37.

See Also

ecmlsrobj | ecmmvnrml | mvnrml

ecmlsrobj

Log-likelihood function for least-squares regression with missing data

Syntax

Objective = ecmlsrobj(Data, Design, Parameters, Covariance)

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are represented as NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use <code>mvnrmlc</code> .)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Parameters	NUMPARAMS-by-1 column vector of estimates for the parameters of the regression model.
Covariance	(Optional) NUMSERIES-by-NUMSERIES matrix that contains a user-supplied estimate for the covariance matrix of the residuals of the regression. Default is an identity matrix.

Description

`Objective = ecmlsrobj(Data, Design, Parameters, Covariance)` computes a least-squares objective function based on current parameter estimates with missing data. `Objective` is a scalar that contains the least-squares objective function.

Notes

`ecmlsrobj` requires that `Covariance` be positive-definite.

Note that

```
ecmlsrobj(Data, Design, Parameters) = ecmmvnrobj(Data, ...  
Design, Parameters, IdentityMatrix)
```

where `IdentityMatrix` is a `NUMSERIES`-by-`NUMSERIES` identity matrix.

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES = 1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES > 1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

See Also

`ecmlsrmlle` | `mvnrmlle` | `mvnrobj`

ecmmvnrfish

Fisher information matrix for multivariate normal regression model

Syntax

```
Fisher = ecmmvnrfish(Data, Design, Covariance, Method,
MatrixFormat, CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are represented as NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use <code>mvnrfish</code> .)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Covariance	NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the residuals of the regression.
Method	<p>(Optional) String that identifies method of calculation for the information matrix:</p> <ul style="list-style-type: none"> • <code>hessian</code> - Default method. Use the expected Hessian matrix of the observed log-likelihood function. This

	<p>method is recommended since the resultant standard errors incorporate the increased uncertainties due to missing data.</p> <ul style="list-style-type: none"> • fisher - Use the Fisher information matrix.
MatrixFormat	<p>(Optional) String that identifies parameters to be included in the Fisher information matrix:</p> <ul style="list-style-type: none"> • full - Default format. Compute the full Fisher information matrix for both model and covariance parameter estimates. • paramonly - Compute only components of the Fisher information matrix associated with the model parameter estimates.
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • 'full' - Default method. The covariance matrix is a full matrix. • 'diagonal' - The covariance matrix is a diagonal matrix.

Description

`Fisher = ecmmvnrfish(Data, Design, Covariance, Method, MatrixFormat, CovarFormat)` computes a Fisher information matrix based on current maximum likelihood or least-squares parameter estimates that account for missing data.

`Fisher` is a `NUMPARAMS`-by-`NUMPARAMS` Fisher information matrix or Hessian matrix. The size of `NUMPARAMS` depends on `MatrixFormat` and on current parameter estimates. If `MatrixFormat = 'full'`,

$$\text{NUMPARAMS} = \text{NUMSERIES} * (\text{NUMSERIES} + 3) / 2$$

If `MatrixFormat = 'paramonly'`,

$$\text{NUMPARAMS} = \text{NUMSERIES}$$

Note `ecmmvnrfish` operates slowly if you calculate the full Fisher information matrix.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

See Also

ecmmle | ecnstd

ecmmvnrml

Multivariate normal regression with missing data

Syntax

```
[Parameters, Covariance, Resid, Info] = ecmmvnrml(Data, Design,
MaxIterations, TolParam, TolObj, Param0, Covar0, CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are represented as NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use <code>mvnrml</code> .)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
MaxIterations	(Optional) Maximum number of iterations for the estimation algorithm. Default value is 100.
TolParam	(Optional) Convergence tolerance for estimation algorithm based on changes in model parameter estimates. Default value is <code>sqrt(eps)</code> which is about 1.0e-8 for double

	<p>precision. The convergence test for changes in model parameters is</p> $\ Param_k - Param_{k-1}\ < TolParam \times (1 + \ Param_k\)$ <p>where Param represents the output Parameters, and iteration $k = 2, 3, \dots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both $TolParam \leq 0$ and $TolObj \leq 0$, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.</p>
TolObj	<p>(Optional) Convergence tolerance for estimation algorithm based on changes in the objective function. Default value is eps # 3/4 which is about 1.0e-12 for double precision. The convergence test for changes in the objective function is</p> $ Obj_k - Obj_{k-1} < TolObj \times (1 + Obj_k)$ <p>for iteration $k = 2, 3, \dots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both $TolParam \leq 0$ and $TolObj \leq 0$, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.</p>
Param0	<p>(Optional) NUMPARAMS-by-1 column vector that contains a user-supplied initial estimate for the parameters of the regression model.</p>
Covar0	<p>(Optional) NUMSERIES-by-NUMSERIES matrix that contains a user-supplied initial or known estimate for the covariance matrix of the regression residuals.</p>
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • 'full' - Default method. Compute the full covariance matrix. • 'diagonal' - Force the covariance matrix to be a diagonal matrix.

Description

`[Parameters, Covariance, Resid, Info] = ecmmvnrml(Data, Design, MaxIterations, TolParam, TolObj, Param0, Covar0, CovarFormat)` estimates a multivariate normal regression model with missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

for samples $k = 1, \dots, \text{NUMSAMPLES}$.

`ecmmvnrml` estimates a `NUMPARAMS`-by-1 column vector of model parameters called `Parameters`, and a `NUMSERIES`-by-`NUMSERIES` matrix of covariance parameters called `Covariance`.

`ecmmvnrml(Data, Design)` with no output arguments plots the log-likelihood function for each iteration of the algorithm.

To summarize the outputs of `ecmmvnrml`:

- `Parameters` is a `NUMPARAMS`-by-1 column vector of estimates for the parameters of the regression model.
- `Covariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the regression model's residuals.
- `Resid` is a `NUMSAMPLES`-by-`NUMSERIES` matrix of residuals from the regression. For any missing values in `Data`, the corresponding residual is the difference between the conditionally imputed value for `Data` and the model, that is, the imputed residual.

Note: The covariance estimate `Covariance` cannot be derived from the residuals.

Another output, `Info`, is a structure that contains additional information from the regression. The structure has these fields:

- `Info.Obj` – A variable-extent column vector, with no more than `MaxIterations` elements, that contains each value of the objective function at each iteration of the estimation algorithm. The last value in this vector, `Obj(end)`, is the terminal estimate of the objective function. If you do maximum likelihood estimation, the objective function is the log-likelihood function.
- `Info.PrevParameters` – `NUMPARAMS`-by-1 column vector of estimates for the model parameters from the iteration just prior to the terminal

`iteration.nfo.PrevCovariance` – NUMSERIES-by-NUMSERIES matrix of estimates for the covariance parameters from the iteration just prior to the terminal iteration.

Notes

`ecmmvnrml` does not accept an initial parameter vector, since the parameters are estimated directly from the first iteration onward.

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES = 1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES > 1`, each cell contains a NUMSERIES-by-`NUMPARAMS` matrix.

These points concern how `Design` handles missing data:

- Although `Design` should not have NaN values, ignored samples due to NaN values in `Data` are also ignored in the corresponding `Design` array.
- If `Design` is a 1-by-1 cell array, which has a single `Design` matrix for each sample, no NaN values are permitted in the array. A model with this structure must have `NUMSERIES ≥ NUMPARAMS` with `rank(Design{1}) = NUMPARAMS`.
- `ecmmvnrml` is more strict than `mvnrml` about the presence of NaN values in the `Design` array.

Use the estimates in the optional output structure `Info` for diagnostic purposes.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

References

Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

Xiao-Li Meng and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

Joe Sexton and Anders Rygh Swensen, “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

A. P. Dempster, N.M. Laird, and D. B. Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, 1977, pp. 1-37.

See Also

`ecmmvnrobj` | `mvnrml`

ecmmvnrobj

Log-likelihood function for multivariate normal regression with missing data

Syntax

Objective = ecmmvnrobj(Data, Design, Parameters, Covariance, CovarFormat)

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are represented as NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use <code>mvnrmlc</code> .)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Parameters	NUMPARAMS-by-1 column vector of estimates for the parameters of the regression model.
Covariance	NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the residuals of the regression.
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • 'full' - Default method. The covariance matrix is a full matrix.

- 'diagonal' - The covariance matrix is a diagonal matrix.

Description

`Objective = ecmmvnrobj(Data, Design, Parameters, Covariance, CovarFormat)` computes a log-likelihood function based on current maximum likelihood parameter estimates with missing data. `Objective` is a scalar that contains the least-squares objective function.

Notes

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES = 1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES > 1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

See Also

`ecmmvnrmle` | `mvnrmle` | `mvnrobj`

ecmmvnrstd

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters, StdCovariance] = ecmmvnrstd(Data, Design, Covariance, Method, CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are represented as NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use <code>mvnrstd</code> .)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Covariance	NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the regression residuals.
Method	<p>(Optional) String that identifies method of calculation for the information matrix:</p> <ul style="list-style-type: none"> • <code>hessian</code> - Default method. Use the expected Hessian matrix of the observed log-likelihood function. This method is recommended since the resultant standard errors incorporate the increased uncertainties due to missing data.

	<ul style="list-style-type: none"> • <code>fisher</code> - Use the Fisher information matrix.
<code>CovarFormat</code>	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • <code>'full'</code> - Default method. The covariance matrix is a full matrix. • <code>'diagonal'</code> - The covariance matrix is a diagonal matrix.

Description

`[StdParameters, StdCovariance] = ecmmvnrstd(Data, Design, Covariance, Method, CovarFormat)` evaluates standard errors for a multivariate normal regression model with missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

for samples $k = 1, \dots, \text{NUMSAMPLES}$.

`ecmmvnrstd` computes two outputs:

- `StdParameters` is a `NUMPARAMS`-by-1 column vector of standard errors for each element of `Parameters`, the vector of estimated model parameters.
- `StdCovariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of standard errors for each element of `Covariance`, the matrix of estimated covariance parameters.

Note `ecmmvnrstd` operates slowly when you calculate the standard errors associated with the covariance matrix `Covariance`.

Notes

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES = 1`, each cell contains a `NUMPARAMS` row vector.

- If `Design` is a cell array and `NUMSERIES > 1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

References

Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

See Also

`ecmmvnrmlc` | `ecmmvnrstd` | `mvnrmlc`

ecmnfish

Fisher information matrix

Syntax

```
Fisher = ecmnfish(Data, Covariance, InvCovariance, MatrixFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix of observed multivariate normal data
Covariance	NUMSERIES-by-NUMSERIES matrix with covariance estimate of Data
InvCovariance	(Optional) Inverse of covariance matrix: <code>inv(Covariance)</code>
MatrixFormat	(Optional) String that identifies parameters included in the Fisher information matrix. If <code>MatrixFormat = []</code> or <code>'</code> , the default method <code>full</code> is used. The parameter choices are <ul style="list-style-type: none"> <code>full</code> — (Default) Compute full Fisher information matrix. <code>meanonly</code> — Compute only components of the Fisher information matrix associated with the mean.

Description

`Fisher = ecmnfish(Data, Covariance, InvCovariance, MatrixFormat)` computes a `NUMPARAMS`-by-`NUMPARAMS` Fisher information matrix based on current parameter estimates, where

$$\text{NUMPARAMS} = \text{NUMSERIES} * (\text{NUMSERIES} + 3) / 2$$

if `MatrixFormat = 'full'` and

$$\text{NUMPARAMS} = \text{NUMSERIES}$$

```
if MatrixFormat = 'meanonly'.
```

The data matrix has NaNs for missing observations. The multivariate normal model has

```
NUMPARAMS = NUMSERIES + NUMSERIES*(NUMSERIES + 1)/2
```

distinct parameters. Therefore, the full Fisher information matrix is of size NUMPARAMS-by-NUMPARAMS. The first NUMSERIES parameters are estimates for the mean of the data in `Mean`, and the remaining NUMSERIES*(NUMSERIES + 1)/2 parameters are estimates for the lower-triangular portion of the covariance of the data in `Covariance`, in row-major order.

If `MatrixFormat = 'meanonly'`, the number of parameters is reduced to NUMPARAMS = NUMSERIES, where the Fisher information matrix is computed for the mean parameters only. In this format, the routine executes fastest.

This routine expects the inverse of the covariance matrix as an input. If you do not pass in the inverse, the routine computes it. You can obtain an approximation for the lower-bound standard errors of estimation of the parameters from

```
Stderr = (1.0/sqrt(NumSamples)) .* sqrt(diag(inv(Fisher)));
```

Because of missing information, these standard errors may be smaller than the estimated standard errors derived from the expected Hessian matrix. To see the difference, compare to standard errors calculated with `ecmnhess`.

See Also

`ecmnhess` | `ecmnmle`

ecmhess

Hessian of negative log-likelihood function

Syntax

```
Hessian = ecmhess(Data, Covariance, InvCovariance, MatrixFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix of observed multivariate normal data
Covariance	NUMSERIES-by-NUMSERIES matrix with covariance estimate of Data
InvCovariance	(Optional) Inverse of covariance matrix: <code>inv(Covariance)</code>
MatrixFormat	(Optional) String that identifies parameters included in the Hessian matrix. If <code>MatrixFormat = []</code> or <code>'</code> , the default method <code>full</code> is used. The parameter choices are <ul style="list-style-type: none"> <code>full</code> — (Default) Compute full Hessian matrix. <code>meanonly</code> — Compute only components of the Hessian matrix associated with the mean.

Description

`Hessian = ecmhess(Data, Covariance, InvCovariance, MatrixFormat)` computes a `NUMPARAMS`-by-`NUMPARAMS` Hessian matrix of the observed negative log-likelihood function based upon current parameter estimates, where

```
NUMPARAMS = NUMSERIES*(NUMSERIES + 3)/2
```

```
if MatrixFormat = 'full' and
```

```
NUMPARAMS = NUMSERIES
```

```
if MatrixFormat = 'meanonly'.
```

This routine is very slow for `NUMSERIES > 10` or `NUMSAMPLES > 1000`.

The data matrix has NaNs for missing observations. The multivariate normal model has

`NUMPARAMS = NUMSERIES + NUMSERIES*(NUMSERIES + 1)/2`

distinct parameters. Therefore, the full Hessian is a `NUMPARAMS`-by-`NUMPARAMS` matrix.

The first `NUMSERIES` parameters are estimates for the mean of the data in `Mean` and the remaining `NUMSERIES*(NUMSERIES + 1)/2` parameters are estimates for the lower-triangular portion of the covariance of the data in `Covariance`, in row-major order.

If `MatrixFormat = 'meanonly'`, the number of parameters is reduced to `NUMPARAMS = NUMSERIES`, where the Hessian is computed for the mean parameters only. In this format, the routine executes fastest.

This routine expects the inverse of the covariance matrix as an input. If you do not pass in the inverse, the routine computes it.

The equation

```
Stderr = (1.0/sqrt(NumSamples)) .* sqrt(diag(inv(Hessian)));
```

provides an approximation for the observed standard errors of estimation of the parameters.

Because of the additional uncertainties introduced by missing information, these standard errors may be larger than the estimated standard errors derived from the Fisher information matrix. To see the difference, compare to standard errors calculated from `ecmfish`.

See Also

`ecmfish` | `ecmnml`

ecmninit

Initial mean and covariance

Syntax

```
[Mean, Covariance] = ecmninit(Data, InitMethod)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are indicated by NaNs.
InitMethod	(Optional) String that identifies one of three defined initialization methods to compute initial estimates for the mean and covariance of the data. If <code>InitMethod = []</code> or <code>''</code> , the default method <code>nanskip</code> is used. The initialization methods are <ul style="list-style-type: none">• <code>nanskip</code> — (Default) Skip all records with NaNs.• <code>twostage</code> — Estimate mean. Fill NaNs with the mean. Then estimate the covariance.• <code>diagonal</code> — Form a diagonal covariance.

Description

`[Mean, Covariance] = ecmninit(Data, InitMethod)` creates initial mean and covariance estimates for the function `ecmmle`. `Mean` is a NUMSERIES-by-1 column vector estimate for the mean of `Data`. `Covariance` is a NUMSERIES-by-NUMSERIES matrix estimate for the covariance of `Data`.

More About

Algorithms

Model

The general model is

$$Z \sim N(\text{Mean}, \text{Covariance}),$$

where each row of `Data` is an observation of Z .

Each observation of Z is assumed to be iid (independent, identically distributed) multivariate normal, and missing values are assumed to be missing at random (MAR).

Initialization Methods

This routine has three initialization methods that cover most cases, each with its advantages and disadvantages.

nanskip

The `nanskip` method works well with small problems (fewer than 10 series or with monotone missing data patterns). It skips over any records with NaNs and estimates initial values from complete-data records only. This initialization method tends to yield fastest convergence of the ECM algorithm. This routine switches to the `twostage` method if it determines that significant numbers of records contain NaN.

twostage

The `twostage` method is the best choice for large problems (more than 10 series). It estimates the mean for each series using all available data for each series. It then estimates the covariance matrix with missing values treated as equal to the mean rather than as NaNs. This initialization method is quite robust but tends to result in slower convergence of the ECM algorithm.

diagonal

The `diagonal` method is a worst-case approach that deals with problematic data, such as disjoint series and excessive missing data (more than 33% missing data). Of the three initialization methods, this method causes the slowest convergence of the ECM algorithm.

See Also

`ecmmle`

ecmmle

Mean and covariance of incomplete multivariate normal data

Syntax

```
[Mean, Covariance] = ecmmle(Data, InitMethod, MaxIterations,  
Tolerance, Mean0, Covar0)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are indicated by NaNs. A sample is also called an <i>observation</i> or a <i>record</i> .
InitMethod	(Optional) String that identifies one of three defined initialization methods to compute initial estimates for the mean and covariance of the data. If <code>InitMethod = []</code> or <code>''</code> , the default method <code>nanskip</code> is used. The initialization methods are: <ul style="list-style-type: none"> <code>nanskip</code> — (Default) Skip all records with NaNs. <code>twostage</code> — Estimate mean. Fill NaNs with mean. Then estimate covariance. <code>diagonal</code> — Form a diagonal covariance. <hr/> <p>Note If you supply <code>Mean0</code> and <code>Covar0</code>, <code>InitMethod</code> is not executed.</p>
MaxIterations	(Optional) Maximum number of iterations for the expectation conditional maximization (ECM) algorithm. Default = 50.
Tolerance	(Optional) Convergence tolerance for the ECM algorithm (Default = $1.0e-8$.) If <code>Tolerance</code> ≤ 0 , perform maximum iterations specified by <code>MaxIterations</code> and do not evaluate

	the objective function at each step unless in display mode, as described below.
Mean0	(Optional) Initial NUMSERIES-by-1 column vector estimate for the mean. If you leave Mean0 unspecified ([]), the method specified by InitMethod is used. If you specify Mean0, you must also specify Covar0.
Covar0	(Optional) Initial NUMSERIES-by-NUMSERIES matrix estimate for the covariance, where the input matrix must be positive-definite. If you leave Covar0 unspecified ([]), the method specified by InitMethod is used. If you specify Covar0, you must also specify Mean0.

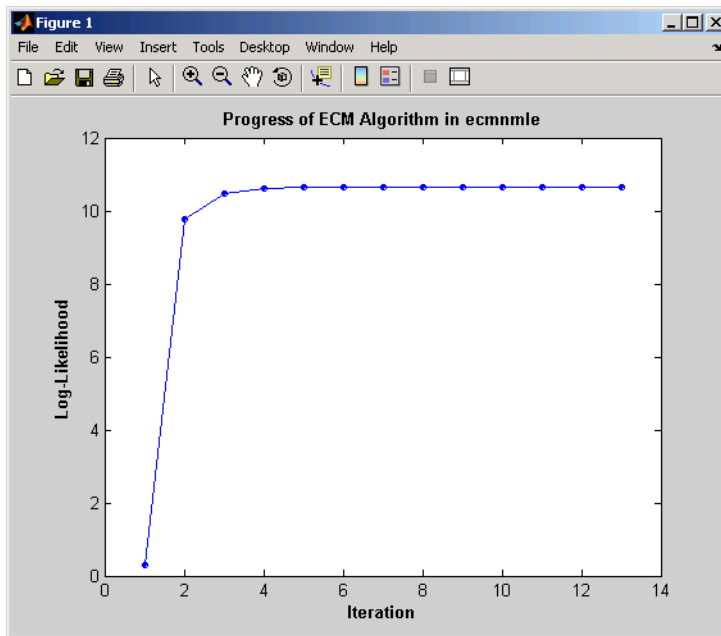
Description

`[Mean, Covariance] = ecmmle(Data, InitMethod, MaxIterations, Tolerance, Mean0, Covar0)` estimates the mean and covariance of a data set. If the data set has missing values, this routine implements the ECM algorithm of Meng and Rubin [2] with enhancements by Sexton and Swensen [3]. ECM stands for *expectation conditional maximization*, a conditional maximization form of the EM algorithm of Dempster, Laird, and Rubin [4].

This routine has two operational modes.

Display Mode

With no output arguments, this mode displays the convergence of the ECM algorithm. It estimates and plots objective function values for each iteration of the ECM algorithm until termination, as shown in the following plot.



Display mode can determine `MaxIter` and `Tolerance` values or serve as a diagnostic tool. The objective function is the negative log-likelihood function of the observed data and convergence to a maximum likelihood estimate corresponds with minimization of the objective.

Estimation Mode

With output arguments, this mode estimates the mean and covariance via the ECM algorithm.

Examples

To see an example of how to use `ecmmle`, run the program `ecmguidemo`.

More About

Algorithms

Model

The general model is

$$Z \sim N(\text{Mean}, \text{Covariance}),$$

where each row of `Data` is an observation of Z .

Each observation of Z is assumed to be iid (independent, identically distributed) multivariate normal, and missing values are assumed to be missing at random (MAR). See Little and Rubin [1] for a precise definition of MAR.

This routine estimates the mean and covariance from given data. If data values are missing, the routine implements the ECM algorithm of Meng and Rubin [2] with enhancements by Sexton and Swensen [3].

If a record is empty (every value in a sample is `NaN`), this routine ignores the record because it contributes no information. If such records exist in the data, the number of nonempty samples used in the estimation is $\leq \text{NumSamples}$.

The estimate for the covariance is a biased maximum likelihood estimate (MLE). To convert to an unbiased estimate, multiply the covariance by `Count/(Count - 1)`, where `Count` is the number of nonempty samples used in the estimation.

Requirements

This routine requires consistent values for `NUMSAMPLES` and `NUMSERIES` with `NUMSAMPLES > NUMSERIES`. It must have enough nonmissing values to converge. Finally, it must have a positive-definite covariance matrix. Although the references provide some necessary and sufficient conditions, general conditions for existence and uniqueness of solutions in the missing-data case do not exist. The main failure mode is an ill-conditioned covariance matrix estimate. Nonetheless, this routine works for most cases that have less than 15% missing data (a typical upper bound for financial data).

Initialization Methods

This routine has three initialization methods that cover most cases, each with its advantages and disadvantages. The ECM algorithm always converges to a minimum of the observed negative log-likelihood function. If you override the initialization methods, you must ensure that the initial estimate for the covariance matrix is positive-definite.

The following is a guide to the supported initialization methods.

nanskip

The `nanskip` method works well with small problems (fewer than 10 series or with monotone missing data patterns). It skips over any records with NaNs and estimates initial values from complete-data records only. This initialization method tends to yield fastest convergence of the ECM algorithm. This routine switches to the `twostage` method if it determines that significant numbers of records contain NaN.

twostage

The `twostage` method is the best choice for large problems (more than 10 series). It estimates the mean for each series using all available data for each series. It then estimates the covariance matrix with missing values treated as equal to the mean rather than as NaNs. This initialization method is quite robust but tends to result in slower convergence of the ECM algorithm.

diagonal

The `diagonal` method is a worst-case approach that deals with problematic data, such as disjoint series and excessive missing data (more than 33% of data missing). Of the three initialization methods, this method causes the slowest convergence of the ECM algorithm. If problems occur with this method, use `display` mode to examine convergence and modify either `MaxIterations` or `Tolerance`, or try alternative initial estimates with `Mean0` and `Covar0`. If all else fails, try

```
Mean0 = zeros(NumSeries);  
Covar0 = eye(NumSeries,NumSeries);
```

Given estimates for mean and covariance from this routine, you can estimate standard errors with the companion routine `ecmnstd`.

Convergence

The ECM algorithm does not work for all patterns of missing values. Although it works in most cases, it can fail to converge if the covariance becomes singular. If this occurs, plots of the log-likelihood function tend to have a constant upward slope over many iterations as the log of the negative determinant of the covariance goes to zero. In some cases, the objective fails to converge due to machine precision errors. No general theory of missing data patterns exists to determine these cases. An example of a known failure occurs when two time series are proportional wherever both series contain nonmissing values.

References

- [1] Little, Roderick J. A. and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.
- [2] Meng, Xiao-Li and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.
- [3] Sexton, Joe and Anders Rygh Swensen, “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.
- [4] Dempster, A. P., N. M. Laird, and Donald B. Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, 1977, pp. 1-37.

See Also

ecmfish | ecmnhess | ecmninit | ecmnobj | ecmnstd

ecmnojb

Multivariate normal negative log-likelihood function

Syntax

`Objective = ecmnojb(Data, Mean, Covariance, CholCovariance)`

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix of observed multivariate normal data
Mean	NUMSERIES-by-1 column vector with mean estimate of Data
Covariance	NUMSERIES-by-NUMSERIES matrix with covariance estimate of Data
CholCovariance	(Optional) Cholesky decomposition of covariance matrix: chol(Covariance)

Description

`Objective = ecmnojb(Data, Mean, Covariance, CholCovariance)` computes the value of the observed negative log-likelihood function over the data given current estimates for the mean and covariance of the data.

The data matrix has NaNs for missing observations. The inputs `Mean` and `Covariance` are current estimates for model parameters.

This routine expects the Cholesky decomposition of the covariance matrix as an input. The routine computes the Cholesky decomposition if you do not explicitly specify it.

See Also

chol | ecmnmle

ecmnstd

Standard errors for mean and covariance of incomplete data

Syntax

```
[StdMean, StdCovariance] = ecmnstd(Data, Mean, Covariance, Method)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are indicated by NaNs.
Mean	NUMSERIES-by-1 column vector of maximum-likelihood parameter estimates for the mean of Data using the expectation conditional maximization (ECM) algorithm
Covariance	NUMSERIES-by-NUMSERIES matrix of maximum-likelihood covariance estimates for the covariance of Data using the ECM algorithm
Method	(Optional) String indicating method of estimation for standard error calculations. The methods are: <ul style="list-style-type: none"> • <code>hessian</code> — (Default) Hessian of the observed negative log-likelihood function. • <code>fisher</code> — Fisher information matrix.

Description

`[StdMean, StdCovariance] = ecmnstd(Data, Mean, Covariance, Method)` computes standard errors for mean and covariance of incomplete data.

`StdMean` is a NUMSERIES-by-1 column vector of standard errors of estimates for each element of the mean vector `Mean`.

`StdCovariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of standard errors of estimates for each element of the covariance matrix `Covariance`.

Use this routine after estimating the mean and covariance of `Data` with `ecmnml`. If the mean and distinct covariance elements are treated as the parameter θ in a complete-data maximum-likelihood estimation, then as the number of samples increases, θ attains asymptotic normality such that

$$\theta - E[\theta] \sim N(0, I^{-1}(\theta)),$$

where $E[\theta]$ is the mean and $I(\theta)$ is the Fisher information matrix.

With missing data, the Hessian $H(\theta)$ is a good approximation for the Fisher information (which can only be approximated when data is missing).

It is usually advisable to use the default `Method` since the resultant standard errors incorporate the increased uncertainty due to missing data. In particular, standard errors calculated with the Hessian are generally larger than standard errors calculated with the Fisher information matrix.

Note This routine is very slow for `NUMSERIES` > 10 or `NUMSAMPLES` > 1000.

See Also

`ecmnml`

effrr

Effective rate of return

Syntax

Return = effrr(Rate, NumPeriods)

Arguments

Rate	Annual percentage rate. Enter as a decimal fraction.
NumPeriods	Number of compounding periods per year, an integer.

Description

Return = effrr(Rate, NumPeriods) calculates the annual effective rate of return. Compounding continuously returns Return equivalent to $(e^{\text{Rate}} - 1)$.

Examples

Compute the Annual Effective Rate of Return

This example shows how to find the effective annual rate of return based on an annual percentage rate of 9% compounded monthly.

Return = effrr(0.09, 12)

Return =

0.0938

See Also

nomrr

elpm

Compute expected lower partial moments for normal asset returns

Syntax

```
elpm(Mean, Sigma)
```

```
elpm(Mean, Sigma, MAR)
```

```
elpm(Mean, Sigma, MAR, Order)
```

```
moment = elpm(Mean, Sigma, MAR, Order)
```

Arguments

Mean	NUMSERIES vector with mean returns for a collection of NUMSERIES assets.
Sigma	NUMSERIES vector with standard deviation of returns for a collection of NUMSERIES assets.
MAR	(Optional) Scalar minimum acceptable return (default MAR = 0). This is a cutoff level of return such that all returns above MAR contribute nothing to the lower partial moment.
Order	(Optional) Either a scalar or a NUMORDERS vector of nonnegative integer moment orders. If no order specified, default Order = 0, which is the shortfall probability. This function will not work for negative or noninteger orders.

Description

Given NUMSERIES asset returns with a vector of mean returns in a NUMSERIES vector **Mean**, a vector of standard deviations of returns in a NUMSERIES vector **Sigma**, a scalar minimum acceptable return **MAR**, and one or more nonnegative integer moment orders in a NUMORDERS vector **Order**, compute expected lower partial moments (**elpm**) relative to **MAR** for each asset in a NUMORDERS-by-NUMSERIESmatrix **Moment**.

The output, `Moment`, is a `NUMORDERS`-by-`NUMSERIES` matrix of expected lower partial moments with `NUMORDERS` Orders and `NUMSERIES` series, that is, each row contains expected lower partial moments for a given order.

Note: To compute upper partial moments, just reverse the signs of both the input `Mean` and `MAR` (do not reverse the signs of either `Sigma` or the output). This function computes expected lower partial moments with the mean and standard deviation of normally distributed asset returns. To compute sample lower partial moments from asset returns which have no distributional assumptions, use `lpm`.

Examples

See “Expected Lower Partial Moments” on page 7-13.

See Also

`lpm`

emaxdrawdown

Compute expected maximum drawdown for Brownian motion

Syntax

EDD = emaxdrawdown(Mu, Sigma, T)

Arguments

Mu	Scalar. Drift term of a Brownian motion with drift.
Sigma	Scalar. Diffusion term of a Brownian motion with drift.
T	A time period of interest or a vector of times.

Description

EDD = emaxdrawdown(Mu, Sigma, T) computes the expected maximum drawdown for a Brownian motion for each time period in T using the following equation:

$$dX(t) = \mu dt + \sigma dW(t).$$

If the Brownian motion is geometric with the stochastic differential equation

$$dS(t) = \mu_0 S(t) dt + \sigma_0 S(t) dW(t)$$

then use Ito's lemma with $X(t) = \log(S(t))$ such that

$$\begin{aligned}\mu &= \mu_0 - 0.5\sigma_0^2, \\ \sigma &= \sigma_0\end{aligned}$$

converts it to the form used here.

The output argument `ExpDrawdown` is computed using an interpolation method. Values are accurate to a fraction of a basis point. Maximum drawdown is nonnegative since it is the change from a peak to a trough.

Note To compare the actual results from `maxdrawdown` with the expected results of `emaxdrawdown`, set the `Format` input argument of `maxdrawdown` to either of the nondefault values ('arithmetic' or 'geometric'). These are the only two formats `emaxdrawdown` supports.

Examples

See “Expected Maximum Drawdown” on page 7-18.

References

Malik Magdon-Ismail, Amir F. Atiya, Amrit Pratap, and Yaser S. Abu-Mostafa, “On the Maximum Drawdown of a Brownian Motion,” *Journal of Applied Probability*, Volume 41, Number 1, March 2004, pp. 147-161.

See Also

`maxdrawdown`

end

Last date entry

Syntax

end

Description

end returns the index to the last date entry in a financial time series object.

Examples

Consider a financial time series object called `fts`:

```
fts =
```

```
desc: DJI30MAR94.dat
freq: Daily (1)

'dates: (20)'      'Open: (20)'
'04-Mar-1994'     [      3830.9]
'07-Mar-1994'     [      3851.7]
'08-Mar-1994'     [      3858.5]
'09-Mar-1994'     [       3854]
'10-Mar-1994'     [      3852.6]
'11-Mar-1994'     [      3832.6]
'14-Mar-1994'     [      3870.3]
'16-Mar-1994'     [       3851]
'17-Mar-1994'     [      3853.6]
'18-Mar-1994'     [      3865.4]
'21-Mar-1994'     [      3878.4]
'22-Mar-1994'     [      3865.7]
'23-Mar-1994'     [      3868.9]
'24-Mar-1994'     [      3849.9]
'25-Mar-1994'     [      3827.1]
'28-Mar-1994'     [      3776.5]
```

```
'29-Mar-1994' [ 3757.2]
'30-Mar-1994' [ 3688.4]
'31-Mar-1994' [ 3639.7]
```

The command `fts(15:end)` returns

```
ans =
```

```
desc: DJI30MAR94.dat
freq: Daily (1)
```

```
'dates: (6)'      'Open: (6)'
```

'24-Mar-1994'	[3849.9]
'25-Mar-1994'	[3827.1]
'28-Mar-1994'	[3776.5]
'29-Mar-1994'	[3757.2]
'30-Mar-1994'	[3688.4]
'31-Mar-1994'	[3639.7]

See Also

`subsasgn` | `end` | `subsref`

eomdate

Last date of month

Syntax

```
DayMonth = eomdate(Date)
```

```
DayMonth = eomdate(Year, Month)
```

Description

`DayMonth = eomdate(Date)` returns the serial date number of the last date of the month for the given `Date`. Enter `Date` as a four-digit integer or a date string.

`DayMonth = eomdate(Year, Month)` returns the serial date number of the last date of the month for the given year and month. Enter `Year` as a four-digit integer; enter `Month` as an integer from 1 through 12.

Either input argument can contain multiple values, but if so, the other input must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-*n* vector of integers, then `Month` must be a 1-by-*n* vector of integers or a single integer. `DayMonth` is then a 1-by-*n* vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings or `datenum` to convert date and time to serial date number.

Examples

```
DayMonth = eomdate(2001, 2)
```

```
DayMonth =  
    730910
```

```
datestr(DayMonth)
```

```
ans =  
28-Feb-2001
```

```
Year = [2002 2003 2004 2005];
```

```
DayMonth = eomdate(Year, 2)
```

```
DayMonth =  
    731275    731640    732006    732371
```

```
datestr(DayMonth)
```

```
ans =  
28-Feb-2002  
28-Feb-2003  
29-Feb-2004  
28-Feb-2005
```

See Also

day | eomday | lbusdate | month | year

eq (fts)

Multiple financial times series object equality

Syntax

```
tsobj_1 == tsobj_2
```

```
iseq = eq(tsobj_1, tsobj_2)
```

Arguments

tsobj_1	Financial time series object.
tsobj_2	Financial time series object.

Description

`tsobj_1 == tsobj_2` returns True (1) if both financial time series objects have the same dates, frequencies, data series names, and data values. Otherwise, `eq` returns False (0).

Note: The data series names are case-sensitive, but do not have to be in the same order within each object.

Examples

Determine Multiple Financial Times Series Object Equality

This example shows how to determine if multiple financial times series objects are equal.

```
load disney
dis == dis
```

ans =

1

See Also

isequal

estimateAssetMoments

Class: Portfolio

Estimate mean and covariance of asset returns from data

Syntax

```
obj = estimateAssetMoments(obj,AssetReturns)
obj = estimateAssetMoments(obj,AssetReturns,varargin)
```

Description

`obj = estimateAssetMoments(obj,AssetReturns)` to estimate the mean and covariance of asset returns from data.

`obj = estimateAssetMoments(obj,AssetReturns,varargin)` to estimate mean and covariance of asset returns from data with additional options specified by one or more Name,Value pair arguments.

Tips

You can also use dot notation to estimate the mean and covariance of asset returns from data.

```
obj = obj.estimateAssetMoments(AssetReturns, varargin);
```

Input Arguments

obj

Portfolio object [Portfolio].

AssetReturns

Either a matrix or `fints` object that contains asset data that can be converted to asset returns [NumSamples-by-NumAssets matrix].

Note: This method estimates the mean and covariance of asset returns from either price or return data. Data can reside in a `NumSamples`-by-`NumAssets` matrix of `NumSamples` prices or returns. This matrix represents a given periodicity for a collection of `NumAssets` assets or a `fints` object with `NumSamples` observations and `NumAssets` time series.

Name-Value Pair Arguments for `varargin`

Specify optional comma-separated pairs of `Name`,`Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`,`Value1`, . . . ,`NameN`,`ValueN`.

'DataFormat'

If the input data are prices, these values can be converted into returns with the `DataFormat` flag, where the default format is assumed to be returns. Be careful using price data because portfolio optimization requires total returns and not simply price returns.

Acceptable values for `DataFormat` are:

- 'Returns' — Data in `AssetReturns` contains asset total returns.
- 'Prices' — Data in `AssetReturns` contains asset total return prices.

Default: 'Returns'

'MissingData'

To handle time series with missing data (indicated with `NaN` values), the `MissingData` flag either uses the ECM algorithm to obtain maximum likelihood estimates in the presences of `NaN` values or excludes samples with `NaN` values. Since the default is `false`, it is necessary to specify `MissingData` as `true` to use the ECM algorithm.

Acceptable values for `MissingData` are:

- `false` — Do not use ECM algorithm to handle `NaN` values (just exclude `NaN` values).
- `true` — Use ECM algorithm to handle `NaN` values.

For more information on the ECM algorithm, see `ecmmle` and “Multivariate Normal Regression” on page 9-2.

Default: false

'GetAssetList'

If a `fints` object is passed into this method and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is currently `'Asset'`.

If the `GetAssetList` flag is `false`, no action occurs, which is the default behavior.

Acceptable values for `GetAssetList` are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

Default: false

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Mean and Covariance of Asset Returns from Data

To illustrate using the `estimateAssetMoments` method, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with the `portsim` function. The default behavior `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by the `portsim` function in the variable `X`, a price series is created in the variable `Y`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
X = portsim(m', C, 120);
Y = ret2tick(X);
```

Given asset returns and prices in the variables `X` and `Y` from above, the following examples demonstrate equivalent ways to estimate asset moments for the Portfolio object. A Portfolio object is created in `p` with the moments of asset returns set directly in the constructor and a second Portfolio object is created in `q` to obtain the mean and covariance of asset returns from asset return data in `X` using the `estimateAssetMoments` method.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
p = Portfolio('mean',m,'covar',C);
q = Portfolio;
q = estimateAssetMoments(q, X);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)
```

```
passetmean =
```

```
0.0042
0.0083
0.0100
0.0150
```

```
passetcovar =
```

```
0.0005  0.0003  0.0002  0
0.0003  0.0024  0.0017  0.0010
0.0002  0.0017  0.0048  0.0028
0  0.0010  0.0028  0.0102
```

```
qassetmean =
```

```
0.0042
0.0083
0.0100
0.0150
```

```
qassetcovar =
```

```
0.0005  0.0003  0.0002  0.0000
0.0003  0.0024  0.0017  0.0010
0.0002  0.0017  0.0048  0.0028
0.0000  0.0010  0.0028  0.0102
```

Notice how either approach yields the same moments. The default behavior of the `estimateAssetMoments` method is to work with asset returns. If, instead, you have asset prices, such as in the variable `Y`, the `estimateAssetMoments` method accepts a parameter name `'DataFormat'` with a corresponding value set to `'prices'` to indicate that the input to the method is in the form of asset prices and not returns (the default parameter value for `'DataFormat'` is `'returns'`). The following example compares direct assignment of moments in the Portfolio object `p` with estimated moments from asset price data in `Y` in the Portfolio object `q`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
Y = ret2tick(X);

p = Portfolio('mean',m,'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Y, 'dataformat', 'prices');

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =

    0.0005    0.0003    0.0002    0.0000
```

0.0003	0.0024	0.0017	0.0010
0.0002	0.0017	0.0048	0.0028
0.0000	0.0010	0.0028	0.0102

- “Asset Returns and Moments of Asset Returns” on page 4-41
- “Estimating Asset Moments from Time Series Data” on page 4-50

See Also

`estimateBounds` | `Portfolio` | `portsim`

More About

- “Portfolio Optimization Theory” on page 4-2

estimateBounds

Class: Portfolio

Estimate global lower and upper bounds for set of portfolios

Syntax

```
[glb,gub,isbounded] = estimateBounds(obj)
[glb,gub,isbounded] = estimateBounds(obj,
obtainExactBounds)
```

Description

[glb,gub,isbounded] = estimateBounds(obj) estimates the global lower and upper bounds for a given set of portfolios.

[glb,gub,isbounded] = estimateBounds(obj, obtainExactBounds) estimates the global lower and upper bounds for a given set of portfolios with an additional option specified for obtainExactBounds.

Tips

- You can also use dot notation to estimate the global lower and upper bounds for a given set of portfolios.

```
[glb, gub, isbounded] = obj.estimateBounds;
```
- Estimated bounds are accurate in most cases to within $1.0e-8$. If you intend to use these bounds directly in a Portfolio object, ensure that if you impose such bound constraints, a lower bound of 0 is probably preferable to a lower bound of, for example, $1.0e-10$ for portfolio weights.

Input Arguments

obj

Portfolio object [Portfolio].

obtainExactBounds

(Optional) Boolean flag to specify whether to solve for all bounds or to accept specified bounds whenever available [logical]. If bounds are known, set **obtainExactBounds** to **false** to accept known bounds.

Default: True

Output Arguments

glb

Global lower bounds for portfolio set [vector].

gub

Global upper bounds for portfolio set [vector].

isbounded

Indicates if set is empty ([]), bounded (**true**), or unbounded (**false**).

Note: By definition, any portfolio set must be nonempty and bounded:

- If the set is empty, **isbounded** = [].
 - If the set is nonempty and unbounded, **isbounded** = **false**.
 - If the set is nonempty and bounded, **isbounded** = **true**.
 - If the set is empty, **glb** and **gub** are set to NaN vectors.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Create an Unbounded Portfolio

Create an unbounded portfolio set.

```
p = Portfolio('AInequality', [1 -1; 1 1 ], 'bInequality', 0);  
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
-Inf  
-Inf
```

```
ub =
```

```
1.0e-08 *  
-0.3712  
    Inf
```

```
isbounded =
```

```
0
```

The `estimateBounds` method returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

- “Validating a Portfolio Set” on page 4-79

See Also

`checkFeasibility`

More About

- “Portfolio Optimization Theory” on page 4-2

estimateBounds

Class: PortfolioCVaR

Estimate global lower and upper bounds for portfolio set in PortfolioCVaR object

Syntax

```
[glb,gub,isbounded] = estimateBounds(obj)
[glb,gub,isbounded] = estimateBounds(obj,
obtainExactBounds)
```

Description

[glb,gub,isbounded] = estimateBounds(obj) estimates the global lower and upper bounds for a given set of portfolios.

[glb,gub,isbounded] = estimateBounds(obj, obtainExactBounds) estimates the global lower and upper bounds for a given set of portfolios with an additional option specified for obtainExactBounds.

Tips

- You can also use dot notation to estimate the global lower and upper bounds for a given set of CVaR portfolios.

```
[glb, gub, isbounded] = obj.estimateBounds;
```
- Estimated bounds are accurate in most cases to within $1.0e-8$. If you intend to use these bounds directly in a CVaR portfolio object, keep in mind that if you impose such bound constraints, a lower bound of 0 is probably preferable to a lower bound of, for example, $1.0e-10$ for portfolio weights.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

obtainExactBounds

(Optional) Boolean flag to specify whether to solve for all bounds or to accept specified bounds whenever available [logical]. If bounds are known, set **obtainExactBounds** to **false** to accept known bounds.

Default: true

Output Arguments

glb

Global lower bounds for portfolio set [vector].

gub

Global upper bounds for portfolio set [vector].

isbounded

Indicates if the portfolio set is empty ([]), bounded (**true**), or unbounded (**false**).

Note: By definition, any portfolio set must be nonempty and bounded:

- If the set is empty, **isbounded** = [].
 - If the set is nonempty and unbounded, **isbounded** = **false**.
 - If the set is nonempty and bounded, **isbounded** = **true**.
 - If the set is empty, **glb** and **gub** are set to NaN vectors.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Create an Unbounded Portfolio

Create an unbounded portfolio set.

```
p = PortfolioCVaR('AInequality', [1 -1; 1 1 ], 'bInequality', 0);  
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
-Inf  
-Inf
```

```
ub =
```

```
1.0e-08 *  
-0.3712  
    Inf
```

```
isbounded =
```

```
0
```

The `estimateBounds` method returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

- “Validating a CVaR Portfolio Set” on page 5-78

See Also

`checkFeasibility`

More About

- “Portfolio Optimization Theory” on page 5-2

estimateBounds

Class: PortfolioMAD

Estimate global lower and upper bounds for portfolio set in PortfolioMAD object

Syntax

```
[glb,gub,isbounded] = estimateBounds(obj)
[glb,gub,isbounded] = estimateBounds(obj,
obtainExactBounds)
```

Description

[glb,gub,isbounded] = estimateBounds(obj) estimates the global lower and upper bounds for a given set of portfolios.

[glb,gub,isbounded] = estimateBounds(obj, obtainExactBounds) estimates the global lower and upper bounds for a given set of portfolios with an additional option specified for obtainExactBounds.

Tips

- You can also use dot notation to estimate the global lower and upper bounds for a given set of MAD portfolios.

```
[glb, gub, isbounded] = obj.estimateBounds;
```

- Estimated bounds are accurate in most cases to within $1.0e-8$. If you intend to use these bounds directly in a MAD portfolio object, keep in mind that if you impose such bound constraints, a lower bound of 0 is probably preferable to a lower bound of, for example, $1.0e-10$ for portfolio weights. If bounds are known, set obtainExactBounds to false to accept known bounds.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

obtainExactBounds

(Optional) Boolean flag to specify whether to solve for all bounds or to accept specified bounds whenever available [logical]. If bounds are known, set **obtainExactBounds** to **false** to accept known bounds.

Default: true

Output Arguments

glb

Global lower bounds for portfolio set [vector].

gub

Global upper bounds for portfolio set [vector].

isbounded

Indicates if the portfolio set is empty ([]), bounded (**true**), or unbounded (**false**).

Note: By definition, any portfolio set must be nonempty and bounded:

- If the set is empty, **isbounded** = [].
 - If the set is empty, **glb** and **gub** are set to NaN vectors.
 - If the set is nonempty and unbounded, **isbounded** = **false**.
 - If the set is nonempty and bounded, **isbounded** = **true**.
-

Attributes

Access public

Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Create an Unbounded Portfolio

Create an unbounded portfolio set.

```
p = PortfolioMAD('AInequality', [1 -1; 1 1 ], 'bInequality', 0);  
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
-Inf  
-Inf
```

```
ub =
```

```
1.0e-08 *  
-0.3712  
    Inf
```

```
isbounded =
```

```
0
```

The `estimateBounds` method returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

- “Validating a MAD Portfolio Set” on page 6-78

See Also

checkFeasibility

More About

- “Portfolio Optimization Theory” on page 6-2

estimateFrontier

Class: Portfolio

Estimate specified number of optimal portfolios on the efficient frontier

Syntax

```
[pwgt,pbuy,psell] = estimateFrontier(obj)
[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)
```

Description

[pwgt,pbuy,psell] = estimateFrontier(obj) estimates a default number of optimal portfolios on the efficient frontier.

[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts) estimates the specified number of optimal portfolios on the efficient frontier with an additional option for NumPorts.

Tips

You can also use dot notation to estimate the specified number of optimal portfolios over the entire efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontier(NumPorts);
```

Input Arguments

obj

Portfolio object [Portfolio].

NumPorts

(Optional) Number of points to obtain on the efficient frontier [scalar integer].

Note: If no value is specified for `NumPorts`, the default value is obtained from the hidden property `defaultNumPorts` (default value is 10). If `NumPorts = 1`, this method returns the portfolio specified by the hidden property `defaultFrontierLimit` (current default value is 'min').

Default: 10

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified number of portfolios spaced equally from minimum to maximum portfolio return [`NumAssets-by-NumPorts` matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [`NumAssets-by-NumPorts` matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [`NumAssets-by-NumPorts` matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be 0 such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain the Default Number of Efficient Portfolios

Obtain the default number of efficient portfolios over the entire range of the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
disp(pwgt);
```

Columns 1 through 7

0.8891	0.7215	0.5540	0.3865	0.2190	0.0515	0
0.0369	0.1289	0.2209	0.3129	0.4049	0.4969	0.4049
0.0404	0.0567	0.0730	0.0893	0.1056	0.1219	0.1320
0.0336	0.0929	0.1521	0.2113	0.2705	0.3297	0.4630

Columns 8 through 10

0	0	0
0.2314	0.0579	0
0.1394	0.1468	0
0.6292	0.7953	1.0000

Obtain Purchases and Sales for Portfolios on the Efficient Frontier

Starting from the initial portfolio, the `estimateFrontier` method returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. Given an initial portfolio in `pwgt0`, you can obtain purchases and sales.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
```

```

0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
display(pbuy);
display(psell);

```

```
pwgt =
```

```
Columns 1 through 7
```

0.8891	0.7215	0.5540	0.3865	0.2190	0.0515	0
0.0369	0.1289	0.2209	0.3129	0.4049	0.4969	0.4049
0.0404	0.0567	0.0730	0.0893	0.1056	0.1219	0.1320
0.0336	0.0929	0.1521	0.2113	0.2705	0.3297	0.4630

```
Columns 8 through 10
```

0	0	0
0.2314	0.0579	0
0.1394	0.1468	0
0.6292	0.7953	1.0000

```
pbuy =
```

```
Columns 1 through 7
```

0.5891	0.4215	0.2540	0.0865	0	0	0
0	0	0	0.0129	0.1049	0.1969	0.1049
0	0	0	0	0	0	0
0	0	0.0521	0.1113	0.1705	0.2297	0.3630

```
Columns 8 through 10
```

0	0	0
0	0	0
0	0	0

0.5292 0.6953 0.9000

psell =

Columns 1 through 7

0	0	0	0	0.0810	0.2485	0.3000
0.2631	0.1711	0.0791	0	0	0	0
0.1596	0.1433	0.1270	0.1107	0.0944	0.0781	0.0680
0.0664	0.0071	0	0	0	0	0

Columns 8 through 10

0.3000	0.3000	0.3000
0.0686	0.2421	0.3000
0.0606	0.0532	0.2000
0	0	0

- “Estimate Efficient Frontiers” on page 4-96
- “Estimate Efficient Portfolios” on page 4-84

See Also

estimateFrontierByReturn | estimateFrontierByRisk |
estimateFrontierLimits

estimateFrontier

Class: PortfolioCVaR

Estimate specified number of optimal portfolios on efficient frontier for PortfolioCVaR object

Syntax

```
[pwgt,pbuy,psell] = estimateFrontier(obj)
[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)
```

Description

[pwgt,pbuy,psell] = estimateFrontier(obj) estimates a default number of optimal portfolios on the efficient frontier.

[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts) estimates the specified number of optimal portfolios on the efficient frontier with an additional option for NumPorts.

Tips

You can also use dot notation to estimate the specified number of optimal portfolios over the entire efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontier(NumPorts);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

NumPorts

(Optional) Number of points to obtain on the efficient frontier [scalar integer].

Note: If no value is specified for `NumPorts`, the default value is obtained from the hidden property `defaultNumPorts` (default value is 10). If `NumPorts = 1`, this method returns the portfolio specified by the hidden property `defaultFrontierLimit` (current default value is 'min').

Default: 10

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with a specified number of portfolios spaced equally from minimum to maximum portfolio return [`NumAssets-by-NumPorts` matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [`NumAssets-by-NumPorts` matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [`NumAssets-by-NumPorts` matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be 0 such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain the Default Number of Efficient Portfolios

Obtain the default number of efficient portfolios over the entire range of the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);

disp(pwgt);

Columns 1 through 7

    0.8454    0.6847    0.5151    0.3541    0.1902    0.0314    0.0000
    0.0599    0.1427    0.2302    0.3165    0.3980    0.4733    0.3513
    0.0462    0.0639    0.0945    0.1079    0.1345    0.1583    0.1756
    0.0485    0.1087    0.1602    0.2215    0.2773    0.3371    0.4731

Columns 8 through 10

         0         0    0.0000
    0.1806         0    0.0000
    0.1916    0.2212    0.0000
```

```
0.6278    0.7788    1.0000
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Purchases and Sales for Portfolios on the Efficient Frontier

Starting from the initial portfolio, the `estimateFrontier` method returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. Given an initial portfolio in `pwgt0`, you can obtain purchases and sales.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);
p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =
```

```
Columns 1 through 7
```

```
0.8454    0.6847    0.5151    0.3541    0.1902    0.0314    0.0000
0.0599    0.1427    0.2302    0.3165    0.3980    0.4733    0.3513
0.0462    0.0639    0.0945    0.1079    0.1345    0.1583    0.1756
0.0485    0.1087    0.1602    0.2215    0.2773    0.3371    0.4731
```


Columns 8 through 10

0	0	0.0000
0.1806	0	0.0000
0.1916	0.2212	0.0000
0.6278	0.7788	1.0000

pbuy =

Columns 1 through 7

0.5454	0.3847	0.2151	0.0541	0	0	0
0	0	0	0.0165	0.0980	0.1733	0.0513
0	0	0	0	0	0	0
0	0.0087	0.0602	0.1215	0.1773	0.2371	0.3731

Columns 8 through 10

0	0	0
0	0	0
0	0.0212	0
0.5278	0.6788	0.9000

psell =

Columns 1 through 7

0	0	0	0	0.1098	0.2686	0.3000
0.2401	0.1573	0.0698	0	0	0	0
0.1538	0.1361	0.1055	0.0921	0.0655	0.0417	0.0244
0.0515	0	0	0	0	0	0

Columns 8 through 10

0.3000	0.3000	0.3000
0.1194	0.3000	0.3000
0.0084	0	0.2000
0	0	0

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 5-83

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` |
`estimateFrontierLimits` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

estimateFrontier

Class: PortfolioMAD

Estimate specified number of optimal portfolios on efficient frontier for PortfolioMAD object

Syntax

```
[pwgt,pbuy,psell] = estimateFrontier(obj)
[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)
```

Description

[pwgt,pbuy,psell] = estimateFrontier(obj) estimates a default number of optimal portfolios on the efficient frontier.

[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts) estimates the specified number of optimal portfolios on the efficient frontier with an additional option for NumPorts.

Tips

You can also use dot notation to estimate the specified number of optimal portfolios over the entire efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontier(NumPorts);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

NumPorts

(Optional) Number of points to obtain on the efficient frontier [scalar integer].

Note: If no value is specified for `NumPorts`, the default value is obtained from the hidden property `defaultNumPorts` (default value is 10). If `NumPorts = 1`, this method returns the portfolio specified by the hidden property `defaultFrontierLimit` (current default value is 'min').

Default: 10

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with a specified number of portfolios spaced equally from minimum to maximum portfolio return [`NumAssets-by-NumPorts` matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [`NumAssets-by-NumPorts` matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [`NumAssets-by-NumPorts` matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be 0 such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain the Default Number of Efficient Portfolios

Obtain the default number of efficient portfolios over the entire range of the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p);

disp(pwgt);

Columns 1 through 7

    0.8815    0.7151    0.5487    0.3817    0.2170    0.0499    0.0000
    0.0431    0.1282    0.2128    0.2981    0.3825    0.4662    0.3609
    0.0389    0.0605    0.0826    0.1053    0.1241    0.1492    0.1786
    0.0365    0.0963    0.1559    0.2149    0.2764    0.3348    0.4605

Columns 8 through 10

    0.0000         0    0.0000
    0.1755    0.0000    0.0000
    0.2095    0.2266    0.0000
    0.6150    0.7734    1.0000
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Purchases and Sales for Portfolios on the Efficient Frontier

Starting from the initial portfolio, the `estimateFrontier` method returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. Given an initial portfolio in `pwgt0`, you can obtain purchases and sales.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);
p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =
```

```
Columns 1 through 7
```

0.8815	0.7151	0.5487	0.3817	0.2170	0.0499	0.0000
0.0431	0.1282	0.2128	0.2981	0.3825	0.4662	0.3609
0.0389	0.0605	0.0826	0.1053	0.1241	0.1492	0.1786
0.0365	0.0963	0.1559	0.2149	0.2764	0.3348	0.4605

```
Columns 8 through 10
```

0.0000	0	0.0000
0.1755	0.0000	0.0000
0.2095	0.2266	0.0000
0.6150	0.7734	1.0000

pbuy =

Columns 1 through 7

0.5815	0.4151	0.2487	0.0817	0	0	0
0	0	0	0	0.0825	0.1662	0.0609
0	0	0	0	0	0	0
0	0	0.0559	0.1149	0.1764	0.2348	0.3605

Columns 8 through 10

0	0	0
0	0	0
0.0095	0.0266	0
0.5150	0.6734	0.9000

psell =

Columns 1 through 7

0	0	0	0	0.0830	0.2501	0.3000
0.2569	0.1718	0.0872	0.0019	0	0	0
0.1611	0.1395	0.1174	0.0947	0.0759	0.0508	0.0214
0.0635	0.0037	0	0	0	0	0

Columns 8 through 10

0.3000	0.3000	0.3000
0.1245	0.3000	0.3000
0	0	0.2000
0	0	0

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 6-83

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` |
`estimateFrontierLimits` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

estimateFrontierByReturn

Class: Portfolio

Estimate optimal portfolios with targeted portfolio returns

Syntax

```
[pwgt, pbuy, psell] = estimateFrontierByReturn(obj,  
TargetReturn)
```

Description

[pwgt, pbuy, psell] = estimateFrontierByReturn(obj, TargetReturn) estimates optimal portfolios with targeted portfolio returns.

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio returns.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByReturn(TargetReturn);
```

Input Arguments

obj

Portfolio object [Portfolio].

TargetReturn

Target values for portfolio return [NumPorts vector].

Note: TargetReturn specifies target returns for portfolios on the efficient frontier. If any TargetReturn values are outside the range of returns for efficient portfolios, the TargetReturn is replaced with the minimum or maximum efficient portfolio return,

depending upon whether the target return is below or above the range of efficient portfolio returns.

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified target returns from TargetReturn that are [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier that are [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier that are [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0, such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain the Portfolio for Targeted Portfolio Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` method accepts one or more target portfolio returns and

obtains efficient portfolios with the specified returns. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 6%, 9%, and 12%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);

display(pwgt);
```

```
pwgt =

    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022
```

- “Estimate Efficient Frontiers” on page 4-96
- “Estimate Efficient Portfolios” on page 4-84

See Also

[estimateFrontier](#) | [estimateFrontierByRisk](#) | [estimateFrontierLimits](#)

estimateFrontierByReturn

Class: PortfolioCVaR

Estimate optimal portfolios with targeted portfolio returns for PortfolioCVaR object

Syntax

```
[pwgt, pbuy, psell] = estimateFrontierByReturn(obj,  
TargetReturn)
```

Description

[pwgt, pbuy, psell] = estimateFrontierByReturn(obj, TargetReturn) estimates optimal portfolios with targeted portfolio returns.

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio returns.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByReturn(TargetReturn);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

TargetReturn

Target values for CVaR portfolio object returns [NumPorts vector].

Note: TargetReturn specifies target returns for portfolios on the efficient frontier. If any TargetReturn values are outside the range of returns for efficient portfolios, TargetReturn is replaced with the minimum or maximum efficient portfolio return,

depending whether the target return is below or above the range of efficient portfolio returns.

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified target returns from TargetReturn [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0, such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain the Portfolio for Targeted Portfolio Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` method accepts one or more target portfolio returns and

obtains efficient portfolios with the specified returns. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 13%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioCVaR;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierByReturn(p, [0.07 0.10, 0.13]);

display(pwgt);

pwgt =

    0.7371    0.3071         0
    0.1504    0.3919    0.4396
    0.0286    0.1011    0.1360
    0.0839    0.1999    0.4244
```

The function `rng(seed)` is used to reset the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 5-83

See Also

`estimateFrontier` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

estimateFrontierByReturn

Class: PortfolioMAD

Estimate optimal portfolios with targeted portfolio returns for PortfolioMAD object

Syntax

```
[pwgt, pbuy, psell] = estimateFrontierByReturn(obj,  
TargetReturn)
```

Description

[pwgt, pbuy, psell] = estimateFrontierByReturn(obj,
TargetReturn) estimates optimal portfolios with targeted portfolio returns.

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio returns.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByReturn(TargetReturn);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

TargetReturn

Target values for MAD portfolio object returns [NumPorts vector].

Note: TargetReturn specifies target returns for portfolios on the efficient frontier. If any TargetReturn values are outside the range of returns for efficient portfolios, TargetReturn is replaced with the minimum or maximum efficient portfolio return,

depending whether the target return is below or above the range of efficient portfolio returns.

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified target returns from TargetReturn [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0, such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain the Portfolio for Targeted Portfolio Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` method accepts one or more target portfolio returns and

obtains efficient portfolios with the specified returns. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 13%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioMAD;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByReturn(p, [0.07 0.10, 0.13]);

display(pwgt);

pwgt =

    0.7436    0.3147    0.0000
    0.1357    0.3835    0.4422
    0.0328    0.0939    0.1324
    0.0879    0.2079    0.4254
```

The function `rng(seed)` is used to reset the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 6-83

See Also

`estimateFrontier` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

estimateFrontierByRisk

Class: Portfolio

Estimate optimal portfolios with targeted portfolio risks

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)
```

Description

[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk) estimates optimal portfolios with targeted portfolio risks.

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio risks.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByRisk(TargetRisk);
```

Input Arguments

obj

Portfolio object [Portfolio].

TargetRisk

Target values for portfolio risk [NumPorts vector].

Note: If any **TargetRisk** values are outside the range of risks for efficient portfolios, the target risk is replaced with the minimum or maximum efficient portfolio risk, depending on whether the target risk is below or above the range of efficient portfolio risks.

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified target risks from `TargetRisk` that are [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier that are [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier that are [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0 such that so `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Portfolios with Targeted Portfolio Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` method accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Assume you have a universe of four

assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);
```

```
pwgt =
```

```
    0.3984    0.2659    0.1416
    0.3064    0.3791    0.4474
    0.0882    0.1010    0.1131
    0.2071    0.2540    0.2979
```

- “Estimate Efficient Frontiers” on page 4-96
- “Estimate Efficient Portfolios” on page 4-84

See Also

[estimateFrontier](#) | [estimateFrontierByReturn](#) | [estimateFrontierLimits](#)

estimateFrontierByRisk

Class: PortfolioCVaR

Estimate optimal portfolios with targeted portfolio risks for PortfolioCVaR object

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)
```

Description

[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk) estimates optimal portfolios with targeted portfolio risks.

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio risks.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByRisk(TargetRisk);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

TargetRisk

Target values for CVaR portfolio object risk [NumPorts vector].

Note: If any **TargetRisk** values are outside the range of risks for efficient portfolios, the target risk is replaced with the minimum or maximum efficient portfolio risk, depending on whether the target risk is below or above the range of efficient portfolio risks.

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified target risks from `TargetRisk` [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0 such that so `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Portfolios with Targeted Portfolio Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` method accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Assume you have a universe of four

assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 20%, and 30%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioCVaR;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierByRisk(p, [0.12, 0.20, 0.30]);

display(pwgt);

pwgt =

    0.5363    0.1388    0.0000
    0.2655    0.4990    0.3830
    0.0568    0.1239    0.1461
    0.1413    0.2383    0.4709
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 5-83

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierLimits` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

estimateFrontierByRisk

Class: PortfolioMAD

Estimate optimal portfolios with targeted portfolio risks for PortfolioMAD object

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)
```

Description

[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk) estimates optimal portfolios with targeted portfolio risks.

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio risks.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByRisk(TargetRisk);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

TargetRisk

Target values for MAD portfolio object risk [NumPorts vector].

Note: If any **TargetRisk** values are outside the range of risks for efficient portfolios, the target risk is replaced with the minimum or maximum efficient portfolio risk, depending on whether the target risk is below or above the range of efficient portfolio risks.

Output Arguments

pwgt

Optimal portfolios on the efficient frontier with specified target risks from `TargetRisk` [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0 such that so `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Portfolios with Targeted Portfolio Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` method accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Assume you have a universe of four

assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 20%, and 25%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioMAD;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByRisk(p, [0.12, 0.20, 0.25]);

display(pwgt);

pwgt =

    0.1613    0.0000         0
    0.4777    0.2139    0.0037
    0.1118    0.1381    0.1214
    0.2492    0.6480    0.8749
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 6-83

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierLimits` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

estimateFrontierLimits

Class: Portfolio

Estimate optimal portfolios at endpoints of efficient frontier

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierLimits(obj)  
[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice)
```

Description

[pwgt,pbuy,psell] = estimateFrontierLimits(obj) estimates the optimal portfolios at the endpoints of the efficient frontier.

[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice) estimates the optimal portfolios at the endpoints of the efficient frontier with an additional option specified for the Choice argument.

Tips

You can also use dot notation to estimate the optimal portfolios at the endpoints of the efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontierLimits(Choice);
```

Input Arguments

obj

Portfolio object [Portfolio].

Choice

Indicates which portfolios to obtain at the extreme ends of the efficient frontier [string].

Choice specifies various actions with default value `[]`. The options for `Choice` action are:

- `[]` — Compute both minimum-risk and maximum-return portfolios.
- `'Both'` — Compute both minimum-risk and maximum-return portfolios.
- `'Min'` — Compute minimum-risk portfolio only.
- `'Max'` — Compute maximum-return portfolio only.

Default: `[]`

Output Arguments

pwgt

Optimal portfolios at the endpoints of the efficient frontier `TargetReturn` that are `[NumAssets-by-NumPorts matrix]`.

pbuy

Purchases relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier that are `[NumAssets-by-NumPorts matrix]`.

psell

Sales relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier that are `[NumAssets-by-NumPorts matrix]`.

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0 such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Endpoint Portfolios

Given portfolio `p`, the `estimateFrontierLimits` method obtains the endpoint portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
```

```
disp(pwgt);
```

```
0.8891      0
0.0369      0
0.0404      0
0.0336      1.0000
```

- “Estimate Efficient Frontiers” on page 4-96
- “Estimate Efficient Portfolios” on page 4-84

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk`

estimateFrontierLimits

Class: PortfolioCVaR

Estimate optimal portfolios at endpoints of efficient frontier for PortfolioCVaR object

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierLimits(obj)
[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice)
```

Description

[pwgt,pbuy,psell] = estimateFrontierLimits(obj) estimates the optimal portfolios at the endpoints of the efficient frontier.

[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice) estimates the optimal portfolios at the endpoints of the efficient frontier with an additional option specified for the Choice argument.

Tips

You can also use dot notation to estimate the optimal portfolios at the endpoints of the efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontierLimits(Choice);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Choice

(Optional) Indicates which portfolios to obtain at the extreme ends of the efficient frontier [string].

Choice specifies various actions with default value []. The options for Choice action are:

- [] — Compute both minimum-risk and maximum-return portfolios.
- 'Both' — Compute both minimum-risk and maximum-return portfolios.
- 'Min' — Compute minimum-risk portfolio only.
- 'Max' — Compute maximum-return portfolio only.

The number of portfolios returned by this method is either 1 or 2.

Default: []

Output Arguments

pwgt

Optimal portfolios at the endpoints of the efficient frontier TargetReturn [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in `obj.InitPort`, it is assumed to be 0 such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Endpoint Portfolios

Given portfolio `p`, the `estimateFrontierLimits` method obtains the endpoint portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVar;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierLimits(p);

disp(pwgt);

    0.8454    0.0000
    0.0599    0.0000
    0.0462    0.0000
    0.0485    1.0000
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 5-83

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

estimateFrontierLimits

Class: PortfolioMAD

Estimate optimal portfolios at endpoints of efficient frontier for PortfolioMAD object

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierLimits(obj)
[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice)
```

Description

[pwgt,pbuy,psell] = estimateFrontierLimits(obj) estimates the optimal portfolios at the endpoints of the efficient frontier.

[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice) estimates the optimal portfolios at the endpoints of the efficient frontier with an additional option specified for the Choice argument.

Tips

You can also use dot notation to estimate the optimal portfolios at the endpoints of the efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontierLimits(Choice);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Choice

(Optional) Indicates which portfolios to obtain at the extreme ends of the efficient frontier [string].

Choice specifies various actions with default value []. The options for Choice action are:

- [] — Compute both minimum-risk and maximum-return portfolios.
- 'Both' — Compute both minimum-risk and maximum-return portfolios.
- 'Min' — Compute minimum-risk portfolio only.
- 'Max' — Compute maximum-return portfolio only.

The number of portfolios returned by this method is either 1 or 2.

Default: []

Output Arguments

pwgt

Optimal portfolios at the endpoints of the efficient frontier TargetReturn [NumAssets-by-NumPorts matrix].

pbuy

Purchases relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier [NumAssets-by-NumPorts matrix].

psell

Sales relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier [NumAssets-by-NumPorts matrix].

Note: If no initial portfolio is specified in obj.InitPort, it is assumed to be 0 such that pbuy = max(0, pwgt) and psell = max(0, -pwgt).

Attributes

Access	public
Static	false

Hidden false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Endpoint Portfolios

Given portfolio `p`, the `estimateFrontierLimits` method obtains the endpoint portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierLimits(p);

disp(pwgt);

    0.8815    0.0000
    0.0431    0.0000
    0.0389    0.0000
    0.0365    1.0000
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios” on page 6-83

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

estimateMaxSharpeRatio

Class: Portfolio

Estimate efficient portfolio to maximize Sharpe ratio

Syntax

```
[pwgt,pbuy,pse11] = estimateMaxSharpeRatio(obj)
```

Description

[pwgt,pbuy,pse11] = estimateMaxSharpeRatio(obj) estimates an efficient portfolio that maximizes the Sharpe ratio.

Tips

You can also use dot notation to estimate an efficient portfolio that maximizes the Sharpe ratio.

```
[pwgt,pbuy,pse11] = obj.estimateMaxSharpeRatio;
```

Input Arguments

obj

Portfolio object [Portfolio].

Note: The risk-free rate is obtained from the property `RiskFreeRate` in the Portfolio object. If you leave the `RiskFreeRate` unset, it is assumed to be 0.

Output Arguments

pwgt

A portfolio on the efficient frontier with a maximum Sharpe ratio [NumAssets vector].

pbuy

Purchases relative to an initial portfolio for a portfolio on the efficient frontier with a maximum Sharpe ratio [NumAssets vector].

psell

Sales relative to an initial portfolio for a portfolio on the efficient frontier with maximum Sharpe ratio [NumAssets vector].

Definitions

Sharpe Ratio

The Sharpe ratio is the ratio of the difference between the mean of portfolio returns and the risk-free rate divided by the standard deviation of portfolio returns. This method maximizes the Sharpe ratio among portfolios on the efficient frontier.

Attributes

Access	public
Static	false
Hidden	false

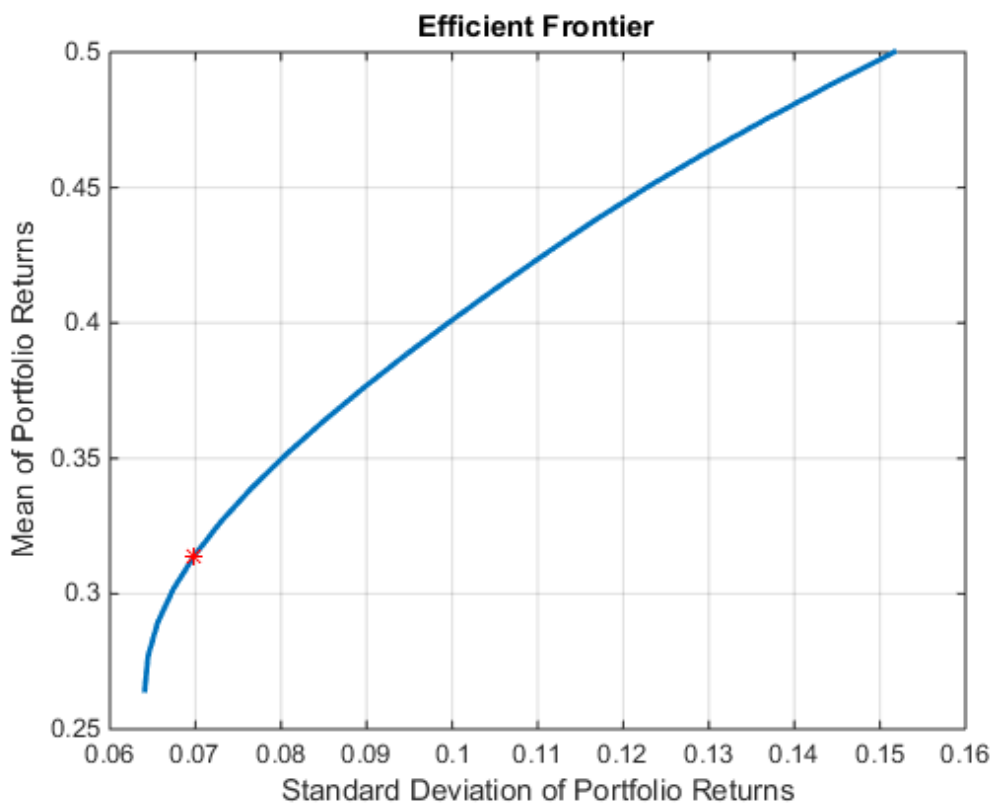
To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Efficient Portfolio that Maximizes the Sharpe Ratio

Estimate the efficient portfolio that maximizes the Sharpe ratio.

```
p = Portfolio('AssetMean',[0.3, 0.1, 0.5], 'AssetCovar',...  
[0.01, -0.010, 0.004; -0.010, 0.040, -0.002; 0.004, -0.002, 0.023]);  
p = setDefaultConstraints(p);  
plotFrontier(p, 20);  
weights = estimateMaxSharpeRatio(p);  
[risk, ret] = estimatePortMoments(p, weights);  
hold on  
plot(risk,ret, '*r');
```



- “Efficient Portfolio that Maximizes Sharpe Ratio” on page 4-92

Algorithms

The maximization of the Sharpe ratio is accomplished by a one-dimensional optimization using `fminbnd` to find the portfolio that minimizes the negative of the Sharpe ratio. The method takes only a fully qualified Portfolio object as its input and uses all information in the object to solve the problem.

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `Portfolio`

More About

- “Portfolio Optimization Theory” on page 4-2

estimatePortMoments

Class: Portfolio

Estimate moments of portfolio returns

Syntax

```
[prsk, pret] = estimatePortMoments(obj,pwgt)
```

Description

`[prsk, pret] = estimatePortMoments(obj,pwgt)` estimates the moments of portfolio returns.

The estimate of port moments is specific to mean-variance portfolio optimization and computes the mean and standard deviation (which is the square-root of variance) of portfolio returns.

Tips

You can also use dot notation to estimate the moments of portfolio returns.

```
[prsk, pret] = obj.estimatePortMoments(pwgt);
```

Input Arguments

obj

Portfolio object [Portfolio].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where NumAssets is the number of asset in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

prsk

Estimates for standard deviations of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

pret

Estimates for means of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Identify the Range of Risks and Returns for Efficient Portfolios

Given portfolio `p`, use the `estimatePortMoments` method to show the range of risks and returns for efficient portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
```

```
[prsk, pret] = estimatePortMoments(p, pwgt);  
disp([prsk, pret]);
```

```
    0.0769    0.0590  
    0.3500    0.1800
```

- “Estimate Efficient Portfolios” on page 4-84

See Also

`estimatePortReturn` | `estimatePortRisk`

More About

- “Portfolio Optimization Theory” on page 4-2

estimatePortReturn

Class: Portfolio

Estimate mean of portfolio returns

Syntax

```
pret = estimatePortReturn(obj ,pwgt)
```

Description

`pret = estimatePortReturn(obj ,pwgt)` estimates the mean of portfolio returns (as the proxy for portfolio return).

Note: Depending on whether costs have been set, the portfolio return is either gross or net portfolio returns.

Tips

You can also use dot notation to estimate the mean of portfolio returns (as the proxy for portfolio return).

```
pret = obj.estimatePortReturn(pwgt);
```

Input Arguments

obj

Portfolio object [Portfolio].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where NumAssets is the number of assets in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

pret

Estimates for means of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate the Mean of Portfolio Returns

Given portfolio `p`, use the `estimatePortReturn` method to estimate the mean of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
pret = estimatePortReturn(p, pwgt);
disp(pret)
```

```
0.0590
0.1800
```

- “Obtaining Efficient Portfolios for Target Returns” on page 4-87

See Also

[estimateFrontierByReturn](#) | [estimateFrontierByRisk](#) | [estimatePortRisk](#)

More About

- “Portfolio Optimization Theory” on page 4-2

estimatePortReturn

Class: PortfolioCVaR

Estimate mean of portfolio returns for PortfolioCVaR object

Syntax

```
pret = estimatePortReturn(obj ,pwgt)
```

Description

`pret = estimatePortReturn(obj ,pwgt)` estimates the mean of portfolio returns (as the proxy for portfolio returns).

Note: Depending on whether costs have been set, the portfolio return is either gross or net portfolio returns.

Tips

You can also use dot notation to estimate the mean of portfolio returns (as the proxy for portfolio return).

```
pret = obj.estimatePortReturn(pwgt);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where NumAssets is the number of assets in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

pret

Estimates for means of portfolio returns for each portfolio in `pwgt` [NumPorts vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate the Mean of Portfolio Returns

Given portfolio `p`, use the `estimatePortReturn` method to estimate the mean of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);
```

```
pwgt = estimateFrontierLimits(p);  
pret = estimatePortReturn(p, pwgt);  
disp(pret)  
  
    0.0050  
    0.0154
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Efficient Portfolios for Target Returns” on page 5-87

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

estimatePortReturn

Class: PortfolioMAD

Estimate mean of portfolio returns for PortfolioMAD object

Syntax

```
pret = estimatePortReturn(obj ,pwgt)
```

Description

`pret = estimatePortReturn(obj ,pwgt)` estimates the mean of portfolio returns (as the proxy for portfolio returns).

Note: Depending on whether costs have been set, the portfolio return is either gross or net portfolio returns.

Tips

You can also use dot notation to estimate the mean of portfolio returns (as the proxy for portfolio return).

```
pret = obj.estimatePortReturn(pwgt);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Output Arguments

pret

Estimates for means of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Portfolio weights are a [`NumAssets`-by-`NumPorts` matrix] and `pret` is a `NumPorts`-by-1 vector, where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate the Mean of Portfolio Returns

Given portfolio `p`, use the `estimatePortReturn` method to estimate the mean of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);
```

```
p = PortfolioMAD;  
p = setScenarios(p, AssetScenarios);  
p = setDefaultConstraints(p);  
  
pwgt = estimateFrontierLimits(p);  
pret = estimatePortReturn(p, pwgt);  
disp(pret)  
  
    0.0048  
    0.0154
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Efficient Portfolios for Target Returns” on page 6-87

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

estimatePortRisk

Class: Portfolio

Estimate standard deviation of portfolio returns (portfolio risk)

Syntax

```
prsk = estimatePortRisk(obj, pwgt)
```

Description

`prsk = estimatePortRisk(obj, pwgt)` estimates standard deviation of portfolio returns (portfolio risk).

Tips

You can also use dot notation to estimate standard deviation of portfolio returns (portfolio risk).

```
prsk = obj.estimatePortRisk(pwgt);
```

Input Arguments

obj

Portfolio object [Portfolio].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where NumAssets is the number of assets in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

prsk

Estimates for standard deviations of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Conditional Value-at-Risk of Portfolio Returns as the Proxy for Portfolio Risk

Given portfolio `p`, use the `estimatePortRisk` method to show the standard deviation of portfolio returns for each portfolio in `pwgt`.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
prsk = estimatePortRisk(p, pwgt);
disp(prsk)
```

```
0.0769
0.3500
```

- “Obtaining Efficient Portfolios for Target Risks” on page 4-90

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortRisk`

More About

- “Portfolio Optimization Theory” on page 4-2

estimatePortRisk

Class: PortfolioCVaR

Estimate conditional value-at-risk (CVaR) portfolio risk proxy

Syntax

```
prsk = estimatePortRisk(obj,pwgt)
```

Description

`prsk = estimatePortRisk(obj,pwgt)` estimates conditional value-at-risk of portfolio returns (portfolio risk proxy), where the probability level is from the PortfolioCVaR property `ProbabilityLevel`.

Tips

You can also use dot notation to estimate conditional value-at-risk of portfolio returns (portfolio risk proxy).

```
prsk = obj.estimatePortRisk(pwgt);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Output Arguments

prsk

Estimate for conditional value-at-risk (CVaR) of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Conditional Value-at-Risk of Portfolio Returns as the Proxy for Portfolio Risk

Given a portfolio `pwgt`, use the `estimatePortRisk` method to show the conditional value-at-risk (CVaR) of portfolio returns for each portfolio.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
p = setProbabilityLevel(p, 0.95);  
  
pwgt = estimateFrontierLimits(p);  
prsk = estimatePortRisk(p, pwgt);  
disp(prsk)  
  
    0.0407  
    0.1911
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Efficient Portfolios for Target Risks” on page 5-89

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortReturn`
| `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

estimatePortRisk

Class: PortfolioMAD

Estimate mean-absolute deviation (MAD) for portfolio risk proxy

Syntax

```
prsk = estimatePortRisk(obj, pwgt)
```

Description

`prsk = estimatePortRisk(obj, pwgt)` estimates the mean-absolute deviation of portfolio returns (portfolio risk proxy).

Tips

You can also use dot notation to estimate mean-absolute deviation of portfolio returns (portfolio risk proxy).

```
prsk = obj.estimatePortRisk(pwgt);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix] where NumAssets is the number of assets in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

prsk

Estimate for mean-absolute deviation of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Mean-Absolute Deviation Returns as the Proxy for Portfolio Risk

Given a portfolio `pwgt`, use the `estimatePortRisk` method to show the mean-absolute deviation of portfolio returns for each portfolio.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
pwgt = estimateFrontierLimits(p);  
prsk = estimatePortRisk(p, pwgt);  
disp(prsk)
```

```
0.0177  
0.0809
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Efficient Portfolios for Target Risks” on page 6-89

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortReturn`
| `rng`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

estimatePortStd

Class: PortfolioCVaR

Estimate standard deviation of portfolio returns for PortfolioCVaR object

Syntax

```
pstd = estimatePortStd(obj,pwgt)
```

Description

`pstd = estimatePortStd(obj,pwgt)` estimates the standard deviation of portfolio returns.

Tips

You can also use dot notation to estimate the standard deviation of portfolio returns.

```
pstd = obj.estimatePortStd(pwgt);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix], where NumAssets is the number of assets in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

pstd

Estimates for standard deviations of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Standard Deviations for Portfolio Returns

Given a portfolio `pwgt`, use the `estimatePortStd` method to show the standard deviation of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVar;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```



```
p = setProbabilityLevel(p, 0.95);  
pwgt = estimateFrontierLimits(p);  
pstd = estimatePortStd(p, pwgt);  
disp(pstd)  
  
    0.0223  
    0.1010
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Portfolio Standard Deviation and VaR” on page 5-97

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortReturn`
| `estimatePortVaR` | `rng`

estimatePortStd

Class: PortfolioMAD

Estimate standard deviation of portfolio returns for PortfolioMAD object

Syntax

```
pstd = estimatePortStd(obj,pwgt)
```

Description

`pstd = estimatePortStd(obj,pwgt)` estimates the standard deviation of portfolio returns. This method uses the asset return scenarios to compute the standard deviation of portfolio returns.

Tips

You can also use dot notation to estimate the standard deviation of portfolio returns.

```
pstd = obj.estimatePortStd(pwgt);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

pwgt

Collection of portfolios [NumAssets-by-NumPorts matrix], where NumAssets is the number of assets in the universe and NumPorts is the number of portfolios in the collection of portfolios.

Output Arguments

pstd

Estimates for standard deviations of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Standard Deviations for Portfolio Returns

Given a portfolio `pwgt`, use the `estimatePortStd` method to show the standard deviation of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
pwgt = estimateFrontierLimits(p);  
pstd = estimatePortStd(p, pwgt);  
disp(pstd)  
    0.0222  
    0.1010
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Portfolio Standard Deviation” on page 6-96

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortReturn`
| `rng`

estimatePortVaR

Class: PortfolioCVaR

Estimate value-at-risk for PortfolioCVaR object

Syntax

```
pvar = estimatePortVaR(obj,pwgt)
```

Description

`pvar = estimatePortVaR(obj,pwgt)` estimates the value-at-risk for PortfolioCVaR object, where the probability level used is from the `PortfolioCVaR` property `ProbabilityLevel`.

Tips

You can also use dot notation to estimate the value-at-risk of PortfolioCVaR object.

```
pvar = obj.estimatePortVaR(pwgt);
```

Input Arguments

obj

CVaR portfolio object [`PortfolioCVaR`].

pwgt

Collection of portfolios [`NumAssets`-by-`NumPorts` matrix] where `NumAssets` is the number of asset in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Output Arguments

pvar

Estimates for value-at-risk of portfolio returns for each portfolio in `pwgt` [`NumPorts` vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Value-at-Risk for Portfolio

Given a portfolio `pwgt`, use the `estimatePortVaR` method to estimate the value-at-risk of portfolio.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVar;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
p = setProbabilityLevel(p, 0.95);  
pwgt = estimateFrontierLimits(p);  
pvar = estimatePortVaR(p, pwgt);  
disp(pvar)  
  
    0.0314  
    0.1483
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Obtaining Portfolio Standard Deviation and VaR” on page 5-97

See Also

`estimatePortStd` | `PortfolioVCaR` | `rng`

More About

- “Conditional Value-at-Risk” on page 5-5

estimateScenarioMoments

Class: PortfolioCVaR

Estimate mean and covariance of asset return scenarios in PortfolioCVaR object

Syntax

```
[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(obj)
```

Description

`[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(obj)` estimates the mean and covariance of asset return scenarios for a PortfolioCVaR object.

Tips

You can also use dot notation to estimate the mean and covariance of asset return scenarios for a portfolio.

```
[ScenarioMean, ScenarioCovar] = obj.estimateScenarioMoments
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

ScenarioMean

Estimate for mean of scenarios [NumAssets vector] or [].

ScenarioCovar

Estimate for covariance of scenarios [NumAssets-by-NumAssets] matrix or [].

Note: If no scenarios are associated with the specified object, both ScenarioMean and ScenarioCovar are set to empty [].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Mean and Covariance of Asset Return Scenarios

Given PortfolioCVaR object `p`, use the `estimatePortRisk` method to estimate mean and covariance of asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
p = setProbabilityLevel(p, 0.95);  
[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(p)  
  
ScenarioMean =  
  
    0.0039  
    0.0082  
    0.0102  
    0.0154  
  
ScenarioCovar =  
  
    0.0005    0.0003    0.0001   -0.0001  
    0.0003    0.0024    0.0017    0.0010  
    0.0001    0.0017    0.0048    0.0028  
   -0.0001    0.0010    0.0028    0.0102
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Asset Returns and Scenarios” on page 5-41

See Also

PortfolioVCaR | rng | setScenarios

More About

- “How Stochastic Optimization Works” on page 5-41

estimateScenarioMoments

Class: PortfolioMAD

Estimate mean and covariance of asset return scenarios in PortfolioMAD object

Syntax

```
[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(obj)
```

Description

[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(obj) estimates the mean and covariance of asset return scenarios for a PortfolioMAD object.

Tips

You can also use dot notation to estimate the mean and covariance of asset return scenarios for a portfolio.

```
[ScenarioMean, ScenarioCovar] = obj.estimateScenarioMoments
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

ScenarioMean

Estimate for mean of scenarios [NumAssets vector] or [].

ScenarioCovar

Estimate for covariance of scenarios [NumAssets-by-NumAssets] matrix or [].

Note: If no scenarios are associated with the specified object, both `ScenarioMean` and `ScenarioCovar` are set to empty [].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Mean and Covariance of Asset Return Scenarios

Given `PortfolioMAD` object `p`, use the `estimatePortRisk` method to estimate mean and covariance of asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(p)
```

```
ScenarioMean =
```

```
0.0039  
0.0082  
0.0102  
0.0154
```

```
ScenarioCovar =
```

```
0.0005    0.0003    0.0001   -0.0001  
0.0003    0.0024    0.0017    0.0010  
0.0001    0.0017    0.0048    0.0028  
-0.0001   0.0010    0.0028    0.0102
```

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Asset Returns and Scenarios” on page 6-41

See Also

`estimatePortRisk` | `PortfolioVCaR` | `rng` | `setScenarios` |
`simulateNormalScenariosByMoments`

More About

- “How Stochastic Optimization Works” on page 6-41

ewstats

Expected return and covariance from return time series

Syntax

```
[ExpReturn, ExpCovariance, NumEffObs] = ewstats(RetSeries,
DecayFactor, WindowLength)
```

Arguments

RetSeries	Return Series: number of observations (NUMOBS) by number of assets (NASSETS) matrix of equally spaced incremental return observations. The first row is the oldest observation, and the last row is the most recent.
DecayFactor	(Optional) Controls how much less each observation is weighted than its successor. The k th observation back in time has weight DecayFactor^k . DecayFactor must lie in the range: $0 < \text{DecayFactor} \leq 1$. Default = 1, the equally weighted linear moving average model (BIS).
WindowLength	(Optional) Number of recent observations in the computation. Default = NUMOBS.

Description

`[ExpReturn, ExpCovariance, NumEffObs] = ewstats(RetSeries, DecayFactor, WindowLength)` computes estimated expected returns, estimated covariance matrix, and the number of effective observations. These are maximum likelihood estimates which are generally biased.

`ExpReturn` is a 1-by-NASSETS vector of estimated expected returns.

`ExpCovariance` is an NASSETS-by-NASSETS estimated covariance matrix. The standard deviations of the asset return processes are given by

```
STDVec = sqrt(diag(ExpCovariance))
```

The correlation matrix is

```
CorrMat = ExpCovariance./( STDVec*STDVec' )
```

NumEffObs is the number of effective observations = $(1 - \text{DecayFactor}^{\text{WindowLength}}) / (1 - \text{DecayFactor})$.

A smaller `DecayFactor` or `WindowLength` emphasizes recent data more strongly but uses less of the available data set.

Examples

Compute Estimated Expected Returns and Estimated Covariance Matrix

This example shows how to compute the estimated expected returns and the estimated covariance matrix.

```
RetSeries = [ 0.24 0.08
              0.15 0.13
              0.27 0.06
              0.14 0.13 ];
```

```
DecayFactor = 0.98;
```

```
[ExpReturn, ExpCovariance] = ewstats(RetSeries, DecayFactor)
```

```
ExpReturn =
```

```
0.1995    0.1002
```

```
ExpCovariance =
```

```
0.0032    -0.0017
-0.0017    0.0010
```

See Also

[cov](#) | [mean](#)

exp

Exponential values

Syntax

```
newfts = exp(tsobj)
```

Description

`newfts = exp(tsobj)` calculates the natural exponential (base e) of all the data in the data series of the financial time series object `tsobj` and returns the result in the object `newfts`.

See Also

`log` | `log2` | `log10`

extfield

Data series extraction

Syntax

```
ftse = extfield(tsoj, fieldnames)
```

Arguments

<code>tsoj</code>	Financial time series object
<code>fieldnames</code>	Data series to be extracted. A cell array if a list of data series names (<code>fieldnames</code>) is supplied. A string if only one is wanted.

Description

`ftse = extfield(tsoj, fieldnames)` extracts from `tsoj` the dates and data series specified by `fieldnames` into a new financial time series object `ftse`. `ftse` has all the dates in `tsoj` but contains a smaller number of data series.

Examples

`extfield` is identical to referencing a field in the object. For example,

```
ftse = extfield(fts, 'Close')
```

is the same as

```
ftse = fts.Close
```

This function is the complement of the function `rmfield`.

See Also

`rmfield`

fanplot

Plot combined historical and forecast data to visualize possible outcomes

Syntax

```
fanplot(historical, forecast)
```

Description

`fanplot(historical, forecast)` generates a fan chart. In time series analysis, a fan chart is a chart that joins a simple line chart for observed past data with ranges for possible values of future data. The historical data and possible future data are joined together with a line showing a central estimate or most likely value for the future outcomes.

`fanplot` supports three plotting scenarios:

- **Matching** — This scenario occurs when the time period perfectly matches for historical and forecast data.
- **Backtest** — This scenario occurs when there are overlaps between historical and forecast data.
- **Gap** — This scenario occurs when there are NaN values in the historical or forecast data.

Examples

Create a Fan Plot Using Cell Array Data

Define the data inputs for `historical` as a 5-by-2 cell array and `forecast` as a 5-by-21 cell array with 20 observations.

```
historical = {[2006]    [-0.0110]  
             [2007]    [ 0.0120]  
             [2008]    [ 0.0090]  
             [2009]    [ 0.0120]  
             [2010]    [ 0.0150]};
```

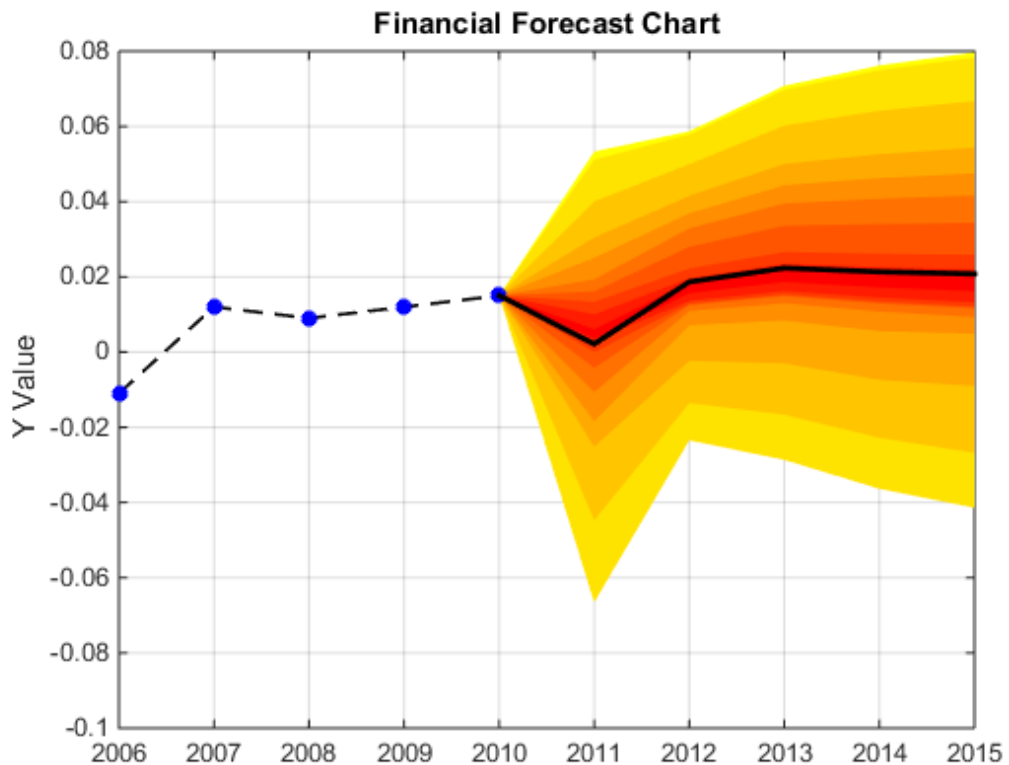
```

forecast = {[2011] [0.0203] [-0.0155] [0.0311] [-0.0026] [0.0035]
            [0.0533] [0.0139] [0.0037] [-0.0727] [-0.0291]
            [2012] [0.0430] [-0.0094] [0.0587] [0.0095] [0.0185]
            [0.0141] [0.0337] [0.0187] [0.0132] [-0.0292]
            [2013] [0.0518] [-0.0116] [0.0708] [0.0112] [0.0221]
            [0.0168] [0.0405] [0.0224] [0.0157] [-0.0356]
            [2014] [0.0546] [-0.0171] [0.0762] [0.0088] [0.0210]
            [0.0151] [0.0419] [0.0214] [0.0139] [-0.0442]
            [2015] [0.0565] [-0.0207] [0.0797] [0.0072] [0.0203]
            [0.0139] [0.0428] [0.0207] [0.0126] [-0.0499]}

```

Generate the fan plot.

```
fanplot (historical, forecast);
```



The dotted points are the historical lines and the filled lines indicate the mean for the forecasts. This fanplot represents a matching scenario where the time period perfectly matches for the historical and forecast data.

Create a Fan Plot Using Matrix Data

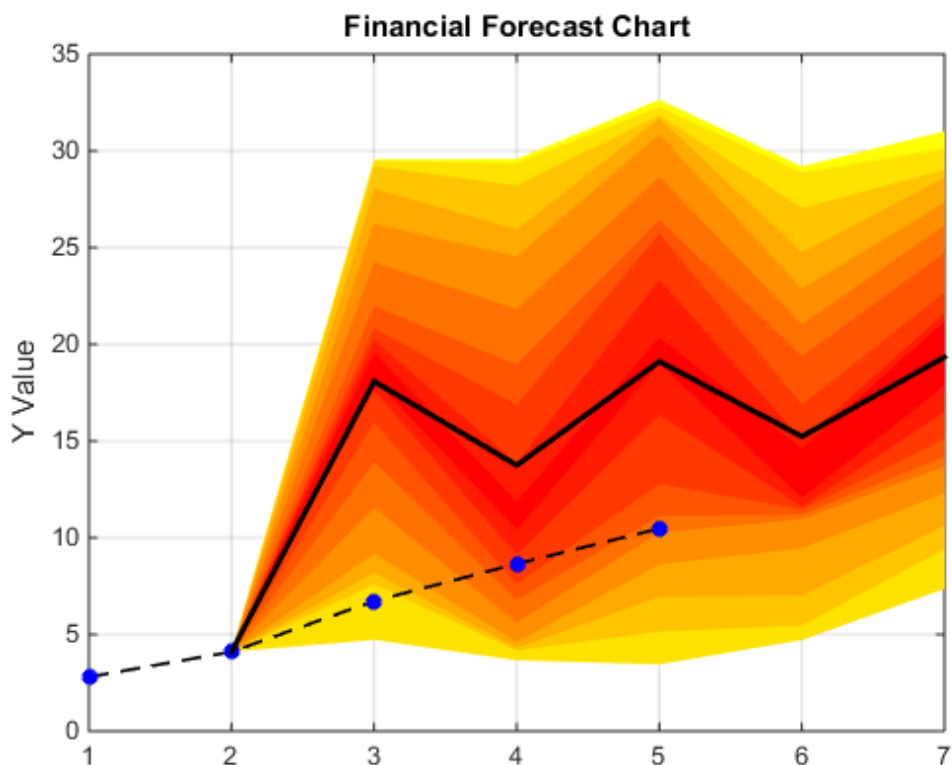
Define the data inputs for `historical` as a 5-by-2 matrix and `forecast` as a 5-by-21 matrix with 20 observations.

```
historical = [ 1.0000    2.8046 ;
              2.0000    4.1040 ;
              3.0000    6.7292 ;
              4.0000    8.6486 ;
              5.0000   10.4747 ];

forecast = [ 3.0000   28.9874   18.3958   19.6376   29.5627    8.3462    7.1502   29.
            20.8557   27.0691   23.0803   20.7885   18.0205   17.2294   10.
              4.0000    4.8933   27.2659    7.2206   24.4703   10.5895   15.0212   29.
              8.0007   18.7114   19.1691   24.5963    4.2835    4.0676    3.
              5.0000   20.9732   19.7069   11.6862   25.7018   31.8940    7.2664   19.
              31.7996    3.6419    3.2695   27.1422   10.5487   32.6529   18.
              6.0000   11.0069   29.1965    4.5551   20.2627   10.9209   15.2675   28.
              23.9532   18.4804   25.5484    4.8747    8.0036   11.5329   11.
              7.0000    5.9699   11.1486   26.0449   13.4619   21.1196   28.8068   28.
              21.2390   29.2396   18.4828   28.3945   21.9342   14.4642   17.
```

Generate the fan plot.

```
fanplot (historical, forecast);
```



The dotted points are the historical lines and the filled lines indicate the mean for the forecasts. This fanplot represents a backtest scenario where there is an overlap between the historical and forecast data.

Input Arguments

historical — Historical dates and data

matrix | cell array

Historical dates and data, specified as an N-by-2 matrix or cell array, where the first column is the date, and the second column is the data associated for that date. N indicates the number of dates.

Data Types: `cell` | `double`

forecast — Forecast dates and data

matrix | cell array

Forecast dates and data, specified as an N-by-M matrix or cell array, where the first column is the date, and the second to the last columns are the data observations. N indicates the number of the dates and M is the number for data observations.

Data Types: `cell` | `double`

See Also

`bolling` | `candle` | `highlow` | `linebreak` | `movavg` | `pointfig` | `priceandvol` | `renko` | `volarea`

fbusdate

First business date of month

Syntax

Date = fbusdate(Year, Month, Holiday, Weekend)

Arguments

Year	Enter as four-digit integer.
Month	Enter as integer from 1 to 12.
Holiday	(Optional) Vector of holidays and nontrading-day dates. All dates in Holiday must be the same format: either serial date numbers or date strings. (Using date numbers improves performance.) The holidays function supplies the default vector.
Weekend	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), then Weekend = [1 0 0 0 0 0 1].

Description

Date = fbusdate(Year, Month, Holiday, Weekend) returns the serial date number for the first business date of the given year and month. **Holiday** specifies nontrading days.

Year and **Month** can contain multiple values. If one contains multiple values, the other must contain the same number of values or a single value that applies to all. For example, if **Year** is a 1-by-n vector of integers, then **Month** must be a 1-by-n vector of integers or a single integer. **Date** is then a 1-by-n vector of date numbers.

Use the function **datestr** to convert serial date numbers to formatted date strings.

Examples

Return a Serial Date Number for the First Business Date

This example shows how to return serial date numbers for the first business date, given year and month.

```
Date = fbusdate(2001, 11)
datestr(Date)
```

```
Year = [2002 2003 2004];
Date = fbusdate(Year, 11)
datestr(Date)
```

```
Date =
```

```
731156
```

```
ans =
```

```
01-Nov-2001
```

```
Date =
```

```
731521      731888      732252
```

```
ans =
```

```
01-Nov-2002
```

```
03-Nov-2003
```

```
01-Nov-2004
```

Return a Serial Date Number for the First Business Date Using the Weekend Argument

This example shows how to return serial date numbers for the first business date, given year and month, and also indicate that Saturday is a business day by setting the

Weekend argument. March 1, 2003, is a Saturday. Use `fbusdate` to check that this Saturday is actually the first business day of the month.

```
Weekend = [1 0 0 0 0 0 0];  
Date = datestr(fbusdate(2003, 3, [], Weekend))
```

```
Date =
```

```
01-Mar-2003
```

See Also

`busdate` | `eomdate` | `holidays` | `isbusday` | `lbusdate`

fetch

Data from financial time series object

Syntax

```
newfts = fetch(oldfts, StartDate, StartTime, EndDate, EndTime,
delta, dmy_specifier, time_ref)
```

Arguments

<code>oldfts</code>	Existing financial time series object.
<code>StartDate</code>	First date in the range from which data is to be extracted.
<code>StartTime</code>	Beginning time on each day. If you do not require specific times or <code>oldfts</code> does not contain time information, use <code>[]</code> . If you specify <code>StartTime</code> , you must also specify <code>EndTime</code> .
<code>EndDate</code>	Last date in the range from which data is to be extracted.
<code>EndTime</code>	Ending time on each day. If you do not require specific times or <code>oldfts</code> does not contain time information, use <code>[]</code> . If you specify <code>EndTime</code> , you must also specify <code>StartTime</code> .
<code>delta</code>	Skip interval. Can be any positive integer. Units for the skip interval specified by <code>dmy_specifier</code> .
<code>dmy_specifier</code>	Specifies the units for <code>delta</code> . Can be <ul style="list-style-type: none"> • D, d (Days) • M, m (Months) • Y, y (Years)
<code>time_ref</code>	Time reference intervals or specific times. Valid time reference intervals are 1, 5, 15, or 60 minutes. Enter specific times as <code>'hh:mm'</code> .

Description

`newfts = fetch(oldfts, StartDate, StartTime, EndDate, EndTime, delta, dmy_specifier, time_ref)` requests data from a financial time series object beginning from the start date and/or start time to the end date and/or end time, skipping a specified number of days, months, or years.

Note If time information is present in `oldfts`, using `[]` for start or end times results in `fetch` returning all instances of a specific date.

Examples

Example 1. Create a financial time series object containing both dates and times:

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
         '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates,1),1), ...
                      times]);
myFts = fints(dates_times, (1:6)', {'Data1'}, 1, 'My first FINTS')
```

```
myFts =
```

```
desc: My first FINTS
freq: Daily (1)

      'dates: (6)'      'times: (6)'      'Data1: (6)'
      '01-Jan-2001'    '11:00'          [          1]
      '      "      '    '12:00'          [          2]
      '02-Jan-2001'    '11:00'          [          3]
      '      "      '    '12:00'          [          4]
      '03-Jan-2001'    '11:00'          [          5]
      '      "      '    '12:00'          [          6]
```

To fetch all dates and times from this financial time series, enter

```
fetch(myFts, '01-Jan-2001', [], '03-Jan-2001', [], 1, 'd')
```

or

```
fetch(myFts, '01-Jan-2001', '11:00', '03-Jan-2001', '12:00', 1, 'd')
```

These commands reproduce the entire time series shown above.

To fetch every other day's data, enter

```
fetch(myFts, '01-Jan-2001', [], '03-Jan-2001', [], 2, 'd')
```

This returns:

```
ans =
```

```
desc: My first FINTS
freq: Daily (1)

'dates: (4)'      'times: (4)'      'Data1: (4)'
'01-Jan-2001'    '11:00'           [          1]
'    "    '    '12:00'           [          2]
'03-Jan-2001'    '11:00'           [          5]
'    "    '    '12:00'           [          6]
```

Example 2. Create a financial time series object with time intervals of less than 1 hour:

```
dates2 = ['01-Jan-2001'; '01-Jan-2001'; '01-Jan-2001'; ...
'02-Jan-2001'; '02-Jan-2001'; '02-Jan-2001'];
times2 = ['11:00'; '11:05'; '11:06'; '12:00'; '12:05'; '12:06'];
dates_times2 = cellstr([dates2, repmat(' ', size(dates2,1), 1), ...
times2]);
myFts2 = fints(dates_times2, (1:6), {'Data1'}, 1, 'My second
FINTS')
```

```
myFts2 =
```

```
desc: My second FINTS
freq: Daily (1)

'dates: (6)'      'times: (6)'      'Data1: (6)'
'01-Jan-2001'    '11:00'           [          1]
'    "    '    '11:05'           [          2]
'    "    '    '11:06'           [          3]
'02-Jan-2001'    '12:00'           [          4]
'    "    '    '12:05'           [          5]
'    "    '    '12:06'           [          6]
```

Use `fetch` to extract data from this time series object at 5-minute intervals for each day starting at 11:00 o'clock on January 1, 2001.

```
fetch(myFts2, '01-Jan-2001', [], '02-Jan-2001', [], 1, 'd', 5)
```

```
desc: My second FINTS
freq: Daily (1)
```

```

'dates: (4)'      'times: (4)'      'Data1: (4)'
'01-Jan-2001'    '11:00'          [          1]
'      "      '  '11:05'          [          2]
'02-Jan-2001'    '12:00'          [          4]
'      "      '  '12:05'          [          5]

```

You can use this version of `fetch` to extract data at specific times. For example, to fetch data only at 11:06 and 12:06 from `myFts2`, enter

```

fetch(myFts2, '01-Jan-2001', [], '02-Jan-2001', [], 1, 'd', ...
{'11:06'; '12:06'})

```

ans =

```

desc: My second FINTS
freq: Daily (1)

'dates: (2)'      'times: (2)'      'Data1: (2)'
'01-Jan-2001'    '11:06'          [          3]
'02-Jan-2001'    '12:06'          [          6]

```

See Also

`extfield` | `ftsbound` | `getfield` | `subsref`

fieldnames

Get names of fields

Syntax

```
fnames = fieldnames(tsoobj)
```

```
fnames = fieldnames(tsoobj, srsnameonly)
```

Arguments

<code>tsoobj</code>	Financial time series object
<code>srsnameonly</code>	Field names returned: 0 = All field names (default). 1 = Data series names only.

Description

`fieldnames` gets field names in a financial time series object.

`fnames = fieldnames(tsoobj)` returns the field names associated with the financial time series object `tsoobj` as a cell array of strings, including the common fields: `desc`, `freq`, `dates` (and `times` if present).

`fnames = fieldnames(tsoobj, srsnameonly)` returns field names depending upon the setting of `srsnameonly`. If `srsnameonly` is 0, the function returns all field names, including the common fields: `desc`, `freq`, `dates`, and `times`. If `srsnameonly` is set to 1, `fieldnames` returns only the data series in `fnames`.

See Also

`chfield` | `getfield` | `isfield` | `rmfield` | `setfield`

fillts

Fill missing values in time series

Syntax

```
newfts = fillts(oldfts, fill_method)
newfts = fillts(oldfts, fill_method, newdates)
newfts = fillts(oldfts, fill_method,
newdates, {'T1','T2',...})
newfts = fillts(oldfts, fill_method, newdates,
'SPAN', {'TS','TE'}, delta)
newfts = fillts(... sortmode)
```

Arguments

<code>oldfts</code>	Financial time series object.
<code><i>fill_method</i></code>	<p>(Optional) Replaces missing values (NaN) in <code>oldfts</code> using an interpolation process, a constant, or a zero-order hold.</p> <p>Valid fill methods (interpolation methods) are:</p> <ul style="list-style-type: none"> • linear - 'linear' - 'l' (default) • linear with extrapolation - 'linearExtrap' - 'le' • cubic - 'cubic' - 'c' • cubic with extrapolation - 'cubicExtrap' - 'ce' • spline - 'spline' - 's' • spline with extrapolation - 'splineExtrap' - 'se' • nearest - 'nearest' - 'n' • nearest with extrapolation - 'nearestExtrap' - 'ne' • pchip - 'pchip' - 'p' • pchip with extrapolation - 'pchipExtrap' - 'pe'

	(See <code>interp1</code> for a discussion of extrapolation.) To fill with a constant, enter that constant. A zero-order hold ('zero') fills a missing value with the value immediately preceding it. If the first value in the time series is missing, it remains a NaN.
<code>newdates</code>	(Optional) Column vector of serial dates, a date string, or a column cell array of date strings. If <code>oldfts</code> contains time of day information, <code>newdates</code> must be accompanied by a time vector (<code>newtimes</code>). Otherwise, <code>newdates</code> is assumed to have times of '00:00'.
<code>T1, T2, TS, TE</code>	First time, second time, start time, end time
<code>delta</code>	Time interval in minutes to span between the start time and end time
<code>sortmode</code>	(Optional) Default = 0 (unsorted). 1 = sorted.

Description

`newfts = fillts(oldfts, fill_method)` replaces missing values (represented by NaN) in the financial time series object `oldfts` with real values, using either a constant or the interpolation process indicated by `fill_method`.

`newfts = fillts(oldfts, fill_method, newdates)` replaces all the missing values on the specified dates `newdates` added to the financial time series `oldfts` with new values. The values can be a single constant or values obtained through the interpolation process designated by `fill_method`. If any of the dates in `newdates` exists in `oldfts`, the existing one has precedence.

`newfts = fillts(oldfts, fill_method, newdates, {'T1', 'T2', ...})` additionally allows the designation of specific times of day for addition or replacement of data.

`newfts = fillts(oldfts, fill_method, newdates, 'SPAN', {'TS', 'TE'}, delta)` is similar to the previous format except that you designate only a start time and an end time. You follow these times with a spanning time interval, `delta`.

If you specify only one date for `newdates`, specifying a start and end time generates only times for that specific date.

`newfts = fillts(... sortmode)` additionally denotes whether you want the order of the dates in the output object to stay the same as in the input object or to be sorted chronologically.

`sortmode = 0` (unsorted) appends any new dates to the end. The interpolation and zero-order processes that calculate the values for the new dates work on a sorted object. Upon completion, the existing dates are reordered as they were originally, and the new dates are appended to the end.

`sortmode = 1` sorts the output. After interpolation, no reordering of the date sequence occurs.

Examples

Example 1. Create a financial time series object with missing data in the fourth and fifth rows.

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
         '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates,1),1), ...
                       times]);
OpenFts = fints(dates_times, [(1:3)'; nan; nan; 6], {'Data1'}, 1, ...
               'Open Financial Time Series');
```

`OpenFts` looks like this:

```
OpenFts =

  desc: Open Financial Time Series
  freq: Daily (1)

  'dates: (6)'   'times: (6)'   'Data1: (6)'
  '01-Jan-2001' '11:00'         [          1]
  '   "         '12:00'         [          2]
  '02-Jan-2001' '11:00'         [          3]
  '   "         '12:00'         [         NaN]
  '03-Jan-2001' '11:00'         [         NaN]
  '   "         '12:00'         [          6]
```

Example 2. Fill the missing data in `OpenFts` using cubic interpolation.

```
FilledFts = fillfts(OpenFts, 'cubic')  
  
FilledFts =  
  
      desc: Filled Open Financial Time Series  
      freq: Unknown (0)  
  
      'dates: (6)'      'times: (6)'      'Data1: (6)'  
      '01-Jan-2001'    '11:00'          [          1]  
      '   "   "   '    '12:00'          [          2]  
      '02-Jan-2001'    '11:00'          [          3]  
      '   "   "   '    '12:00'          [    3.0663]  
      '03-Jan-2001'    '11:00'          [    5.8411]  
      '   "   "   '    '12:00'          [    6.0000]
```

Example 3. Fill the missing data in `OpenFts` with a constant value.

```
FilledFts = fillfts(OpenFts, 0.3)  
  
FilledFts =  
  
      desc: Filled Open Financial Time Series  
      freq: Unknown (0)  
  
      'dates: (6)'      'times: (6)'      'Data1: (6)'  
      '01-Jan-2001'    '11:00'          [          1]  
      '   "   "   '    '12:00'          [          2]  
      '02-Jan-2001'    '11:00'          [          3]  
      '   "   "   '    '12:00'          [    0.3000]  
      '03-Jan-2001'    '11:00'          [    0.3000]  
      '   "   "   '    '12:00'          [          6]
```

Example 4. You can use `fillfts` to identify a specific time on a specific day for the replacement of missing data. This example shows how to replace missing data at 12:00 on January 2 and 11:00 on January 3.

```
FilltimeFts = fillfts(OpenFts, 'c', ...  
{'02-Jan-2001'; '03-Jan-2001'}, {'12:00'; '11:00'}, 0)  
  
FilltimeFts =  
  
      desc: Filled Open Financial Time Series  
      freq: Unknown (0)  
  
      'dates: (6)'      'times: (6)'      'Data1: (6)'
```

```

'01-Jan-2001' '11:00' [ 1]
'   "   '12:00' [ 2]
'02-Jan-2001' '11:00' [ 3]
'   "   '12:00' [ 3.0663]
'03-Jan-2001' '11:00' [ 5.8411]
'   "   '12:00' [ 6.0000]

```

Example 5. Use a spanning time interval to add an additional day to OpenFts.

```
SpanFts = fillts(OpenFts, 'c', '04-Jan-2001', 'span', ...
                {'11:00'; '12:00'}, 60, 0)
```

```
SpanFts =
```

```

desc: Filled Open Financial Time Series
freq: Unknown (0)

'dates: (8)'   'times: (8)'   'Data1: (8)'
'01-Jan-2001' '11:00'   [ 1]
'   "   '12:00'   [ 2]
'02-Jan-2001' '11:00'   [ 3]
'   "   '12:00'   [ 3.0663]
'03-Jan-2001' '11:00'   [ 5.8411]
'   "   '12:00'   [ 6.0000]
'04-Jan-2001' '11:00'   [ 9.8404]
'   "   '12:00'   [ 9.9994]

```

See Also

interp1

filter

Linear filtering

Syntax

```
newfts = filter(B, A, oldfts)
```

Description

`filter` filters an entire financial time series object with certain filter specifications. The filter is specified in a transfer function expression.

`newfts = filter(B, A, oldfts)` filters the data in the financial time series object `oldfts` with the filter described by vectors `A` and `B` to create the new financial time series object `newfts`. The filter is a “Direct Form II Transposed” implementation of the standard difference equation. `newfts` is a financial time series object containing the same data series (names) as the input `oldfts`.

See Also

`filter` | `filter2`

fints

Construct financial time series object

Syntax

```
tsobj = fints(dates_and_data)
```

```
tsobj = fints(dates, data)
```

```
tsobj = fints(dates, data, datanames)
```

```
tsobj = fints(dates, data, datanames, freq)
```

```
tsobj = fints(dates, data, datanames, freq, desc)
```

Arguments

<p><code>dates_and_data</code></p>	<p>Column-oriented matrix containing one column of dates and a single column for each series of data. In this format, dates must be entered in serial date number format. If the input serial date numbers encode time-of-day information, the output object contains a column labeled <code>'dates'</code> containing the date information and another labeled <code>'times'</code> containing the time information.</p> <p>You can use the function <code>today</code> to enter date information or the function <code>now</code> to enter date with time information.</p>
<p><code>dates</code></p>	<p>Column vector of dates. Dates can be date strings or serial date numbers and can include time of day information. When entering time-of-day information as serial date numbers, the entry must be a column-oriented matrix when multiple entries are present. If the time-of-day information is in string format, the entry must be a column-oriented cell array of dates and times when multiple entries are present.</p> <p>Valid date and time string formats are:</p> <ul style="list-style-type: none"> • <code>'ddmmyy hh:mm'</code> or <code>'ddmmyyyy hh:mm'</code>

	<ul style="list-style-type: none"> • 'mm/dd/yy hh:mm' or 'mm/dd/yyyy hh:mm' • 'dd-mmm-yy hh:mm' or 'dd-mmm-yyyy hh:mm' • 'mmm.dd,yy hh:mm' or 'mmm.dd,yyyy hh:mm' <p>Dates and times can initially be separate column-oriented vectors, but they must be concatenated into a single column-oriented matrix before being passed to <code>fints</code>. You can use the MATLAB functions <code>today</code> and <code>now</code> to assist in entering date and time information.</p>
<code>data</code>	Column-oriented matrix containing a column for each series of data. The number of values in each data series must match the number of dates. If a mismatch occurs, MATLAB does not generate the financial time series object, and you receive an error message.
<code>datanames</code>	<p>Cell array of data series names. Overrides the default data series names. Default data series names are <code>series1</code>, <code>series2</code>, and so on.</p> <p>Note: Not all strings are accepted as <code>datanames</code> parameters. Supported data series names cannot start with a number and must contain only these characters:</p> <ul style="list-style-type: none"> • Lowercase Latin alphabet, a to z • Uppercase Latin alphabet, A to Z • Underscore, <code>_</code>

freq	<p>Frequency indicator. Allowed values are:</p> <p>UNKNOWN, Unknown, unknown, U, u, 0</p> <p>DAILY, Daily, daily, D, d, 1</p> <p>WEEKLY, Weekly, weekly, W, w, 2</p> <p>MONTHLY, Monthly, monthly, M, m, 3</p> <p>QUARTERLY, Quarterly, quarterly, Q, q, 4</p> <p>SEMIANNUAL, Semiannual, semiannual, S, s, 5</p> <p>ANNUAL, Annual, annual, A, a, 6</p> <p>Default = Unknown.</p>
desc	<p>String providing descriptive name for financial time series object. Default = ''.</p>

Description

`fints` constructs a financial time series object. A financial time series object is a MATLAB object that contains a series of dates and one or more series of data. Before you perform an operation on the data, you must set the frequency indicator (`freq`). You can optionally provide a description (`desc`) for the time series.

`tsobj = fints(dates_and_data)` creates a financial time series object containing the dates and data from the matrix `dates_and_data`. If the dates contain time-of-day information, the object contains an additional series of times. The date series and each data series must each be a column in the input matrix. The names of the data series default to `series1`, ..., `seriesn`. The `desc` and `freq` fields are set to their defaults.

`tsobj = fints(dates, data)` generates a financial time series object containing dates from the `dates` column vector of dates and data from the matrix `data`. If the dates contain time-of-day information, the object contains an additional series of times. The data matrix must be column-oriented, that is, each column in the matrix is a data series. The names of the series default to `series1`, ..., `seriesn`, where n is the total number of columns in `data`. The `desc` and `freq` fields are set to their defaults.

`tsobj = fints(dates, data, datanames)` additionally allows you to rename the data series. The names are specified in the `datanames` cell array. The number of strings in `datanames` must correspond to the number of columns in `data`. The `desc` and `freq` fields are set to their defaults.

`tsobj = fints(dates, data, datanames, freq)` additionally sets the frequency when you create the object. The `desc` field is set to its default `''`.

`tsobj = fints(dates, data, datanames, freq, desc)` provides a description string for the financial time series object.

Note: `fints` only supports hourly and minute time series. Seconds are not supported and will be disregarded when the `fints` object is created (i.e. 01-jan-2001 12:00:01 will be considered as 01-jan-2001 12:00). If there are duplicate dates and times, the `fints` constructor will sort the dates and times and choose the first instance of the duplicate dates and times. The other duplicate dates and times will be removed from the object along with their corresponding data.

Examples

Create a Financial Time Series Object Containing Days and Data

Define the data:

```
data = [1:6]'
```

```
data =
```

```
1  
2  
3  
4  
5  
6
```

Define the dates:

```
dates = [today:today+5]'
```



```
dates =  
    735862  
    735863  
    735864  
    735865  
    735866  
    735867
```

Create the financial times series object:

```
tsobjkt = fints(dates, data)
```

```
tsobjkt =  
    desc: (none)  
    freq: Unknown (0)  
  
    'dates: (6)'      'series1: (6)'  
    '20-Sep-2014'    [          1]  
    '21-Sep-2014'    [          2]  
    '22-Sep-2014'    [          3]  
    '23-Sep-2014'    [          4]  
    '24-Sep-2014'    [          5]  
    '25-Sep-2014'    [          6]
```

Create a Financial Time Series Object Containing Days, Time of Day, and Data

Define the data:

```
data = [1:6]'
```

```
data =  
    1  
    2  
    3  
    4  
    5  
    6
```

Define the dates:

```
dates = [now:now+5]'
```

```
dates =
```

```
1.0e+05 *  
7.3586  
7.3586  
7.3586  
7.3587  
7.3587  
7.3587
```

Create the financial times series object:

```
tsobjkt = fints(dates, data)
```

```
tsobjkt =
```

```
desc: (none)  
freq: Unknown (0)  
  
'dates: (6)'      'times: (6)'      'series1: (6)'  
'20-Sep-2014'    '01:03'           [          1]  
'21-Sep-2014'    '01:03'           [          2]  
'22-Sep-2014'    '01:03'           [          3]  
'23-Sep-2014'    '01:03'           [          4]  
'24-Sep-2014'    '01:03'           [          5]  
'25-Sep-2014'    '01:03'           [          6]
```

Create a Financial Time Series Object From a Single Input for Dates and Times

Define the dates and times:

```
dates_and_times = (now:now+5)'
```

```
dates_and_times =
```

```

1.0e+05 *
7.3586
7.3586
7.3586
7.3587
7.3587
7.3587

```

Create the financial times series object:

```
f = fints(dates_and_times, randn(6,1))
```

```
f =
```

```

desc: (none)
freq: Unknown (0)

'dates: (6)'      'times: (6)'      'series1: (6)'
'20-Sep-2014'    '01:09'           [      0.5377]
'21-Sep-2014'    '01:09'           [      1.8339]
'22-Sep-2014'    '01:09'           [     -2.2588]
'23-Sep-2014'    '01:09'           [      0.8622]
'24-Sep-2014'    '01:09'           [      0.3188]
'25-Sep-2014'    '01:09'           [     -1.3077]

```

This generates a financial time series object, `f`, and obtains the dates and data from the matrix `dates_and_times`. The dates and times in the input matrix must be oriented column-wise (i.e. the date series and each time series are columns in the input matrix). In addition, the dates entered must be in the serial date format (i.e. 01-Jan-2001 is 730852). You can also use the function `now` to enter in date information. The names of the series will default to `'series1'`, ..., `'|seriesN|'` where `N` is the total number of columns in `dates_and_times` less 1 (that is the number of data columns). The default contents of the `desc` and `freq` fields are `|"` and `|'Unknown'| (0)`, respectively.

See Also

`datenum` | `datestr` | `ftstool` | `ftsgui`

floatdiscmargin

Discount margin for floating-rate bond

Syntax

```
Margin = floatdiscmargin(Price,Spread,Settle,Maturity,  
RateInfo,LatestFloatingRate)
```

```
Margin = floatdiscmargin(Price,Spread,Settle,Maturity,  
RateInfo,LatestFloatingRate,Name,Value)
```

Description

`Margin = floatdiscmargin(Price,Spread,Settle,Maturity,RateInfo,LatestFloatingRate)` calculates the discount margin or zero discount margin for a floating-rate bond.

`Margin = floatdiscmargin(Price,Spread,Settle,Maturity,RateInfo,LatestFloatingRate,Name,Value)` calculates the discount margin or zero discount margin for a floating-rate bond with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

Price

NINST-by-1 matrix of bond prices where the discount margin is to be computed. The spread is calculated against the clean price (the function internally does not add the accrued interest to the price specified by the `Price` input). If the spread is required against the dirty price, you must supply the dirty price to this input.

Spread

NINST-by-1 matrix for the number of basis points over the reference rate.

Settle

Settlement date of the floating-rate bond(s). If supplied as a NINST-by-1 vector of dates, all settlement dates must be the same (only a single settlement date is supported).

Maturity

NINST-by-1 vector of dates representing the maturity date of the floating-rate bond.

RateInfo

Interest-rate information. This argument determines if the discount margin or zero discount margin is calculated. If this input is a NINST-by-2 vector, the discount margin is calculated where the:

- First column is the stub rate between the settlement date and the first coupon rate.
- Second column is the reference rate for the term of the floating coupons (for example, the 3-month LIBOR from the settlement date for a bond with `Reset` of 4).

Note: If this argument is an annualized zero-rate term structure created by `intenvset`, the zero discount margin is calculated.

LatestFloatingRate

NINST-by-1 vector representing the rate for the next floating payment set at the last reset date.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1, . . . ,NameN,ValueN`.

'Reset'

NINST-by-1 vector representing the frequency of payments per year.

Default: 1

'Basis'

NINST-by-1 vector representing the basis used for time factor calculations:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see “”.

Default: 0 (actual/actual)

'Principal'

NINST-by-1 of notional principal amounts or NINST-by-1 cell array where each element is a NUMDATES-by-2 cell array where the first column is dates and the second column is the associated principal amount. The date indicates the last day that the principal value is valid.

Default: 100

'EndMonthRule'

NINST-by-1 vector representing the end-of-month rule.

Default: 1 (in effect)

'AdjustCashFlowsBasis'

NINST-by-1 vector of logicals. Adjusts cash flows according to the accrual amount.

Note: Usually you want to set `AdjustCashFlowsBasis` to `true`, so cash flows are calculated with adjustments on accrual amounts. The default is set to `false` to be consistent with `floatbyzero`.

Default: `false`

'Holidays'

NHOLIDAYS-by-1 vector of MATLAB date numbers. Holidays are used in computing business days.

Default: `holidays.m` used

'BusinessDayConvention'

NINST-by-1 cell array of business day convention to be used in computing payment dates. Values are:

- `actual`
- `follow`
- `modifiedfollow`
- `previous`
- `modifiedprevious`

Default: `actual`

Output Arguments

Margin

NINST-by-1 vector of the discount margin if `RateInfo` is specified as a NINST-by-2 vector of stub and spot rates. NINST-by-NCURVES matrix of the zero discount margin if `RateInfo` is specified as an annualized zero rate term structure created by `intenvset`.

Examples

Compute the Zero Discount Margin Using a Yield Curve

Use `floatdiscmargin` to compute the discount margin and zero discount margin for a floating-rate note.

Define data for the floating-rate note.

```
Price = 99.99;
Spread = 50;
Settle = '20-Jan-2011';
Maturity = '15-Jan-2012';
LatestFloatingRate = 0.05;
StubRate = 0.049;
SpotRate = 0.05;
Reset = 4;
Basis = 2;
```

Compute the discount margin.

```
dMargin = floatdiscmargin(Price, Spread, Settle, Maturity, ...
[StubRate, SpotRate], LatestFloatingRate, 'Reset', Reset, 'Basis', Basis, ...
'AdjustCashFlowsBasis', true)
```

```
dMargin =
    48.4810
```

Usually you want to set `AdjustCashFlowsBasis` to `true`, so cash flows are calculated with adjustments on accrual amounts.

Create an annualized zero-rate term structure to calculate the zero discount margin.

```
Rates = [0.0500;
         0.0505;
         0.0510;
         0.0520];
StartDates = ['20-Jan-2011';
              '15-Apr-2011';
              '15-Jul-2011';
              '15-Oct-2011'];
```



```

EndDates = [ '15-Apr-2011';
              '15-Jul-2011';
              '15-Oct-2011';
              '15-Jan-2012' ];
ValuationDate = '20-Jan-2011';
RateSpec = intenvset('Compounding', Reset, 'Rates', Rates, ...
                    'StartDates', StartDates, 'EndDates', EndDates, ...
                    'ValuationDate', ValuationDate, 'Basis', Basis);

```

Calculate the zero discount margin using the previous yield curve.

```

dMargin = floatdiscmargin(Price, Spread, Settle, Maturity, ...
                          RateSpec, LatestFloatingRate, 'Reset', Reset, 'Basis', Basis, ...
                          'AdjustCashFlowsBasis', true)

```

```

dMargin =
    46.0688

```

References

Fabozzi, Frank J., Mann, Steven V., *Floating-Rate Securities*, John Wiley and Sons, New York, 2000.

Fabozzi, Frank J., Mann, Steven V., *Introduction to Fixed Income Analytics: Relative Value Analysis, Risk Measures and Valuation*, John Wiley and Sons, New York, 2010.

O'Kane, Dominic, Sen, Saurav, *Credit Spreads Explained*, Lehman Brothers Fixed Income Quantitative Research, March 2004.

See Also

| floatmargin | floatbyzero | bndspread | intenvset

floatmargin

Margin measures for floating-rate bond

Syntax

```
[Margin,AdjPrice] = floatmargin(Price,Spread,Settle,  
Maturity)  
[Margin,AdjPrice] = floatmargin(Price,Spread,Settle,  
Maturity,Name,Value)
```

Description

[Margin,AdjPrice] = floatmargin(Price,Spread,Settle, Maturity) calculates margin measures for a floating-rate bond.

[Margin,AdjPrice] = floatmargin(Price,Spread,Settle, Maturity,Name,Value) calculates margin measures for a floating-rate bond with additional options specified by one or more Name,Value pair arguments.

Use `floatmargin` to calculate the following types of margin measures for a floating-rate bond:

- Spread for life
- Adjusted simple margin
- Adjusted total margin

To calculate the discount margin or zero discount margin, see `floatdiscmargin`.

Input Arguments

Price

NINST-by-1 matrix of bond prices where spreads are to be computed.

Spread

NINST-by-1 matrix for the number of basis points over the reference rate.

Settle

Settlement date of the floating-rate bond(s). If supplied as a NINST-by-1 vector of dates, all settlement dates must be the same (only a single settlement date is supported).

Maturity

NINST-by-1 vector of dates representing the maturity date of the floating-rate bond.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

'SpreadType'

The type of spread to calculate. Values are:

- spreadforlife
- adjustedsimple
- adjustedtotal

Only single SpreadType values are supported.

If the SpreadType is spreadforlife (default), then the Name-Value arguments LatestFloatingRate and RateInfo are not used. If the SpreadType is adjustedsimple or adjustedtotal, then the Name-Value arguments LatestFloatingRate and RateInfo must be specified.

Default: spreadforlife

'LatestFloatingRate'

NINST-by-1 vector representing the rate for the next floating payment set at the last reset date. This rate must be specified for SpreadType of adjustedsimple and adjustedtotal.

Default:

'RateInfo'

NINST-by-2 vector representing the interest-rate information where the:

- First column is the stub rate between the settlement date and the first coupon rate.
- Second column is the reference rate for the term of the floating coupons (for example, the 3-month LIBOR from settlement date for a bond with `Reset` of 4).

Note: The `RateInfo` must be specified for `SpreadType` of `adjustedsimple` and `adjustedtotal`.

Default:**'Reset'**

NINST-by-1 vector representing the frequency of payments per year.

Default: 1

'Basis'

NINST-by-1 vector representing the basis used for time factor calculations:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see “”.

Default: 0 (actual/actual)

'Principal'

NINST-by-1 of notional principal amounts.

Default: 100

'EndMonthRule'

NINST-by-1 vector representing the end-of-month rule.

Default: 1 (in effect)

'Holidays'

NHOLIDAYS-by-1 vector of MATLAB date numbers. Holidays are used in computing business days.

Default: holidays.m used

'BusinessDayConvention'

NINST-by-1 cell array of business day conventions to be used in computing payment dates. Values are:

- actual
- follow
- modifiedfollow
- previous
- modifiedprevious

Default: actual

Output Arguments

Margin

NINST-by-1 vector of spreads for the floating-rate bond.

AdjPrice

NINST-by-1 vector of the adjusted price used to calculate spreads for SpreadType of adjustedsimple and adjustedtotal.

Examples

Compute Margin Measures for a Floating-Rate Bond

Use floatmargin to compute margin measures for spreadforlife, adjustedsimple, and adjustedtotal for a floating-rate note.

Define data for the floating-rate note.

```
Price = 99.99;
Spread = 50;
Settle = '20-Jan-2011';
Maturity = '15-Jan-2012';
LatestFloatingRate = 0.05;
StubRate = 0.049;
SpotRate = 0.05;
Reset = 4;
Basis = 2;
```

Calculate spreadforlife.

```
Margin = floatmargin(Price, Spread, Settle, Maturity, 'Reset', ...
Reset, 'Basis', Basis)
```

```
Margin =
    51.0051
```

Calculate adjustedsimple margin.

```
[Margin, AdjPrice] = floatmargin(Price, Spread, Settle, Maturity, ...
'SpreadType', 'adjustedsimple', 'RateInfo', [StubRate, SpotRate], ...
'LatestFloatingRate', LatestFloatingRate, 'Reset', Reset, 'Basis', Basis)
```

```
Margin =
```

53.2830

AdjPrice =

99.9673

Calculate adjustedtotal margin.

```
[Margin, AdjPrice] = floatmargin(Price, Spread, Settle, Maturity, ...  
'SpreadType', 'adjustedtotal', 'RateInfo', [StubRate, SpotRate], ...  
'LatestFloatingRate', LatestFloatingRate, 'Reset', Reset, 'Basis', Basis)
```

Margin =

53.4463

AdjPrice =

99.9673

References

Fabozzi, Frank J., Mann, Steven V., *Floating-Rate Securities*, John Wiley and Sons, New York, 2000.

Fabozzi, Frank J., Mann, Steven V., *Introduction to Fixed Income Analytics: Relative Value Analysis, Risk Measures and Valuation*, John Wiley and Sons, New York, 2010.

See Also

| floatdiscmargin | floatbyzero | bndspread

fpctkd

Fast stochastics

Syntax

```
[pctk, pctd] = fpctkd(highp, lowp, closep)
```

```
[pctk, pctd] = fpctkd([highp lowp closep])
```

```
[pctk, pctd] = fpctkd(highp, lowp, closep, kperiods, dperiods,  
    dmamethod)
```

```
[pctk, pctd] = fpctkd([highp lowp closep], kperiods, dperiods,  
    dmamethod)
```

```
pkdts = fpctkd(tsobj, kperiods, dperiods, dmamethod)
```

```
pkdts = fpctkd(tsobj, kperiods, dperiods, dmamethod, ParameterName,  
    ParameterValue, ...)
```

Arguments

highp	High price (vector).
lowp	Low price (vector).
closep	Closing price (vector).
kperiods	(Optional) %K periods. Default = 10.
dperiods	(Optional) %D periods. Default = 3.
damethod	(Optional) %D moving average method. Default = 'e' (exponential).
tsobj	Financial time series object.
ParameterName	Valid parameter names are: <ul style="list-style-type: none"> • HighName: high prices series name • LowName: low prices series name • CloseName: closing prices series name

ParameterValue	Parameter values are the strings that represent the valid parameter names.
----------------	--

Description

fpctkd calculates the stochastic oscillator.

[pctk, pctd] = fpctkd(highp, lowp, closep) calculates the fast stochastics F%K and F%D from the stock price data highp (high prices), lowp (low prices), and closep (closing prices).

[pctk, pctd] = fpctkd([highp lowp closep]) accepts a three-column matrix of high (highp), low (lowp), and closing prices (closep), in that order.

[pctk, pctd] = fpctkd(highp, lowp, closep, kperiods, dperiods, dmamethod) calculates the fast stochastics F%K and F%D from the stock price data highp (high prices), lowp (low prices), and closep (closing prices). kperiods sets the %K period. dperiods sets the %D period.

damethod specifies the %D moving average method. Valid moving average methods for %D are Exponential ('e') and Triangular ('t'). See tsmovavg for explanations of these methods.

[pctk, pctd]= fpctkd([highp lowp closep], kperiods, dperiods, dmamethod) accepts a three-column matrix of high (highp), low (lowp), and closing prices (closep), in that order.

pkdts = fpctkd(tsobj, kperiods, dperiods, dmamethod) calculates the fast stochastics F%K and F%D from the stock price data in the financial time series object tsobj. tsobj must minimally contain the series High (high prices), Low (low prices), and Close (closing prices). pkdts is a financial time series object with similar dates to tsobj and two data series named PercentK and PercentD.

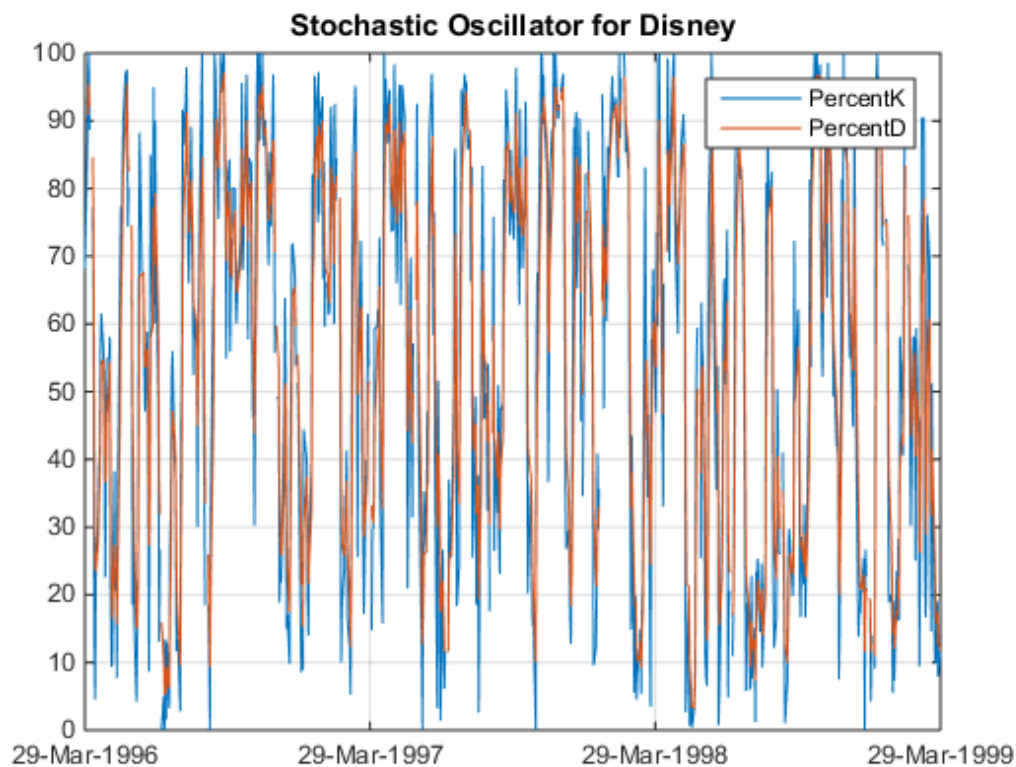
pkdts = fpctkd(tsobj, kperiods, dperiods, dmamethod, ParameterName, ParameterValue, ...) accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Stochastic Oscillator

This example shows how to compute the stochastic oscillator for Disney stock and plot the results.

```
load disney.mat
dis_FastStoc = fpctkd(dis);
plot(dis_FastStoc)
title('Stochastic Oscillator for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second Edition, McGraw-Hill, 1995, pp. 268–271.

See Also

spctkd | stochosc | tsmovavg

frac2cur

Fractional currency value to decimal value

Syntax

```
Decimal = frac2cur(Fraction, Denominator)
```

Description

`Decimal = frac2cur(Fraction, Denominator)` converts a fractional currency value to a decimal value. `Fraction` is the fractional currency value input as a string, and `Denominator` is the denominator of the fraction.

Examples

Convert a Fractional Currency Value to a Decimal Value

This example shows how to convert a fractional currency value to a decimal value.

```
Decimal = frac2cur('12.1', 8)
```

```
Decimal =
```

```
    12.1250
```

See Also

[cur2frac](#) | [cur2str](#)

freqnum

Convert string frequency indicator to numeric frequency indicator

Syntax

```
nfreq = freqnum(sfreq)
```

Arguments

sfreq	UNKNOWN, Unknown, unknown, U, u DAILY, Daily, daily, D, d WEEKLY, Weekly, weekly, W, w MONTHLY, Monthly, monthly, M, m QUARTERLY, Quarterly, quarterly, Q, q SEMIANNUAL, Semiannual, semiannual, S, s ANNUAL, Annual, annual, A, a
-------	--

Description

`nfreq = freqnum(sfreq)` converts a string frequency indicator into a numeric value.

String Frequency Indicator	Numeric Representation
UNKNOWN, Unknown, unknown, U, u	0
DAILY, Daily, daily, D, d	1
WEEKLY, Weekly, weekly, W, w	2
MONTHLY, Monthly, monthly, M, m	3
QUARTERLY, Quarterly, quarterly, Q, q	4

String Frequency Indicator	Numeric Representation
SEMIANNUAL, Semiannual, semiannual, S, s	5
ANNUAL, Annual, annual, A, a	6

See Also

freqstr

freqstr

Convert numeric frequency indicator to string representation

Syntax

```
sfreq = freqstr(nfreq)
```

Arguments

nfreq	0
	1
	2
	3
	4
	5
	6

Description

`sfreq = freqstr(nfreq)` converts a numeric frequency indicator into a string representation.

Numeric Frequency Indicator	String Representation
0	Unknown
1	Daily
2	Weekly
3	Monthly
4	Quarterly

Numeric Frequency Indicator	String Representation
5	Semiannual
6	Annual

See Also

freqnum

frontcon

Mean-variance efficient frontier

As an alternative to `frontcon`, use the `Portfolio` object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using `Portfolio` objects, see “Portfolio Object Workflow”.

Syntax

```
[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn,
ExpCovariance, NumPorts, PortReturn, AssetBounds, Groups,
GroupBounds, varargin)
```

Arguments

ExpReturn	1 by number of assets (NASSETS) vector specifying the expected (mean) return of each asset.
ExpCovariance	NASSETS-by-NASSETS matrix specifying the covariance of asset returns.
NumPorts	(Optional) Number of portfolios generated along the efficient frontier. Returns are equally spaced between the maximum possible return and the minimum risk point. If <code>NumPorts</code> is empty (entered as <code>[]</code>), <code>frontcon</code> computes 10 equally spaced points. When entering a target rate of return (<code>PortReturn</code>), enter <code>NumPorts</code> as an empty matrix <code>[]</code> .
PortReturn	(Optional) Vector of length equal to the number of portfolios (NPORTS) containing the target return values on the frontier. If <code>PortReturn</code> is not entered or <code>[]</code> , <code>NumPorts</code> equally spaced returns between the minimum and maximum possible values are used.
AssetBounds	(Optional) 2-by-NASSETS matrix containing the lower and upper bounds on the weight allocated to each asset in the portfolio. Default lower bound = all 0s (no short-selling).

	Default upper bound = all 1s (any asset may constitute the entire portfolio).
Groups	(Optional) Number of groups (NGROUPS)-by-NASSETS matrix specifying NGROUPS asset groups or classes. Each row specifies a group. $\text{Groups}(i, j) = 1$ (j th asset belongs in the i th group). $\text{Groups}(i, j) = 0$ (j th asset not a member of the i th group).
GroupBounds	(Optional) NGROUPS-by-2 matrix specifying, for each group, the lower and upper bounds of the total weights of all assets in that group. Default lower bound = all 0s. Default upper bound = all 1s.
varargin	<p>(Optional) varargin supports the following parameter-value pairs:</p> <ul style="list-style-type: none"> • 'algorithm' – Defines which algorithm to use with frontcon. Use either a value of 'lcprog' or 'quadprog' to indicate the algorithm to use. The default is 'lcprog'. • 'maxiter' – Maximum number of iterations before termination of algorithm. The default is 100000. • 'tiebreak' – Method to break ties for pivot selection. This value pair applies only to 'lcprog' algorithm. The default is 'first'. Options are: <ul style="list-style-type: none"> • 'first' – Selects pivot with lowest index. • 'last' – Selects pivot with highest index. • 'random' – Selects pivot at random. • 'tolcon' – Tolerance for constraint violations. This value pair applies only to 'lcprog' algorithm. The default is 1.0e-6. • 'tolpiv' – Pivot value below which a number is considered to be zero. This value pair applies only to 'lcprog' algorithm. The default is 1.0e-9.

Description

`[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn, ExpCovariance, NumPorts, PortReturn, AssetBounds, Groups, GroupBounds, varargin)` returns the mean-variance efficient frontier with user-specified asset constraints, covariance, and returns. For a collection of `NASSETS` risky assets, computes a portfolio of asset investment weights that minimize the risk for given values of the expected return. The portfolio risk is minimized subject to constraints on the asset weights or on groups of asset weights.

`PortRisk` is an `NPORTS`-by-1 vector of the standard deviation of each portfolio.

`PortReturn` is a `NPORTS`-by-1 vector of the expected return of each portfolio.

`PortWts` is an `NPORTS`-by-`NASSETS` matrix of weights allocated to each asset. Each row represents a portfolio. The total of all weights in a portfolio is 1.

`frontcon` generates a plot of the efficient frontier if you invoke it without output arguments.

The asset returns are assumed to be jointly normal, with expected mean returns of `ExpReturn` and return covariance `ExpCovariance`. The variance of a portfolio with 1-by-`NASSETS` weights `PortWts` is given by `PortVar = PortWts*ExpCovariance*PortWts'`. The portfolio expected return is `PortReturn = dot(ExpReturn, PortWts)`.

Examples

Compute the Mean-Variance Efficient Frontier

This example shows how to compute the mean-variance efficient frontier for four points, given three assets with expected returns and expected covariance.

```
ExpReturn = [0.1 0.2 0.15];
```

```
ExpCovariance = [ 0.0100  -0.0061   0.0042
                 -0.0061   0.0400  -0.0252
                  0.0042  -0.0252   0.0225];
```

```
NumPorts = 4;
```

```
[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn,...  
ExpCovariance, NumPorts)
```

```
PortRisk =
```

```
0.0426  
0.0483  
0.1089  
0.2000
```

```
PortReturn =
```

```
0.1569  
0.1713  
0.1856  
0.2000
```

```
PortWts =
```

```
0.2134    0.3518    0.4348  
0.0096    0.4352    0.5552  
0         0.7128    0.2872  
0         1.0000    0
```

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

ewstats | frontier | portopt | portstats

frontier

Rolling efficient frontier

Syntax

```
[PortWts, AllMean, AllCovariance] = frontier(Universe, Window,
Offset, NumPorts, ActiveMap, ConSet, NumNonNaN)
```

Arguments

Universe	Number of observations (NUMOBS) by number of assets plus one (NASSETS + 1) time series array containing total return data for a group of securities. Each row represents an observation. Column 1 contains MATLAB serial date numbers. The remaining columns contain the total return data for each security.
Window	Number of data periods used to calculate each frontier.
Offset	Increment in number of periods between each frontier.
NumPorts	Number of portfolios to calculate on each frontier.
ActiveMap	(Optional) Number of observations (NUMOBS) by number of assets (NASSETS) matrix with Boolean elements corresponding to the Universe. Each element indicates if the asset is part of the Universe on the corresponding date. Default = NUMOBS-by-NASSETS matrix of 1's (all assets active on all dates).
Conset	(Optional) Constraint matrix for a portfolio of asset investments, created using portcons with the 'Default' constraint type. This single constraint matrix is applied to each frontier.
NumNonNaN	(Optional) Minimum number of nonNaN points for each active asset in each window of data needed to perform the optimization. The default value is Window - NASSETS.

Description

[PortWts, AllMean, AllCovariance] = frontier(Universe, Window, Offset, NumPorts, ActiveMap, ConSet, NumNonNan) generates a surface of efficient frontiers showing how asset allocation influences risk and return over time.

PortWts is a number of curves (NCURVES)-by-1 cell array, where each element is a NPORTS-by-NASSETS matrix of weights allocated to each asset.

AllMean is a NCURVES-by-1 cell array, where each element is a 1-by-NASSETS vector of the expected asset returns used to generate each curve on the surface.

AllCovariance is a NCURVES-by-1 cell array, where each element is a NASSETS-by-NASSETS vector of the covariance matrix used to generate each curve on the surface.

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

portcons | portopt | portstats

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

fts2ascii

Write elements of time series data into ASCII file

Syntax

```
stat = fts2ascii(filename, tsoj, exttext)
```

```
stat = fts2ascii(filename, dates, data, colheads, desc, exttext)
```

Arguments

<code>filename</code>	Name of an ASCII file
<code>tsoj</code>	Financial time series object
<code>exttext</code>	(Optional) Extra text. A string written after the description line (line 2 in the file).
<code>dates</code>	Column vector containing dates. Dates must be in serial date number format and can specify time of day.
<code>data</code>	Column-oriented matrix. Each column is a series.
<code>colheads</code>	(Optional) Cell array of column headers (names); first cell must always be the one for the dates column. <code>colheads</code> will be written to the file just before the data.
<code>desc</code>	(Optional) Description string, which will be the first line in the file.

Description

`stat = fts2ascii(filename, tsoj, exttext)` writes the financial time series object `tsoj` into an ASCII file `filename`. The data in the file is tab delimited.

`stat = fts2ascii(filename, dates, data, colheads, desc, exttext)` writes into an ASCII file `filename` the dates, times, and data contained in the column vector `dates` and the column-oriented matrix `data`. The first column in `filename` contains the dates, followed by `times` (if specified). Subsequent columns contain the `data`. The data in the file is tab delimited.

stat indicates whether file creation is successful (1) or not (0).

Examples

Use `fts2ascii` to Write a Time Series to an ASCII File

Create a data file with time information.

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
'02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
serial_dates_times = [datenum(dates), datenum(times)];
data = round(10*rand(6,2));
```

Use `fts2ascii` to write the time series to an ascii file.

```
stat = fts2ascii('myfts_file2.txt', serial_dates_times, data, ...
{'dates'; 'times'; 'Data1'; 'Data2'}, 'My FTS with Time')
```

```
stat =
```

```
1
```

Read the data file back and create a financial time series object using `ascii2fts`.

```
MyFts = ascii2fts('myfts_file2.txt', 't', 1, 2, 1)
```

```
MyFts =
```

```
desc: My FTS with Time
freq: Unknown (0)
```

```
'dates: (6)'      'times: (6)'      'Data1: (6)'      'Data2: (6)'
'01-Jan-2001'    '11:00'           [      8]         [      3]
'   "           '12:00'           [      9]         [      5]
'02-Jan-2001'    '11:00'           [      1]         [     10]
'   "           '12:00'           [      9]         [     10]
'03-Jan-2001'    '11:00'           [      6]         [      2]
'   "           '12:00'           [      1]         [     10]
```


See Also

ascii2fts

fts2mat

Convert to matrix

Syntax

```
tsmat = fts2mat(tsoobj)
```

```
tsmat = fts2mat(tsoobj, datesflag)
```

```
tsmat = fts2mat(tsoobj, seriesnames)
```

```
tsmat = fts2mat(tsoobj, datesflag, seriesnames)
```

Arguments

<code>tsoobj</code>	Financial time series object
<code>datesflag</code>	(Optional) Specifies inclusion of dates vector: <code>datesflag = 0</code> (default) excludes dates. <code>datesflag = 1</code> includes dates vector.
<code>seriesnames</code>	(Optional) Specifies the data series to be included in the matrix. Can be a cell array of strings.

Description

`tsmat = fts2mat(tsoobj)` takes the data series in the financial time series object `tsoobj` and puts them into the matrix `tsmat` as columns. The order of the columns is the same as the order of the data series in the object `tsoobj`.

`tsmat = fts2mat(tsoobj, datesflag)` specifies whether or not you want the dates vector included. The dates vector will be the first column. The dates are represented as serial date numbers. Dates can include time-of-day information.

`tsmat = fts2mat(tsoobj, seriesnames)` extracts the data series named in `seriesnames` and puts its values into `tsmat`. The `seriesnames` argument can be a cell array of strings.

`tsmat = fts2mat(tsoobj, datesflag, seriesnames)` puts into `tsmat` the specific data series named in `seriesnames`. The `datesflag` argument must be specified. If `datesflag` is set to 1, the dates vector is included. If you specify an empty matrix (`[]`) for `datesflag`, the default behavior is adopted.

See Also

`subsref`

ftsbound

Start and end dates

Syntax

```
datesbound = ftsbound(tsobj)
```

```
datesbound = ftsbound(tsobj, dateform)
```

Arguments

tsobj	Financial time series object
dateform	dateform is an integer representing the format of a date string. See <code>datestr</code> for a description of these formats.

Description

`ftsbound` returns the start and end dates of a financial time series object. If the object contains time-of-day data, `ftsbound` additionally returns the starting time on the first date and the ending time on the last date.

`datesbound = ftsbound(tsobj)` returns the start and end dates contained in `tsobj` as serial dates in the column matrix `datesbound`. The first row in `datesbound` corresponds to the start date, and the second corresponds to the end date.

`datesbound = ftsbound(tsobj, dateform)` returns the starting and ending dates contained in the object, `tsobj`, as date strings in the column matrix, `datesbound`. The first row in `datesbound` corresponds to the start date, and the second corresponds to the end date. The `dateform` argument controls the format of the output dates.

See Also

`datestr`

ftsgui

Financial time series GUI

Syntax

```
ftsgui
```

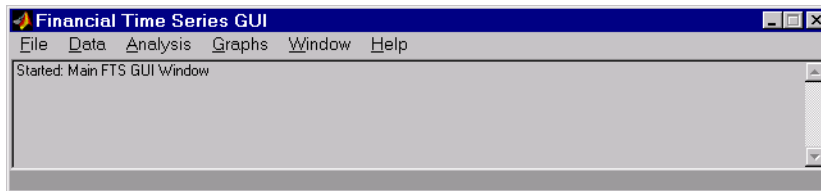
Description

ftsgui displays the financial time series graphical user interface (GUI) main window.

The use of the financial time series GUI is described in “Using the Financial Time Series GUI” on page 14-7.

Examples

```
ftsgui
```



See Also

| ftstool

Tutorials

- “Financial Time Series Graphical User Interface” on page 14-2

ftsinfo

Financial time series object information

Syntax

```
ftsinfo(tsoj)
```

```
infofts = ftsinfo(tsoj)
```

Arguments

<code>tsoj</code>	Financial time series object.
-------------------	-------------------------------

Description

`ftsinfo(tsoj)` displays information about the financial time series object `tsoj`.

`infofts = ftsinfo(tsoj)` stores information about the financial time series object `tsoj` in the structure `infofts`.

`infofts` has these fields.

Field	Contents
<code>version</code>	Financial time series object version.
<code>desc</code>	Description of the time series object (<code>tsoj.desc</code>).
<code>freq</code>	Numeric representation of the time series data frequency (<code>tsoj.freq</code>). See <code>freqstr</code> for list of numeric frequencies and what they represent.
<code>startdate</code>	Earliest date in the time series.
<code>enddate</code>	Latest date in the time series.
<code>seriesnames</code>	Cell array containing the time series data column names.
<code>ndata</code>	Number of data points in the time series.

Field	Contents
nseries	Number of columns of time series data.

Examples

Convert the supplied file `disney.dat` into a financial time series object named `dis`:

```
dis = ascii2fts('disney.dat', 1, 3);
```

Now use `ftsinfo` to obtain information about `dis`:

```
ftsinfo(dis)
```

```
FINTS version: 2.0
Description: Walt Disney Company (DIS)
Frequency: Unknown
Start date: 29-Mar-1996
End date: 29-Mar-1999
Series names: OPEN
              HIGH
              LOW
              CLOSE
              VOLUME
# of data: 782
# of series: 5
```

Then, executing

```
infodis = ftsinfo(dis)
```

creates the structure `infodis` containing the values

```
infodis =
```

```
    ver: '2.0'
   desc: 'Walt Disney Company (DIS)'
   freq: 0
startdate: '29-Mar-1996'
  enddate: '29-Mar-1999'
seriesnames: {5x1 cell}
   ndata: 782
  nseries: 5
```

See Also

fints | freqnum | freqstr | ftsbound

ftstool

Financial Time Series app

Syntax

```
ftstool
```

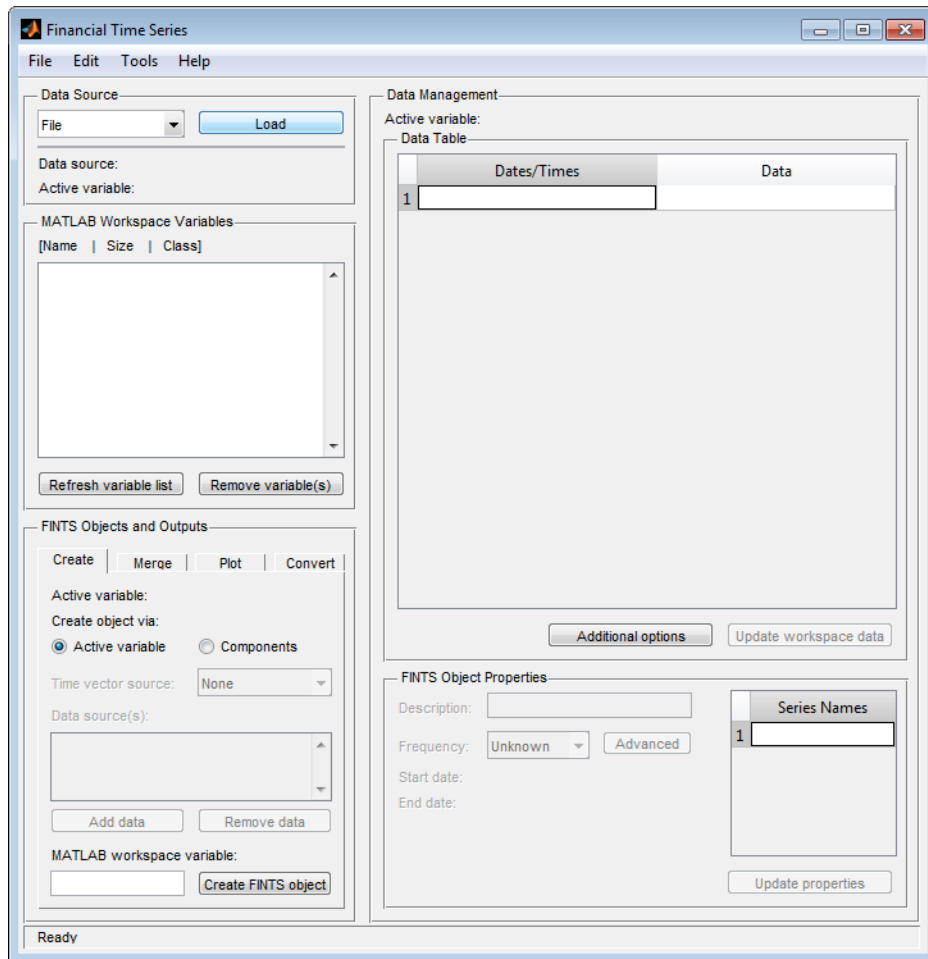
Description

`ftstool` creates and manages Financial Time Series objects. `ftstool` allows the creation and management of Financial Time Series objects via the Financial Time Series app. `ftstool` can interface with `ftsgui`, meaning Line Plots generated with `ftstool` can be analyzed with `ftsgui`. However, `ftsgui` must be running prior to the generation of any Line Plots.

The use of the Financial Time Series app is described in “Getting Started with the Financial Time Series App ” on page 13-3.

Examples

```
ftstool
```



Alternatively, on the MATLAB desktop toolstrip, click the **Apps** tab and in the apps gallery, under **Computational Finance**, click **Financial Time Series**. The Financial Time Series app opens. For an overview of the Financial Time Series app, see “What Is the Financial Time Series App?” on page 13-2.

See Also

| `ftsgui`

Tutorials

- “What Is the Financial Time Series App?” on page 13-2

ftsuniq

Determine uniqueness

Syntax

```
uniq = ftsuniq(dates_and_times)
```

```
[uniq, dup] = ftsuniq(dates_and_times)
```

Arguments

<code>dates_and_times</code>	A single column vector of serial date numbers. The serial date numbers can include time-of-day information.
------------------------------	---

Description

`uniq = ftsuniq(dates_and_times)` returns 1 if the dates and times within the financial time series object are unique and 0 if duplicates exist.

`[uniq, dup] = ftsuniq(dates_and_times)` additionally returns a structure `dup`. In the structure

- `dup.DT` contains the strings of the duplicate dates and times and their locations in the object.
- `dup.intIdx` contains the integer indices of duplicate dates and times in the object.

See Also

`fints`

fvdisc

Future value of discounted security

Syntax

`FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)`

Arguments

Settle	Settlement date. Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Maturity date. Enter as serial date number or date string.
Price	Price (present value) of the security.
Discount	Bank discount rate of the security. Enter as decimal fraction.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252

For more information, see **basis**.

Description

`FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)` finds the amount received at maturity for a fully vested security.

Examples

Find the Amount Received at Maturity for a Fully-Vested Security

This example shows how to find the amount received at maturity for a fully-vested security, using the following data.

```
Settle = '02/15/2001';  
Maturity = '05/15/2001';  
Price = 100;  
Discount = 0.0575;  
Basis = 2;
```

```
FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)
```

```
FutureVal =
```

```
101.4420
```

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition.

See Also

`acrudisc` | `prdisc` | `discrate` | `ylddisc`

fvfix

Future value with fixed periodic payments

Syntax

```
FutureVal = fvfix(Rate, NumPeriods, Payment, PresentVal, Due)
```

Arguments

Rate	Periodic interest rate, as a decimal fraction.
NumPeriods	Number of periods.
Payment	Periodic payment.
PresentVal	(Optional) Initial value. Default = 0.
Due	(Optional) When payments are due or made: 0 = end of period (default), or 1 = beginning of period.

Description

FutureVal = fvfix(Rate, NumPeriods, Payment, PresentVal, Due) returns the future value of a series of equal payments.

Examples

Return the Future Value of a Series of Equal Payments

This example shows how to compute the future value of a series of equal payments using a savings account that has a starting balance of \$1500. \$200 is added at the end of each month for 10 years and the account pays 9% interest compounded monthly.

```
FutureVal = fvfix(0.09/12, 12*10, 200, 1500, 0)
```

```
FutureVal =
```

4.2380e+04

See Also

`fvvar` | `pvfix` | `pvvar`

fvvar

Future value of varying cash flow

Syntax

FutureVal = fvvar(CashFlow, Rate, CFDates)

Arguments

CashFlow	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
Rate	Periodic interest rate. Enter as a decimal fraction.
CFDates	(Optional) For irregular (nonperiodic) cash flows, a vector of dates on which the cash flows occur. Enter dates as serial date numbers or date strings. Default assumes CashFlow contains regular (periodic) cash flows.

Description

FutureVal = fvvar(CashFlow, Rate, CFDates) returns the future value of a varying cash flow.

Examples

This cash flow represents the yearly income from an initial investment of \$10,000. The annual interest rate is 8%.

Year 1	\$2000
Year 2	\$1500
Year 3	\$3000

Year 4	\$3800
Year 5	\$5000

For the future value of this regular (periodic) cash flow

```
FutureVal = fvvar([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
FutureVal =
```

```
2520.47
```

An investment of \$10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

Cash Flow	Dates
(\$10000)	January 12, 2000
\$2500	February 14, 2001
\$2000	March 3, 2001
\$3000	June 14, 2001
\$4000	December 1, 2001

To calculate the future value of this irregular (nonperiodic) cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
```

```
CFDates = ['01/12/2000'  
           '02/14/2001'  
           '03/03/2001'  
           '06/14/2001'  
           '12/01/2001'];
```

```
FutureVal = fvvar(CashFlow, 0.09, CFDates)
```

returns

```
FutureVal =
```

```
170.66
```

See Also

fvfix | irr | payuni | pvfix | pvvar

fwd2zero

Zero curve given forward curve

Syntax

```
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,
Settle, Compounding, Basis)
```

Arguments

ForwardRates	A number of bonds (NUMBONDS)-by-1 vector of annualized implied forward rates, as decimal fractions. In aggregate, the rates in ForwardRates constitute an implied forward curve for the investment horizon represented by CurveDates . The first element pertains to forward rates from the settlement date to the first curve date.	
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the forward rates.	
Settle	A serial date number that is the common settlement date for the forward rates.	
Compounding	(Optional) Output compounding. A scalar that sets the compounding frequency per year for annualizing the output zero rates. Allowed values are:	
	1	Annual compounding
	2	Semiannual compounding (default)
	3	Compounding three times per year
	4	Quarterly compounding
	6	Bimonthly compounding
	12	Monthly compounding
	365	Daily compounding
	- 1	Continuous compounding

Basis	<p>(Optional) Output day-count basis for annualizing the output zero rates.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA) • 11 = 30/360E (ISMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
--------------	---

Description

[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates, Settle, Compounding, Basis) returns a zero curve given an implied forward rate curve and its maturity dates.

ZeroRates	A NUMBONDS-by-1 vector of decimal fractions. In aggregate, the rates in ZeroRates constitute a zero curve for the investment horizon represented by CurveDates .
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates in ZeroRates . This vector is the same as the input vector CurveDates .

Examples

Compute the Zero Curve Given the Forward Curve

This example shows how to compute the zero curve, given an implied forward rate curve over a set of maturity dates, a settlement date, and a compounding rate.

```
ForwardRates = [0.0469
                0.0519
                0.0549
                0.0535
                0.0558
                0.0508
                0.0560
                0.0545
                0.0615
                0.0486];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
Compounding = 1;

[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates, ...
Settle, Compounding)

ZeroRates =

    0.0469
    0.0515
    0.0531
    0.0532
    0.0538
    0.0532
```

```
0.0536  
0.0539  
0.0556  
0.0543
```

```
CurveDates =
```

```
730796  
730831  
730866  
730887  
730914  
730943  
730971  
731027  
731098  
731167
```

This returns the zero curve, `ZeroRates`, at the maturity dates, `CurveDates`.

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

`zero2fwd`

gbm

Create GBM model

Syntax

```
GBM = gbm(Return, Sigma)
```

```
GBM = gbm(Return, Sigma, 'Name1', Value1, 'Name2', Value2, ...)
```

Class

GBM

Description

This function creates and displays geometric Brownian motion (GBM) models, which derive from the CEV (constant elasticity of variance) class. Use GBM models 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 function allows simulation of vector-valued GBM processes of the form:

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$$

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 X_t .
- V is an NVARs-by-NBROWNS instantaneous volatility rate matrix.

- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Return	<p>Return represents the parameter μ. If you specify Return as an array and it must be an NVARS-by-NVARS matrix representing the expected (mean) instantaneous rate of return. As a deterministic function of time, when Return is called with a real-valued scalar time \mathbf{t} as its only input, Return must produce an NVARS-by-NVARS matrix. If you specify Return as a function of time and state, it must return an NVARS-by-NVARS matrix when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V. If you specify Sigma as an array, it must be an NVARS-by-NBROWNS matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of Sigma corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and</p>

	<p>associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARs-by-NBROWNS matrix. If you specify Sigma as a function of time and state, it must return an NVARs-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time <i>t</i>. • An NVARs-by-1 state vector X_t. <p>Although the gbm function enforces no restrictions on the sign of Sigma volatilities, they are usually specified as positive values.</p>
--	--

Optional Input Arguments

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:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, gbm applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, gbm applies a unique initial value to each state variable on all trials.</p>

	<p>If StartState is a matrix, gbm applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>
Simulation	<p>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).</p>

Output Arguments

GBM	<p>Geometric Brownian motion model with the following displayed parameters:</p> <ul style="list-style-type: none"> • 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

Examples

- “Creating Geometric Brownian Motion (GBM) Models” on page 17-26
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., "Transition Densities for Interest Rate and Other Nonlinear Diffusions," *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | diffusion | cev

geom2arith

Geometric to arithmetic moments of asset returns

Syntax

```
[ma, Ca] = geom2arith(mg, Cg)
```

```
[ma, Ca] = geom2arith(mg, Cg, t)
```

Arguments

mg	Continuously-compounded or “geometric” mean of asset returns (positive n-vector).
Cg	Continuously-compounded or “geometric” covariance of asset returns (n-by-n symmetric, positive-semidefinite matrix).
t	(Optional) Target period of arithmetic moments in terms of periodicity of geometric moments with default value 1 (positive scalar).

Description

`geom2arith` transforms moments associated with a continuously-compounded geometric Brownian motion into equivalent moments associated with a simple Brownian motion with a possible change in periodicity.

`[ma, Ca] = geom2arith(mg, Cg, t)` returns `ma`, arithmetic mean of asset returns over the target period (n-vector), and `Ca`, which is an arithmetic covariance of asset returns over the target period (n-by-n matrix).

Geometric returns over period t_G are modeled as multivariate lognormal random variables with moments

$$E[Y] = 1 + m_G$$

and

$$\text{cov}(Y) = C_G$$

Arithmetic returns over period t_A are modeled as multivariate normal random variables with moments

$$E[X] = m_A$$

$$\text{cov}(X) = C_A$$

Given $t = t_A / t_G$, the transformation from geometric to arithmetic moments is

$$C_{A_{ij}} = t \log \left(1 + \frac{C_{G_{ij}}}{(1 + m_{G_i})(1 + m_{G_j})} \right)$$

$$m_{A_i} = t \log(1 + m_{G_i}) - \frac{1}{2} C_{A_{ii}}$$

For $i, j = 1, \dots, n$.

Note: If $t = 1$, then $\mathbf{X} = \log(\mathbf{Y})$.

This function requires that the input mean must satisfy $1 + m_G > 0$ and that the input covariance C_G must be a symmetric, positive, semidefinite matrix.

The functions `geom2arith` and `arith2geom` are complementary so that, given m , C , and t , the sequence

$$\begin{aligned} [m_a, C_a] &= \text{geom2arith}(m, C, t); \\ [m_g, C_g] &= \text{arith2geom}(m_a, C_a, 1/t); \end{aligned}$$

yields $m_g = m$ and $C_g = C$.

Examples

Example 1. Given geometric mean m and covariance C of monthly total returns, obtain annual arithmetic mean ma and covariance Ca . In this case, the output period (1 year) is 12 times the input period (1 month) so that $t = 12$ with

```
[ma, Ca] = geom2arith(m, C, 12);
```

Example 2. Given annual geometric mean m and covariance C of asset returns, obtain monthly arithmetic mean ma and covariance Ca . In this case, the output period (1 month) is 1/12 times the input period (1 year) so that $t = 1/12$ with

```
[ma, Ca] = geom2arith(m, C, 1/12);
```

Example 3. Given geometric means m and standard deviations s of daily total returns (derived from 260 business days per year), obtain annualized arithmetic mean ma and standard deviations sa with

```
[ma, Ca] = geom2arith(m, diag(s.^2), 260);  
sa = sqrt(diag(Ca));
```

Example 4. Given geometric mean m and covariance C of monthly total returns, obtain quarterly arithmetic return moments. In this case, the output is 3 of the input periods so that $t = 3$ with

```
[ma, Ca] = geom2arith(m, C, 3);
```

Example 5. Given geometric mean m and covariance C of 1254 observations of daily total returns over a 5-year period, obtain annualized arithmetic return moments. Since the periodicity of the geometric data is based on 1254 observations for a 5-year period, a 1-year period for arithmetic returns implies a target period of $t = 1254/5$ so that

```
[ma, Ca] = geom2arith(m, C, 1254/5);
```

See Also

arith2geom

getAssetMoments

Class: Portfolio

Obtain mean and covariance of asset returns from Portfolio object

Syntax

```
[AssetMean,AssetCovar] = getAssetMoments(obj)
```

Description

[AssetMean,AssetCovar] = getAssetMoments(obj) obtains the mean and covariance of asset returns from a Portfolio object.

Tips

You can also use dot notation to obtain the mean and covariance of asset returns from a Portfolio object:

```
[AssetMean, AssetCovar] = obj.getAssetMoments;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

AssetMean

Mean of asset returns [vector].

AssetCovar

Covariance of asset returns [matrix].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Asset Moment Properties

Given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set and then obtained using the `getAssetMoments` method:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = setAssetMoments(p, m, C);
[assetmean, assetcovar] = getAssetMoments(p)
```

```
assetmean =
    0.0042
    0.0083
    0.0100
    0.0150
```

```
assetcovar =
```

```
0.0005    0.0003    0.0002         0
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

- “Asset Returns and Moments of Asset Returns” on page 4-41
- “Estimating Asset Moments from Prices or Returns” on page 4-45
- “Estimating Asset Moments with Missing Data” on page 4-48

See Also

setAssetMoments

getBounds

Class: Portfolio

Obtain bounds for portfolio weights from Portfolio object

Syntax

```
[LowerBound,UpperBound] = getBounds(obj)
```

Description

[LowerBound,UpperBound] = getBounds(obj) obtains bounds for portfolio weights from a Portfolio object.

Tips

You can also use dot notation to obtain bounds for portfolio weights from the Portfolio object.

```
[LowerBound, UpperBound] = obj.getBounds;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

LowerBound

Lower-bound weight for each asset [vector].

UpperBound

Upper-bound weight for each asset [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Values for Lower and Upper Bounds

Given portfolio `p` with the default constraints set, obtain the values for `LowerBound` and `UpperBound`.

```
p = Portfolio;  
p = setDefaultConstraints(p, 5);  
[LowerBound, UpperBound] = getBounds(p)
```

```
LowerBound =
```

```
0  
0  
0  
0  
0
```

```
UpperBound =
```

```
[]
```

- “Working with Bound Constraints” on page 4-60

See Also

setBounds

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getBounds

Class: PortfolioCVaR

Obtain bounds for portfolio weights from PortfolioCVaR object

Syntax

```
[LowerBound,UpperBound] = getBounds(obj)
```

Description

[LowerBound,UpperBound] = getBounds(obj) obtains bounds for portfolio weights from a PortfolioCVaR object.

Tips

You can also use dot notation to obtain bounds for portfolio weights from the PortfolioCVaR object.

```
[LowerBound, UpperBound] = obj.getBounds;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

LowerBound

Lower-bound weight for each asset [vector].

UpperBound

Upper-bound weight for each asset [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Values for Lower and Upper Bounds

Given a PortfolioCVaR object `p` with the default constraints set, obtain the values for LowerBound and UpperBound.

```
p = PortfolioCVaR;  
p = setDefaultConstraints(p, 5);  
[LowerBound, UpperBound] = getBounds(p)
```

```
LowerBound =
```

```
0  
0  
0  
0  
0
```

```
UpperBound =
```

```
[]
```

- “Working with Bound Constraints” on page 5-59

See Also

setBounds

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getBounds

Class: PortfolioMAD

Obtain bounds for portfolio weights from PortfolioMAD object

Syntax

```
[LowerBound,UpperBound] = getBounds(obj)
```

Description

[LowerBound,UpperBound] = `getBounds(obj)` obtains bounds for PortfolioMAD portfolio weights from a PortfolioMAD object.

Tips

You can also use dot notation to obtain bounds for PortfolioMAD portfolio weights from the PortfolioMAD object.

```
[LowerBound, UpperBound] = obj.getBounds;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

LowerBound

Lower-bound weight for each asset [vector].

UpperBound

Upper-bound weight for each asset [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Values for Lower and Upper Bounds

Given a PortfolioMAD object `p` with the default constraints set, obtain the values for `LowerBound` and `UpperBound`.

```
p = PortfolioMAD;  
p = setDefaultConstraints(p, 5);  
[LowerBound, UpperBound] = getBounds(p)
```

```
LowerBound =
```

```
0  
0  
0  
0  
0
```

```
UpperBound =
```

```
[]
```

- “Working with Bound Constraints” on page 6-59

See Also

setBounds

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getBudget

Class: Portfolio

Obtain budget constraint bounds from Portfolio object

Syntax

```
[LowerBudget,UpperBudget] = getBudget(obj)
```

Description

[LowerBudget,UpperBudget] = getBudget(obj) obtains budget constraint bounds from a Portfolio object.

Tips

You can also use dot notation to obtain the budget constraint bounds from the Portfolio object.

```
[LowerBudget, UpperBudget] = obj.getBudget;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

LowerBudget

Lower bound for budget constraint [scalar].

UpperBudget

Upper bound for budget constraint [scalar].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Values for Lower and Upper Budgets

Given portfolio `p` with the default constraints set, obtain the values for `LowerBudget` and `UpperBudget`.

```
p = Portfolio;  
p = setDefaultConstraints(p, 5);  
[LowerBudget, UpperBudget] = getBudget(p)
```

```
LowerBudget =
```

```
1
```

```
UpperBudget =
```

```
1
```

- “Working with Budget Constraints” on page 4-62

See Also

`setBudget`

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getBudget

Class: PortfolioCVaR

Obtain budget constraint bounds from PortfolioCVaR object

Syntax

```
[LowerBudget,UpperBudget] = getBudget(obj)
```

Description

[LowerBudget,UpperBudget] = getBudget(obj) obtains budget constraint bounds from a PortfolioCVaR object.

Tips

You can also use dot notation to obtain the budget constraint bounds from the PortfolioCVaR object.

```
[LowerBudget, UpperBudget] = obj.getBudget;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

LowerBudget

Lower-bound for budget constraint [scalar].

UpperBudget

Upper-bound for budget constraint [scalar].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Values for Lower and Upper Budgets

Given a PortfolioCVaR object `p` with the default constraints set, obtain the values for `LowerBudget` and `UpperBudget`.

```
p = PortfolioCVaR;  
p = setDefaultConstraints(p, 5);  
[LowerBudget, UpperBudget] = getBudget(p)
```

```
LowerBudget =
```

```
1
```

```
UpperBudget =
```

```
1
```

- “Working with Budget Constraints” on page 5-61

See Also

`setBudget`

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getBudget

Class: PortfolioMAD

Obtain budget constraint bounds from PortfolioMAD object

Syntax

```
[LowerBudget,UpperBudget] = getBudget(obj)
```

Description

[LowerBudget,UpperBudget] = getBudget(obj) obtains budget constraint bounds from a PortfolioMAD object.

Tips

You can also use dot notation to obtain the budget constraint bounds from the PortfolioMAD object.

```
[LowerBudget, UpperBudget] = obj.getBudget;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

LowerBudget

Lower-bound for budget constraint [scalar].

UpperBudget

Upper-bound for budget constraint [scalar].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Values for Lower and Upper Budgets

Given a PortfolioMAD object `p` with the default constraints set, obtain the values for LowerBudget and UpperBudget.

```
p = PortfolioMAD;  
p = setDefaultConstraints(p, 5);  
[LowerBudget, UpperBudget] = getBudget(p)
```

```
LowerBudget =
```

```
1
```

```
UpperBudget =
```

```
1
```

- “Working with Budget Constraints” on page 6-61

See Also

setBudget

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getCosts

Class: Portfolio

Obtain buy and sell transaction costs from Portfolio object

Syntax

```
[BuyCost, SellCost] = getCosts(obj)
```

Description

[BuyCost, SellCost] = getCosts(obj) obtains buy and sell transaction costs from the Portfolio object.

Tips

You can also use dot notation to obtain the buy and sell transaction costs from the Portfolio object.

```
[BuyCost, SellCost] = obj.getCosts;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

BuyCost

Proportional transaction cost to purchase each asset [vector].

SellCost

Proportional transaction cost to sell each asset [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Buy and Sell Costs

Given portfolio `p` with the costs set, obtain the values for `BuyCost` and `SellCost`.

```
p = Portfolio;  
p = setCosts(p, 0.001, 0.001, 5);  
[BuyCost, SellCost] = getCosts(p)
```

```
BuyCost =  
  
    1.0000e-03
```

```
SellCost =  
  
    1.0000e-03
```

- “Working with Transaction Costs” on page 4-52

See Also

`setCosts`

getCosts

Class: PortfolioCVaR

Obtain buy and sell transaction costs from PortfolioCVaR object

Syntax

```
[BuyCost, SellCost] = getCosts(obj)
```

Description

[BuyCost, SellCost] = `getCosts(obj)` obtains buy and sell transaction costs from the PortfolioCVaR object.

Tips

You can also use dot notation to obtain the buy and sell transaction costs from the PortfolioCVaR object.

```
[BuyCost, SellCost] = obj.getCosts;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

BuyCost

Proportional transaction cost to purchase each asset [vector].

SellCost

Proportional transaction cost to sell each asset [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Buy and Sell Costs

Given a PortfolioCVaR object `p` with the costs set, obtain the values for `BuyCost` and `SellCost`.

```
p = PortfolioCVaR;  
p = setCosts(p, 0.001, 0.001, 5);  
[BuyCost, SellCost] = getCosts(p)
```

```
BuyCost =  
  
    1.0000e-03
```

```
SellCost =  
  
    1.0000e-03
```

- “Working with Transaction Costs” on page 5-51

See Also

`setCosts`

getCosts

Class: PortfolioMAD

Obtain buy and sell transaction costs from PortfolioMAD object

Syntax

```
[BuyCost, SellCost] = getCosts(obj)
```

Description

[BuyCost, SellCost] = `getCosts(obj)` obtains buy and sell transaction costs from the PortfolioMAD object.

Tips

You can also use dot notation to obtain the buy and sell transaction costs from the PortfolioMAD object.

```
[BuyCost, SellCost] = obj.getCosts;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

BuyCost

Proportional transaction cost to purchase each asset [vector].

SellCost

Proportional transaction cost to sell each asset [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Buy and Sell Costs

Given a PortfolioMAD object `p` with the costs set, obtain the values for `BuyCost` and `SellCost`.

```
p = PortfolioMAD;  
p = setCosts(p, 0.001, 0.001, 5);  
[BuyCost, SellCost] = getCosts(p)
```

```
BuyCost =  
  
    1.0000e-03
```

```
SellCost =  
  
    1.0000e-03
```

- “Working with Transaction Costs” on page 6-51

See Also

`setCosts`

getEquality

Class: Portfolio

Obtain equality constraint arrays from Portfolio object

Syntax

```
[AEquality,bEquality] = getEquality(obj)
```

Description

[AEquality,bEquality] = `getEquality(obj)` obtains the equality constraint arrays from a Portfolio object.

Tips

You can also use dot notation to obtain the equality constraint arrays from the Portfolio object.

```
[AEquality, bEquality] = obj.getEquality;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Equality Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are exactly 50% of your portfolio. Given a Portfolio object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];  
b = 0.5;  
p = Portfolio;  
p = setEquality(p, A, b);  
[AEquality, bEquality] = getEquality(p)
```

```
AEquality =
```

```
    1    1    1    0    0
```

```
bEquality =
```

```
    0.5000
```

- “Working with Linear Equality Constraints” on page 4-69

See Also

setEquality

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getEquality

Class: PortfolioCVaR

Obtain equality constraint arrays from PortfolioCVaR object

Syntax

```
[AEquality,bEquality] = getEquality(obj)
```

Description

[AEquality,bEquality] = `getEquality(obj)` obtains the equality constraint arrays from a PortfolioCVaR object.

Tips

You can also use dot notation to obtain the equality constraint arrays from a PortfolioCVaR object.

```
[AEquality, bEquality] = obj.getEquality;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Equality Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioCVaR object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];  
b = 0.5;  
p = PortfolioCVaR;  
p = setEquality(p, A, b);  
[AEquality, bEquality] = getEquality(p)
```

```
AEquality =
```

```
    1    1    1    0    0
```

```
bEquality =
```

```
    0.5000
```

- “Working with Linear Equality Constraints” on page 5-68

See Also

setEquality

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getEquality

Class: PortfolioMAD

Obtain equality constraint arrays from PortfolioMAD object

Syntax

```
[AEquality,bEquality] = getEquality(obj)
```

Description

[AEquality,bEquality] = `getEquality(obj)` obtains the equality constraint arrays from a PortfolioMAD object.

Tips

You can also use dot notation to obtain the equality constraint arrays from a PortfolioMAD object.

```
[AEquality, bEquality] = obj.getEquality;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Equality Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioMAD object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setEquality(p, A, b);
[AEquality, bEquality] = getEquality(p)
```

```
AEquality =
```

```
    1    1    1    0    0
```

```
bEquality =
```

```
    0.5000
```

- “Working with Linear Equality Constraints” on page 6-68

See Also

setEquality

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getGroupRatio

Class: Portfolio

Obtain group ratio constraint arrays from Portfolio object

Syntax

```
[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj)
```

Description

[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj) obtains the group ratio constraint arrays from a Portfolio object.

Tips

You can also use dot notation to obtain the equality constraint arrays from the Portfolio object.

```
[GroupA, GroupB, LowerRatio, UpperRatio] = obj.getGroupRatio;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

LowerRatio

Lower bound for ratio of GroupB groups to GroupA groups [vector].

UpperRatio

Upper bound for ratio of GroupB groups to GroupA groups [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Group Ratio Constraints

Suppose you want to make sure that the ratio of financial to nonfinancial companies in your portfolios never goes above 50%. Assume you have 6 assets with 3 financial companies (assets 1-3) and 3 nonfinancial companies (assets 4-6). After setting group ratio constraints, obtain the values for GroupA, GroupB, LowerRatio, and UpperRatio.

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);
[GroupA, GroupB, LowerRatio, UpperRatio] = getGroupRatio(p)
```

```
GroupA =
```

```
1 1 1 0 0 0
```

```
GroupB =
```

```
0 0 0 1 1 1
```

```
LowerRatio =
```

```
[]
```

```
UpperRatio =
```

```
0.5000
```

- “Working with Group Ratio Constraints” on page 4-66

See Also

setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getGroupRatio

Class: PortfolioCVaR

Obtain group ratio constraint arrays from PortfolioCVaR object

Syntax

```
[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj)
```

Description

[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj) obtains the group ratio constraint arrays from a CVaR portfolio object.

Tips

You can also use dot notation to obtain group ratio constraint arrays from the CVaR portfolio object.

```
[GroupA, GroupB, LowerRatio, UpperRatio] = obj.getGroupRatio;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

LowerRatio

Lower bound for ratio of GroupB groups to GroupA groups [vector].

UpperRatio

Upper bound for ratio of GroupB groups to GroupA groups [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Group Ratio Constraints**

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolios never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). After setting group ratio constraints, obtain the values for GroupA, GroupB, LowerRatio, and UpperRatio.

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = PortfolioCVar;
p = setGroupRatio(p, GA, GB, [], 0.5);
[GroupA, GroupB, LowerRatio, UpperRatio] = getGroupRatio(p)
```

GroupA =

```
1 1 1 0 0 0
```

```
GroupB =
```

```
0 0 0 1 1 1
```

```
LowerRatio =
```

```
[]
```

```
UpperRatio =
```

```
0.5000
```

- “Working with Group Constraints” on page 5-63

See Also

setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getGroupRatio

Class: PortfolioMAD

Obtain group ratio constraint arrays from PortfolioMAD object

Syntax

```
[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj)
```

Description

[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj) obtains the group ratio constraint arrays from a MAD portfolio object.

Tips

You can also use dot notation to obtain group ratio constraint arrays from the MAD portfolio object.

```
[GroupA, GroupB, LowerRatio, UpperRatio] = obj.getGroupRatio;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

LowerRatio

Lower bound for ratio of GroupB groups to GroupA groups [vector].

UpperRatio

Upper bound for ratio of GroupB groups to GroupA groups [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Group Ratio Constraints

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolios never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). After setting group ratio constraints, obtain the values for GroupA, GroupB, LowerRatio, and UpperRatio.

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);
[GroupA, GroupB, LowerRatio, UpperRatio] = getGroupRatio(p)
```

```
GroupA =
```

```
1 1 1 0 0 0
```

```
GroupB =
```

```
0 0 0 1 1 1
```

```
LowerRatio =
```

```
[]
```

```
UpperRatio =
```

```
0.5000
```

- “Working with Group Constraints” on page 6-62

See Also

setGroupRatio

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getGroups

Class: Portfolio

Obtain group constraint arrays from Portfolio object

Syntax

```
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(obj)
```

Description

[GroupMatrix, LowerGroup, UpperGroup] = getGroups(obj) obtains the group constraint arrays from a Portfolio object.

Tips

You can also use dot notation to obtain the group constraint arrays from the Portfolio object.

```
[GroupMatrix, LowerGroup, UpperGroup] = obj.getGroups;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

GroupMatrix

Group constraint matrix [matrix].

LowerGroup

Lower bound for group constraints [vector].

UpperGroup

Upper bound for group constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Group Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a Portfolio object `p` with the group constraints set, obtain the values for `GroupMatrix`, `LowerGroup`, and `UpperGroup`.

```
G = [ true true true false false ];
p = Portfolio;
p = setGroups(p, G, [], 0.3);
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(p)
```

```
GroupMatrix =
```

```
    1    1    1    0    0
```

```
LowerGroup =
```

[]

UpperGroup =

0.3000

- “Working with Group Constraints” on page 4-63

See Also

setGroups

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getGroups

Class: PortfolioCVaR

Obtain group constraint arrays from PortfolioCVaR object

Syntax

```
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(obj)
```

Description

[GroupMatrix, LowerGroup, UpperGroup] = getGroups(obj) obtains the group constraint arrays from a PortfolioCVaR object.

Tips

You can also use dot notation to obtain the group constraint arrays from a PortfolioCVaR object.

```
[GroupMatrix, LowerGroup, UpperGroup] = obj.getGroups;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

GroupMatrix

Group constraint matrix [matrix].

LowerGroup

Lower bound for group constraints [vector].

UpperGroup

Upper bound for group constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Group Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a `PortfolioCVaR` object `p` with the group constraints set, obtain the values for `GroupMatrix`, `LowerGroup`, and `UpperGroup`.

```
G = [ true true true false false ];  
p = PortfolioCVaR;  
p = setGroups(p, G, [], 0.3);  
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(p)
```

```
GroupMatrix =
```

```
    1    1    1    0    0
```

```
LowerGroup =
```

[]

UpperGroup =

0.3000

- “Working with Group Constraints” on page 5-63

See Also

setGroups

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getGroups

Class: PortfolioMAD

Obtain group constraint arrays from PortfolioMAD object

Syntax

```
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(obj)
```

Description

[GroupMatrix, LowerGroup, UpperGroup] = getGroups(obj) obtains the group constraint arrays from a PortfolioMAD object.

Tips

You can also use dot notation to obtain the group constraint arrays from a PortfolioMAD object.

```
[GroupMatrix, LowerGroup, UpperGroup] = obj.getGroups;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

GroupMatrix

Group constraint matrix [matrix].

LowerGroup

Lower bound for group constraints [vector].

UpperGroup

Upper bound for group constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Group Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a PortfolioMAD object `p` with the group constraints set, obtain the values for `GroupMatrix`, `LowerGroup`, and `UpperGroup`.

```
G = [ true true true false false ];
p = PortfolioMAD;
p = setGroups(p, G, [], 0.3);
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(p)
```

```
GroupMatrix =
```

```
    1    1    1    0    0
```

```
LowerGroup =
```

[]

UpperGroup =

0.3000

- “Working with Group Constraints” on page 6-62

See Also

setGroups

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getScenarios

Class: PortfolioCVaR

Obtain scenarios from PortfolioCVaR object

Syntax

```
Y = getScenarios(obj)
```

Description

Y = getScenarios(obj) obtains scenarios from a PortfolioCVaR object.

Tips

You can also use dot notation to obtain scenarios from a PortfolioCVaR object.

```
Y = obj.getScenarios;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

Y

Scenarios matrix [NumScenarios]-by-[NumAssets].

Attributes

Access

public

Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Scenarios for a CVaR Portfolio Object

For a given PortfolioCVaR object `p`, display the defined scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

rng(11);

rng(11);

AssetScenarios = mvnrnd(m, C, 10);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

Y = getScenarios(p)

Y =

    -0.0056    0.0440    0.1186    0.0488
   -0.0368   -0.0753    0.0087    0.1124
    0.0025    0.0856    0.0484    0.1404
    0.0318    0.0826    0.0377    0.0404
    0.0013   -0.0561   -0.1466   -0.0621
```


0.0035	0.0310	-0.0183	0.1225
-0.0519	-0.1634	-0.0526	0.1528
0.0029	-0.1163	-0.0627	-0.0760
0.0192	-0.0182	-0.1243	-0.1346
0.0440	0.0189	0.0098	0.0821

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Setting Scenarios Using the PortfolioCVaR Constructor” on page 5-42

See Also

`rng` | `setScenarios`

More About

- “How Stochastic Optimization Works” on page 5-41

getScenarios

Class: PortfolioMAD

Obtain scenarios from PortfolioMAD object

Syntax

```
Y = getScenarios(obj)
```

Description

Y = getScenarios(obj) obtains scenarios from a PortfolioMAD object.

Tips

You can also use dot notation to obtain scenarios from a PortfolioMAD object.

```
Y = obj.getScenarios;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

Y

Scenarios matrix [NumScenarios]-by-[NumAssets].

Attributes

Access

public

```

Static                false
Hidden                false

```

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain Scenarios for a MAD Portfolio Object

For a given PortfolioMAD object `p`, display the defined scenarios.

```

m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 10);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

Y = getScenarios(p)

Y =

-0.0056    0.0440    0.1186    0.0488
-0.0368   -0.0753    0.0087    0.1124
 0.0025    0.0856    0.0484    0.1404
 0.0318    0.0826    0.0377    0.0404
 0.0013   -0.0561   -0.1466   -0.0621
 0.0035    0.0310   -0.0183    0.1225
-0.0519   -0.1634   -0.0526    0.1528
 0.0029   -0.1163   -0.0627   -0.0760
 0.0192   -0.0182   -0.1243   -0.1346

```

0.0440 0.0189 0.0098 0.0821

The function `rng(seed)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Setting Scenarios Using the PortfolioMAD Constructor” on page 6-42
- “Setting Scenarios Using the setScenarios Method” on page 6-44

See Also

`rng` | `setScenarios`

More About

- “How Stochastic Optimization Works” on page 6-41

getfield

Content of specific field

Syntax

```
fieldval = getfield(tsoobj, field)
```

```
fieldval = getfield(tsoobj, field, {dates})
```

Arguments

<code>tsoobj</code>	Financial time series object.
<code>field</code>	Field name within <code>tsoobj</code> .
<code>dates</code>	Date range. Dates can be expanded to include time-of-day information.

Description

`getfield` treats the contents of a financial times series object `tsoobj` as fields in a structure.

`fieldval = getfield(tsoobj, field)` returns the contents of the specified field. This is equivalent to the syntax `fieldval = tsoobj field`.

`fieldval = getfield(tsoobj, field, {dates})` returns the contents of the specified field for the specified dates. `dates` can be individual cells of date strings or a cell of a date string range using the `::` operator, such as `'03/01/99::03/31/99'`.

Examples

Create a financial time series object containing both date and time-of-day information:

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
         '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
```

```
times = ['11:00';'12:00';'11:00';'12:00';'11:00';'12:00'];
dates_times = cellstr([dates, repmat(' ',size(dates,1),1),...
times]);
AnFts = fints(dates_times,[(1:4)'; nan; 6],{'Data1'},1,...
'Yet Another Financial Time Series')
```

```
AnFts =
```

```
desc: Yet Another Financial Time Series
freq: Daily (1)

'dates: (6)'      'times: (6)'      'Data1: (6)'
'01-Jan-2001'    '11:00'           [          1]
'   "   "   "   '12:00'           [          2]
'02-Jan-2001'    '11:00'           [          3]
'   "   "   "   '12:00'           [          4]
'03-Jan-2001'    '11:00'           [         NaN]
'   "   "   "   '12:00'           [          6]
```

Example 1. Get the contents of the `times` field in `AnFts`:

```
F = datestr(getfield(AnFts, 'times'))
```

```
F =
```

```
11:00 AM
12:00 PM
11:00 AM
12:00 PM
11:00 AM
12:00 PM
```

Example 2. Extract the contents of specific data fields within `AnFts`:

```
FF = getfield(AnFts,'Data1',...
'01-Jan-2001 12:00::02-Jan-2001 12:00')
```

```
FF =
```

```
2
3
4
```

See Also

`chfield` | `fieldnames` | `isfield` | `rmfield` | `setfield`

getInequality

Class: Portfolio

Obtain inequality constraint arrays from Portfolio object

Syntax

```
[AInequality,bInequality] = getInequality(obj)
```

Description

[AInequality,bInequality] = `getInequality(obj)` obtains the inequality constraint arrays from a Portfolio object.

Tips

You can also use dot notation to obtain the inequality constraint arrays from the Portfolio object.

```
[AInequality, bInequality] = obj.getInequality;
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Inequality Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given a Portfolio object `p`, set the linear inequality constraints and then obtain values for `AInequality` and `bInequality`.

```
A = [ 1 1 1 0 0 ];  
b = 0.5;  
p = Portfolio;  
p = setInequality(p, A, b);  
[AInequality, bInequality] = getInequality(p)
```

```
AInequality =
```

```
    1    1    1    0    0
```

```
bInequality =
```

```
    0.5000
```

- “Working with Linear Inequality Constraints” on page 4-71

See Also

setInequality

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getInequality

Class: PortfolioCVaR

Obtain inequality constraint arrays from PortfolioCVaR object

Syntax

```
[AInequality,bInequality] = getInequality(obj)
```

Description

[AInequality,bInequality] = `getInequality(obj)` obtains the inequality constraint arrays from a PortfolioCVaR object.

Tips

You can also use dot notation to obtain the inequality constraint arrays from a PortfolioCVaR object.

```
[AInequality, bInequality] = obj.getInequality;
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Inequality Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 50% of your portfolio. Given a PortfolioCVaR object `p`, set the linear inequality constraints and then obtain values for `AInequality` and `bInequality`.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setInequality(p, A, b);
[AInequality, bInequality] = getInequality(p)
```

```
AInequality =
```

```
    1    1    1    0    0
```

```
bInequality =
```

```
    0.5000
```

- “Working with Linear Inequality Constraints” on page 5-70

See Also

setInequality

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getInequality

Class: PortfolioMAD

Obtain inequality constraint arrays from PortfolioMAD object

Syntax

```
[AInequality,bInequality] = getInequality(obj)
```

Description

[AInequality,bInequality] = `getInequality(obj)` obtains the inequality constraint arrays from a PortfolioMAD object.

Tips

You can also use dot notation to obtain the inequality constraint arrays from a PortfolioMAD object.

```
[AInequality, bInequality] = obj.getInequality;
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples**Obtain Inequality Constraints**

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 50% of your portfolio. Given a PortfolioMAD object `p`, set the linear inequality constraints and then obtain values for `AInequality` and `bInequality`.

```
A = [ 1 1 1 0 0 ];  
b = 0.5;  
p = PortfolioMAD;  
p = setInequality(p, A, b);  
[AInequality, bInequality] = getInequality(p)
```

```
AInequality =
```

```
    1    1    1    0    0
```

```
bInequality =
```

```
    0.5000
```

- “Working with Linear Inequality Constraints” on page 6-70

See Also

setInequality

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getOneWayTurnover

Class: Portfolio

Obtain one-way turnover constraints from Portfolio object

Syntax

```
[BuyTurnover, SellTurnover] = getOneWayTurnover(obj)
```

Description

[BuyTurnover, SellTurnover] = getOneWayTurnover(obj) obtains one-way turnover constraints from the Portfolio object.

Tips

You can also use dot notation to get the one-way turnover constraint for the Portfolio object.

```
[BuyTurnover, SellTurnover] = obj.getOneWayTurnover
```

Input Arguments

obj

Portfolio object [Portfolio].

Output Arguments

BuyTurnover

Turnover constraint on purchases [scalar].

SellTurnover

Turnover constraint on sales [scalar].

Definitions

One-Way Turnover Constraint

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the form

$$1^T \max\{0, x - x_0\} \leq \tau_B$$

$$1^T \max\{0, x_0 - x\} \leq \tau_S$$

with

- x — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper-bound for turnover constraint on purchases (scalar)
- τ_S — Upper-bound for turnover constraint on sales (scalar)

Specify one-way turnover constraints using the following properties in the Portfolio object: `BuyTurnover` for τ_B , `SellTurnover` for τ_S , and `InitPort` for x_0 .

Note: The average turnover constraint (which is set using `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Attributes

Access	public
Static	false
Hidden	false

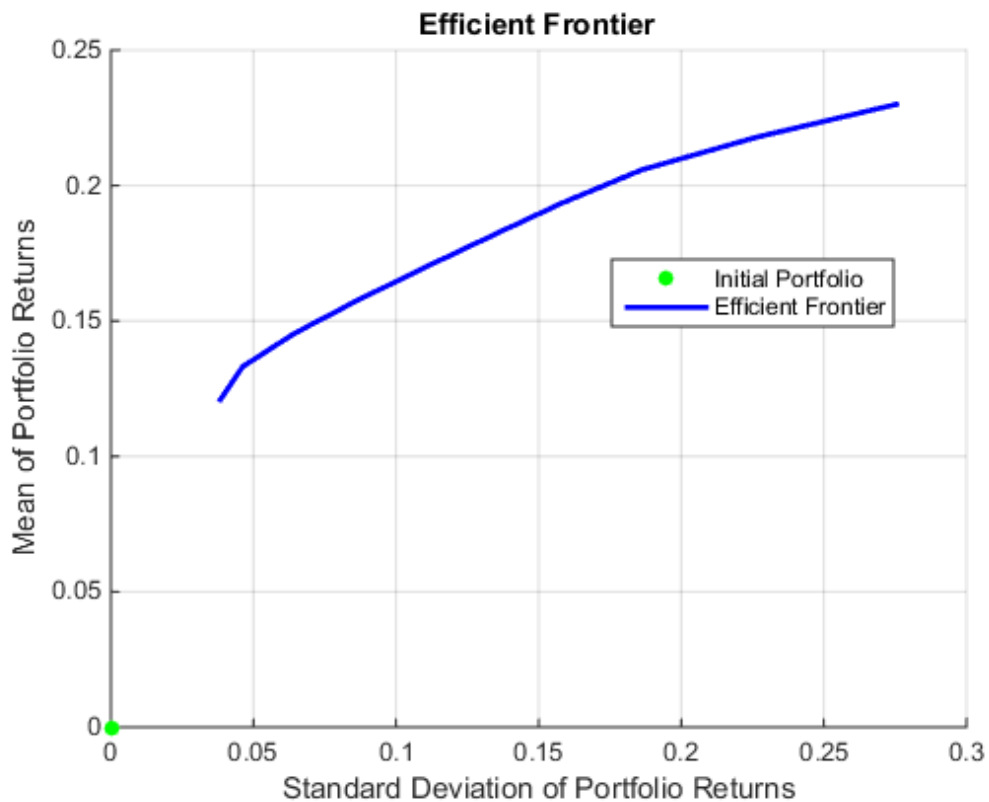
To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain One-Way Turnover Costs

Set one-way turnover costs.

```
p = Portfolio('AssetMean',[0.1, 0.2, 0.15], 'AssetCovar',...
[ 0.005, -0.010, 0.004; -0.010, 0.040, -0.002; 0.004, -0.002, 0.023]);
p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0);    %130-30 portfolio
plotFrontier(p);
```



Obtain one-way turnover costs.

```
[BuyTurnover, SellTurnover] = getOneWayTurnover(p)
```

```
BuyTurnover =
```

```
    1.3000
```

```
SellTurnover =
```

```
    0.3000
```

- “Working with One-Way Turnover Constraints” on page 4-75

See Also

Portfolio | setOneWayTurnover | setTurnover

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

getOneWayTurnover

Class: PortfolioCVaR

Obtain one-way turnover constraints from PortfolioCVaR object

Syntax

```
[BuyTurnover, SellTurnover] = getOneWayTurnover(obj)
```

Description

[BuyTurnover, SellTurnover] = `getOneWayTurnover(obj)` obtains one-way turnover constraints from a PortfolioCVaR object.

Tips

You can also use dot notation to get the one-way turnover constraints from a PortfolioCVaR object.

```
[BuyTurnover, SellTurnover] = obj.getOneWayTurnover
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

Output Arguments

BuyTurnover

Turnover constraint on purchases [scalar].

SellTurnover

Turnover constraint on sales [scalar].

Definitions

One-Way Turnover Constraint

One-way turnover constraints ensure that estimated optimal CVaR portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the form

$$1^T \times \max\{0, x - x_0\} \leq \tau_B$$

$$1^T \times \max\{0, x_0 - x\} \leq \tau_S$$

with

- x — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper bound for turnover constraint on purchases (scalar)
- τ_S — Upper bound for turnover constraint on sales (scalar)

Specify one-way turnover constraints using these properties in the CVaR portfolio object: `BuyTurnover` for τ_B , `SellTurnover` for τ_S , and `InitPort` for x_0 .

Note: The average turnover constraint (which is set using `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Attributes

Access	public
Static	false

Hidden

false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain One-Way Turnover Costs

Set one-way turnover costs and obtain the buy and sell turnover values.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0); %130-30 portfolio

[BuyTurnover, SellTurnover] = getOneWayTurnover(p)

BuyTurnover =

    1.3000

SellTurnover =

    0.3000
```

- “Working with One-Way Turnover Constraints” on page 5-74

See Also

PortfolioCVaR | setOneWayTurnover

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

getOneWayTurnover

Class: PortfolioMAD

Obtain one-way turnover constraints from PortfolioMAD object

Syntax

```
[BuyTurnover, SellTurnover] = getOneWayTurnover(obj)
```

Description

[BuyTurnover, SellTurnover] = getOneWayTurnover(obj) obtains one-way turnover constraints from a PortfolioMAD object.

Tips

You can also use dot notation to get the one-way turnover constraints from a PortfolioMAD object.

```
[BuyTurnover, SellTurnover] = obj.getOneWayTurnover
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Output Arguments

BuyTurnover

Turnover constraint on purchases [scalar].

SellTurnover

Turnover constraint on sales [scalar].

Definitions

One-Way Turnover Constraint

One-way turnover constraints ensure that estimated optimal MAD portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the form

$$1^T \times \max\{0, x - x_0\} \leq \tau_B$$

$$1^T \times \max\{0, x_0 - x\} \leq \tau_S$$

with

- x — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper bound for turnover constraint on purchases (scalar)
- τ_S — Upper bound for turnover constraint on sales (scalar)

Specify one-way turnover constraints using these properties in the MAD portfolio object: `BuyTurnover` for τ_B , `SellTurnover` for τ_S , and `InitPort` for x_0 .

Note: The average turnover constraint (which is set using `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Attributes

Access	public
Static	false

Hidden false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Obtain One-Way Turnover Costs

Set one-way turnover costs and obtain the buy and sell turnover values.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0); %130-30 portfolio

[BuyTurnover, SellTurnover] = getOneWayTurnover(p)

BuyTurnover =

    1.3000

SellTurnover =

    0.3000
```

- “Working with One-Way Turnover Constraints” on page 6-74

See Also

PortfolioCVaR | setOneWayTurnover

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

getnameidx

Find name in list

Syntax

```
nameidx = getnameidx(list, name)
```

Arguments

<code>list</code>	Cell array of name strings.
<code>name</code>	String or cell array of name strings.

Description

`nameidx = getnameidx(list, name)` finds the occurrence of a name or set of names in a list. It returns an index (order number) indicating where the specified names are located within the list. If `name` is not found, `nameidx` returns `0`.

If `name` is a cell array of names, `getnameidx` returns a vector containing the indices (order number) of the `name` strings within `list`. If none of the names in the `name` cell array is in `list`, it returns zero. If some of the names in `name` are not found, the indices for these names will be zeros.

`getnameidx` finds only the first occurrence of the name in the list of names. This function is meant to be used on a list of unique names (strings) only. It does not find multiple occurrences of a name or a list of names within `list`.

Examples

Given

```
poultry = {'duck', 'chicken'}  
animals = {'duck', 'cow', 'sheep', 'horse', 'chicken'}  
nameidx = getnameidx(animals, poultry)
```

```
ans =  
    1    5
```

Given

```
poultry = {'duck', 'goose', 'chicken'}  
animals = {'duck', 'cow', 'sheep', 'horse', 'chicken'}  
nameidx = getnameidx(animals, poultry)
```

```
ans =  
    1    0    5
```

See Also

[strcmp](#) | [strfind](#)

heston

Create Heston model

Syntax

```
heston = heston(Return, Speed, Level, Volatility)
```

```
heston = heston(Return, Speed, Level, Volatility, 'Name1', Value1,  
'Name2', Value2, ...)
```

Class

Heston

Description

This constructor creates and displays **heston** objects, which derive from the **SDEDDO** (SDE from drift and diffusion objects) class. Use **heston** objects to simulate sample paths of two state variables. Each state variable is driven by a single Brownian motion source of risk over **NPERIODS** consecutive observation periods, approximating continuous-time stochastic volatility processes.

Heston models are bivariate composite models. Each Heston model consists of two coupled univariate models:

- A geometric Brownian motion (**GBM**) model with a stochastic volatility function.

$$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

This model usually corresponds to a price process whose volatility (variance rate) is governed by the second univariate model.

- A Cox-Ingersoll-Ross (**CIR**) square root diffusion model.

$$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

This model describes the evolution of the variance rate of the coupled GBM price process.

Input Arguments

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Return	<p>If you specify Return as a scalar, it represents the expected (mean) instantaneous rate of return of the univariate GBM price model. As a deterministic function of time, when Return is called with a real-valued scalar time \mathbf{t} as its only input, Return must produce a scalar. If you specify it as a function of time and state, Return calculates the instantaneous rate of return of the GBM price model. This function generates a scalar when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • A 2-by-1 bivariate state vector X_t.
Speed	<p>If you specify Speed as a scalar, it represents the mean-reversion speed of the univariate CIR stochastic variance model (the speed at which the CIR variance reverts to its long-run average level). As a deterministic</p>

	<p>function of time, when Speed is called with a real-valued scalar time t as its only input, Speed must produce a scalar. If you specify it as a function time and state, Speed calculates the speed of mean reversion of the CIR variance model. This function generates a scalar when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • A 2-by-1 state vector X_t.
Level	<p>If you specify Level as a scalar, it represents the reversion level of the univariate CIR stochastic variance model. As a deterministic function of time, when Level is called with a real-valued scalar time t as its only input, Level must produce a scalar. If you specify it as a function time and state, Level calculates the reversion level of the CIR variance model. This function generates a scalar when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • A 2-by-1 state vector X_t.
Volatility	<p>If you specify Volatility as a scalar, it represents the instantaneous volatility of the CIR stochastic variance model, often called the <i>volatility of volatility</i> or <i>volatility of variance</i>. As a deterministic function of time, when Volatility is called with a real-valued scalar time t as its only input, Volatility must produce a scalar. If you specify it as a function time and state, Volatility generates a scalar when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • A 2-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

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:

- 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.

The following table lists valid parameter names.

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	<p>Scalar, 2-by-1 column vector, or 2-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, heston applies the same initial value to both state variables on all trials.</p> <p>If StartState is a bivariate column vector, heston applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, heston applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify Correlation as a scalar, a 2-by-2 positive semidefinite matrix, or as a deterministic function $C(t)$ that accepts the current time t and returns an 2-by-2 positive semidefinite correlation matrix.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is a 2-by-2 identity matrix representing independent Gaussian processes.</p>

Simulation	A user-defined simulation function or SDE simulation method. If you do not specify a value for <code>Simulation</code> , the default method is simulation by Euler approximation (<code>simByEuler</code>).
------------	---

Output Arguments

heston	<p>Object of class <code>heston</code> with the following displayed parameters:</p> <ul style="list-style-type: none"> • <code>StartTime</code>: Initial observation time • <code>StartState</code>: Initial state at <code>StartTime</code> • <code>Correlation</code>: Access function for the <code>Correlation</code> input, callable as a function of time • <code>Drift</code>: Composite drift-rate function, callable as a function of time and state • <code>Diffusion</code>: Composite diffusion-rate function, callable as a function of time and state • <code>Simulation</code>: A simulation function or method • <code>Return</code>: Access function for the input argument <code>Return</code>, callable as a function of time and state • <code>Speed</code>: Access function for the input argument <code>Speed</code>, callable as a function of time and state • <code>Level</code>: Access function for the input argument <code>Level</code>, callable as a function of time and state • <code>Volatility</code>: Access function for the input argument <code>Volatility</code>, callable as a function of time and state
--------	--

Examples

See “Creating Heston Stochastic Volatility Models” on page 17-31.

References

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.

Ait-Sahalia, Y., "Transition Densities for Interest Rate and Other Nonlinear Diffusions," *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

cir | gbm | sdeddo

hhigh

Highest high

Syntax

```
hhv = hhigh(data)
```

```
hhv = hhigh(data, nperiods, dim)
```

```
hhvts = hhigh(tsobj, nperiods)
```

```
hhvts = hhigh(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

<code>data</code>	Data series matrix.
<code>nperiods</code>	(Optional) Number of periods. Default = 14.
<code>dim</code>	(Optional) Dimension.
<code>tsobj</code>	Financial time series object.
<code>ParameterName</code>	The valid parameter name is: <ul style="list-style-type: none">• <code>HighName</code>: high prices series name
<code>ParameterValue</code>	The parameter value is a string that represents the valid parameter name.

Description

`hhv = hhigh(data)` generates a vector of highest high values for the past 14 periods from the matrix `data`.

`hhv = hhigh(data, nperiods, dim)` generates a vector of highest high values for the past `nperiods` periods. `dim` indicates the direction in which the highest high is to be searched. If you input `[]` for `nperiods`, the default is 14.

`hhvts = hhigh(tsobj, nperiods)` generates a vector of highest high values from `tsobj`, a financial time series object. `tsobj` must include at least the series `High`. The output `hhvts` is a financial time series object with the same dates as `tsobj` and data series named `HighestHigh`. If `nperiods` is specified, `hhigh` generates a financial time series object of highest high values for the past `nperiods` periods.

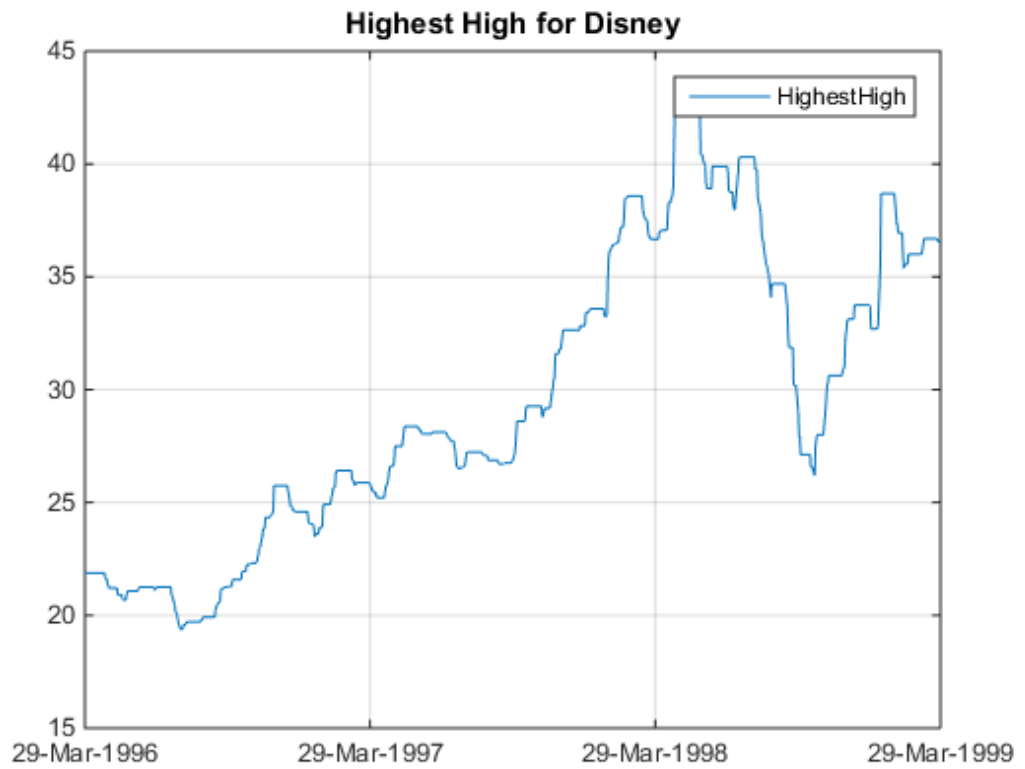
`hhvts = hhigh(tsobj, nperiods, ParameterName, ParameterValue)` specifies the name for the required data series when it is different from the default name. The valid parameter name is `isHighName`. The parameter value is a string that represents the valid parameter name.

Examples

Compute the Highest High Price

This example shows how to compute the highest high price for Disney stock and plot the results.

```
load disney.mat
dis_HHigh = hhigh(dis);
plot(dis_HHigh)
title('Highest High for Disney')
```



See Also
llow

highlow (fts)

Time series High-Low plot

Syntax

```
highlow(tsobj)
```

```
highlow(tsobj, color)
```

```
highlow(tsobj, color, dateform)
```

```
highlow(tsobj, color, dateform, ParameterName, ParameterValue, ...)
```

```
hll = highlow(tsobj, color, dateform, ParameterName,  
ParameterValue, ...)
```

Arguments

tsobj	Financial time series object.
color	(Optional) A three-element row vector representing RGB or a color identifier. (See <code>plot</code> in the MATLAB documentation.)
dateform	(Optional) Date string format used as the <i>x</i> -axis tick labels. (See <code>datetick</code> in the MATLAB documentation.) You can specify a <code>dateform</code> only when <code>tsobj</code> does not contain time-of-day data. If <code>tsobj</code> contains time-of-day data, <code>dateform</code> is restricted to 'dd-mmm-yyyy HH:MM'.
ParameterName	ParameterName can be: <ul style="list-style-type: none"> • <code>HighName</code>: high prices series name • <code>LowName</code>: low prices series name • <code>OpenName</code>: open prices series name • <code>CloseName</code>: closing prices series name
ParameterValue	The parameter value is a string that represents the valid parameter name.

Description

`highlow(tsoobj)` generates a High-Low plot of the data in the financial time series object `tsoobj`. `tsoobj` must contain at least four data series representing the high, low, open, and closing prices. These series must have the names `High`, `Low`, `Open`, and `Close` (case-insensitive).

`highlow(tsoobj, color)` additionally specifies the color of the plot.

`highlow(tsoobj, color, dateform)` additionally specifies the date string format used as the *x*-axis tick labels. See `datestr` for a list of date string formats.

`highlow(tsoobj, color, dateform, ParameterName, ParameterValue, ...)` indicates the actual name(s) of the required data series if the data series do not have the default names.

You can specify open prices as optional by providing the parameter name `'OpenName'` and the parameter value `''` (empty string).

```
highlow(tsoobj, color, dateform, 'OpenName', '')
```

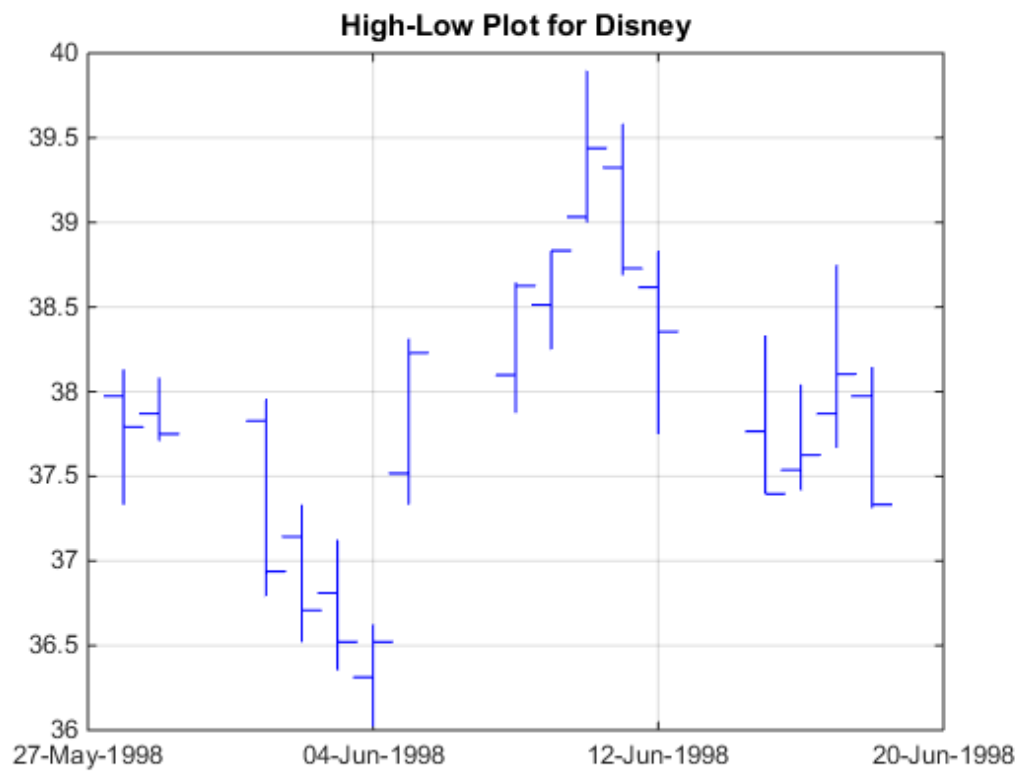
`hhll = highlow(tsoobj, color, dateform, ParameterName, ParameterValue, ...)` returns the handle to the line object that makes up the High-Low plot.

Examples

Generate a High-Low Plot

This example shows how to generate a High-Low plot for Disney stock for the dates May 28 to June 18, 1998.

```
load disney.mat
highlow(dis('28-May-1998:18-Jun-1998'))
title('High-Low Plot for Disney')
```

See Also
candle

highlow

High, low, open, close chart

Syntax

```
highlow(High, Low, Close, Open, Color)
```

```
highlow(High, Low, Close, Open, Color, Dates, Dateform)
```

```
Handles = highlow(High, Low, Close, Open, Color, Dates, Dateform)
```

Arguments

High	High prices for a security. A column vector.
Low	Low prices for a security. A column vector.
Close	Closing prices for a security. A column vector.
Open	(Optional) Opening prices for a security. A column vector. To specify Color when Open is unknown, enter Open as an empty matrix [].
Color	(Optional) Vertical line color. A string. MATLAB software supplies a default color if none is specified. The default color differs depending on the background color of the figure window. See ColorSpec in the MATLAB documentation for color names.
Dates	(Optional) User-defined dates. A column vector.
Dateform	(Optional) Format of the date string as tick labels. For more information on date string formats, see dateaxis .

Description

`highlow(High, Low, Close, Open, Color)` plots the high, low, opening, and closing prices of an asset. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right.

`highlow(High, Low, Close, Open, Color, Dates, Dateform)` plots the high, low, opening, and closing prices of an asset. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right. The plot also contains user-defined dates and date string format for tick labels.

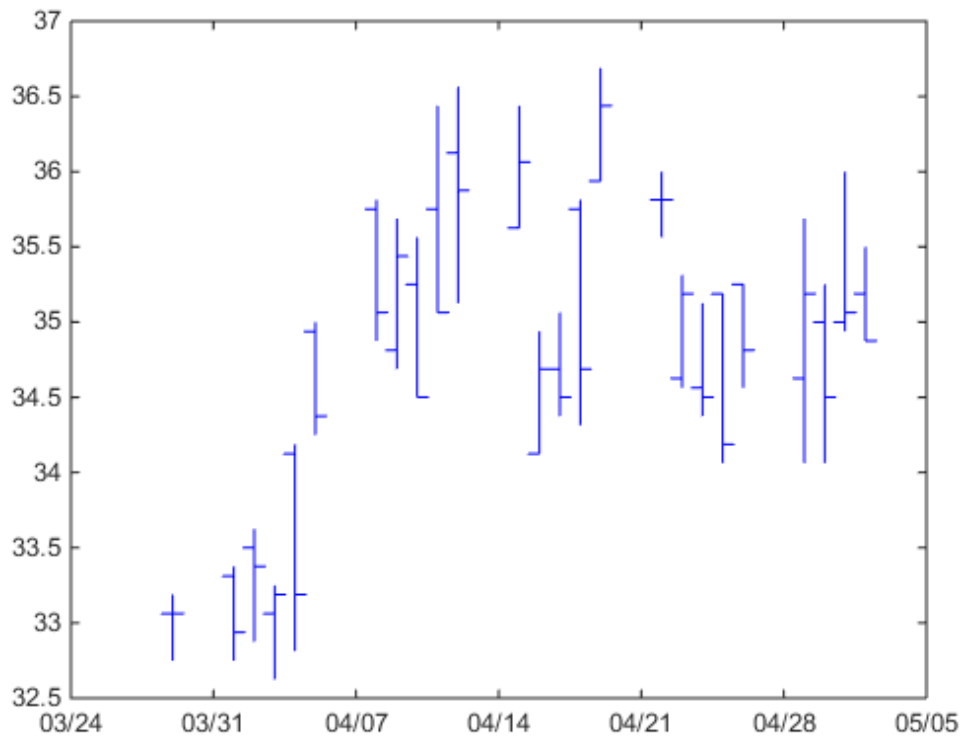
`Handles = highlow(High, Low, Close, Open, Color, Dates, Dateform)`
plots the figure and returns the handles of the lines.

Examples

Create a HighLow Chart

The high, low, open, and closing prices for the equity DIS are stored in equal-length vectors `dis_HIGH`, `dis_LOW`, `dis_OPEN`, and `dis_CLOSE` respectively and `highlow` plots the price data using blue lines.

```
load disney.mat
range = 1:25;
highlow(dis_HIGH(range), dis_LOW(range), dis_CLOSE(range), ...
        dis_OPEN(range), 'blue', dis.dates(range));
```



- “High-Low-Close Chart Example”

See Also

`bolling` | `candle` | `dateaxis` | `highlow` | `movavg` | `pointfig`

hist

Histogram

Syntax

```
hist(tsoj, numbins)
```

```
ftshist = hist(tsoj, numbins)
```

```
[ftshist, binpos] = hist(tsoj, numbins)
```

Arguments

<code>tsoj</code>	Financial time series object.
<code>numbins</code>	(Optional) Number of histogram bins. Default = 10.

Description

`hist(tsoj, numbins)` calculates and displays the histogram of the data series contained in the financial time series object `tsoj`.

`ftshist = hist(tsoj, numbins)` calculates, but does not display, the histogram of the data series contained in the financial time series object `tsoj`. The output `ftshist` is a structure with field names similar to the data series names of `tsoj`.

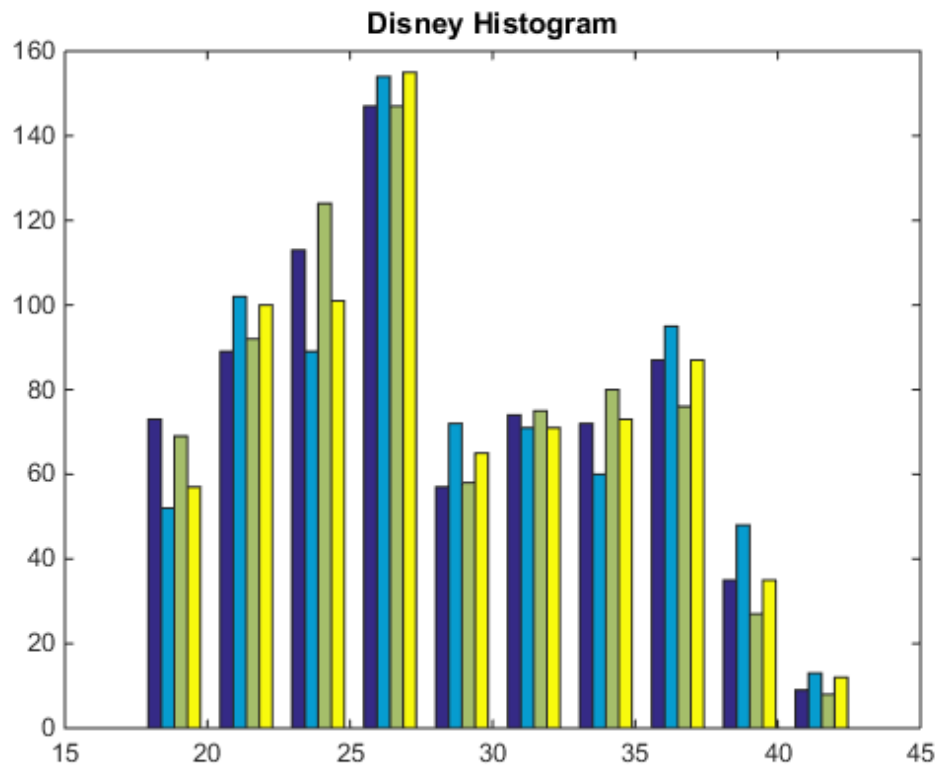
`[ftshist, binpos] = hist(tsoj, numbins)` additionally returns the bin positions `binpos`. The positions are the centers of each bin. `binpos` is a column vector.

Examples

Create a Histogram

This example shows how to generate a histogram of Disney open, high, low, and close prices.

```
load disney.mat
dis = rmfield(dis, 'VOLUME'); % Remove VOLUME field
hist(dis)
title('Disney Histogram')
```



See Also

[mean](#) | [std](#) | [hist](#)

hvw

Create HWV model

Syntax

```
HWV = hvw(Speed, Level, Sigma)
```

```
HWV = hvw(Speed, Level, Sigma, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Class

HWV

Description

This constructor creates and displays HWV objects, which derive from the SDEMVD (SDE with drift rate expressed in mean-reverting form) class. 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:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$$

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

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Speed	<p>Speed represents the function S. 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). As a deterministic function of time, when Speed is called with a real-valued scalar time \mathbf{t} as its only input, Speed must produce an NVARS-by-NVARS matrix. If you specify Speed as a function of time and state, it calculates the speed of mean reversion. This function must generate an NVARS-by-NVARS matrix of reversion rates when called with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t.
Level	<p>Level represents the function L. If you specify Level as an array, it must be an NVARS-by-1 column vector of reversion levels. As a deterministic function of time, when Level is called with a real-valued scalar time \mathbf{t} as its only input, Level must produce an NVARS-by-1 column vector. If you specify Level as a function of time and state, it must generate an NVARS-by-1 column vector of reversion levels when called with two inputs:</p>

	<ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V. 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. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARs-by-NBROWNS matrix. If you specify it as a function of time and state, Sigma must return an NVARs-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • 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

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:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, hwv applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, hwv applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, hwv applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>
Simulation	<p>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).</p>

Output Arguments

HWV	<p>Object of class hwv with the following displayed parameters:</p> <ul style="list-style-type: none"> • 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

Examples

“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 17-29

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`drift` | `diffusion` | `sdeddo`

holdings2weights

Portfolio holdings into weights

Syntax

```
Weights = holdings2weights(Holdings, Prices, Budget)
```

Arguments

Holdings	Number of portfolios (NPORTS) by number of assets (NASSETS) matrix with the holdings of NPORTS portfolios containing NASSETS assets.
Prices	NASSETS vector of asset prices.
Budget	(Optional) Scalar or NPORTS vector of nonzero budget constraints. Default = 1.

Description

`Weights = holdings2weights(Holdings, Prices, Budget)` converts portfolio holdings into portfolio weights. The weights must satisfy a budget constraint such that the weights sum to `Budget` for each portfolio.

`Weights` is a NPORTS by NASSETS matrix containing the normalized weights of NPORTS portfolios containing NASSETS assets.

Notes

- `Holdings` may be negative to indicate a short position, but the overall portfolio weights must satisfy a nonzero budget constraint.
 - The weights in each portfolio sum to the `Budget` value (which is 1 if `Budget` is unspecified.)
-

See Also

weights2holdings

holidays

Holidays and nontrading days

Syntax

`H = holidays`

`H = holidays(StartDate, EndDate)`

`H = holidays(StartDate, EndDate, AltHolidays)`

Description

`H = holidays` returns a vector of serial date numbers corresponding to all holidays and nontrading days.

`H = holidays(StartDate, EndDate)` returns a vector of serial date numbers corresponding to the holidays and nontrading days between `StartDate` and `EndDate`, inclusive.

`H = holidays(StartDate, EndDate, AltHolidays)` returns a vector of serial date numbers corresponding to the alternate list of holidays and nontrading days.

Input Arguments

StartDate

Start date. Enter as a serial date number or date string.

EndDate

End date. Enter as a serial date number or date string.

AltHolidays

Alternate list of holidays and nontrading days stored as serial date numbers.

Output Arguments

H

Returns a vector of serial date numbers corresponding to all holidays and nontrading days.

Definitions

`holidays` is based on a modern 5-day workweek. This function contains all holidays and special nontrading days for the New York Stock Exchange from January 1, 1885 to December 31, 2050. Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures from 1885 to 2050 should include Saturday trading days. To capture these dates, use the function `nyseclosures`. The results from `holidays` and `nyseclosures` are identical if the `WorkWeekFormat` in `nyseclosures` is `'Modern'`.

Examples

Create a vector of serial date numbers corresponding to all holidays and nontrading dates between a specified `StartDate` and `EndDate`:

```
H = holidays('jan 1 2001', 'jun 23 2001')
```

This returns:

```
H =  
  
    730852  
    730866  
    730901  
    730954  
    730999
```

The serial date numbers for these values are:

```
01-Jan-2001 (New Year's Day)  
15-Jan-2001 (Martin Luther King Day)  
19-Feb-2001 (President's Day)
```


13-Apr-2001 (Good Friday)
28-May-2001 (Memorial Day)

See Also

[busdate](#) | [createholidays](#) | [fbusdate](#) | [isbusday](#) | [lbusdate](#) | [nyseclosures](#)

horzcat

Concatenate financial time series objects horizontally

Syntax

```
horzcat
```

Description

`horzcat` implements horizontal concatenation of financial time series objects. `horzcat` essentially merges the data columns of the financial time series objects. The time series objects must contain the exact same dates and times.

When multiple instances of a data series name occur, concatenation adds a suffix to the current names of the data series. The suffix has the format `_objectname<n>`, where `n` is a number indicating the position of the time series, from left to right, in the concatenation command. The `n` part of the suffix appears only when there is more than one instance of a particular data series name.

The description fields are concatenated as well. They are separated by two forward slashes (`//`).

Examples

Construct three financial time series, each containing a data series named `DataSeries`:

```
firstfts = fints((today:today+4)', (1:5)', 'DataSeries', 'd');
secondfts = fints((today:today+4)', (11:15)', 'DataSeries', 'd');
thirdfts = fints((today:today+4)', (21:25)', 'DataSeries', 'd');
```

Concatenate the time series horizontally into a new financial time series `newfts`:

```
newfts = [firstfts secondfts thirdfts secondfts];
```

The resulting object `newfts` has data series names `DataSeries_firstfts`, `DataSeries_secondfts2`, `DataSeries_thirdfts`, and `DataSeries_secondfts4`.

Verify this with the command

```
fieldnames(newfts)
ans =
    'desc'
    'freq'
    'dates'
    'DataSeries_firstfts'
    'DataSeries_secondfts2'
    'DataSeries_thirdfts'
    'DataSeries_secondfts4'
    'times'
```

Use `chfield` to change the data series names.

Note If all input objects have the same frequency, the new object has that frequency as well. However, if one of the objects concatenated has a different frequency from the others, the frequency indicator of the resulting object is set to **Unknown (0)**.

See Also
vertcat

hour

Hour of date or time

Syntax

```
Hour = hour(Date)
```

Description

`Hour = hour(Date)` returns the hour of the day given a serial date number or a date string.

Examples

```
Hour = hour(730473.5584278936)
```

or

```
Hour = hour('19-dec-1999, 13:24:08.17')
```

returns

```
Hour =  
    13
```

See Also

`datevec` | `minute` | `second`

inforatio

Calculate information ratio for one or more assets

Syntax

```
inforatio(Asset, Benchmark)
```

```
Ratio = inforatio(Asset, Benchmark)
```

```
[Ratio, TE] = inforatio(Asset, Benchmark)
```

Arguments

Asset	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES observations of asset returns for NUMSERIES asset return series.
Benchmark	NUMSAMPLES vector of returns for a benchmark asset. The periodicity must be the same as the periodicity of Asset. For example, if Asset is monthly data, then Benchmark must be monthly returns.

Description

Given NUMSERIES assets with NUMSAMPLES returns for each asset in a NUMSAMPLES x NUMSERIES matrix **Asset** and given a NUMSAMPLES vector of benchmark returns in **Benchmark**, **inforatio** computes the information ratio and tracking error for each asset relative to the **Benchmark**.

To summarize the outputs of **inforatio**:

- **Ratio** is a 1 x NUMSERIES row vector of information ratios for each series in **Asset**. Any series in **Asset** with a tracking error of 0 will have a NaN value for its information ratio.
- **TE** is a 1 x NUMSERIES row vector of tracking errors, that is, the standard deviation of **Asset** relative to **Benchmark** returns, for each series.

Note: NaN values in the data are ignored. If the **Asset** and **Benchmark** series are identical, the information ratio will be NaN since the tracking error is 0. The information ratio and the Sharpe ratio of an **Asset** versus a riskless **Benchmark** (a **Benchmark** with standard deviation of returns equal to 0) are equivalent. This equivalence is not necessarily true if the **Benchmark** is risky.

Examples

See “Information Ratio” on page 7-7.

References

Richard C. Grinold and Ronald N. Kahn, *Active Portfolio Management*, 2nd. ed., McGraw-Hill, 2000.

Jack Treynor and Fischer Black, "How to Use Security Analysis to Improve Portfolio Selection," *Journal of Business*, Vol. 46, No. 1, January 1973, pp. 66-86.

See Also

portalalpha | sharpe

interpolate

Brownian interpolation of stochastic differential equations

Syntax

```
[XT, T] = interpolate(MDL, T, Paths)
```

```
[XT, T] = interpolate(MDL, Paths, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Classes

All classes in “SDE Class Hierarchy” on page 17-5.

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

MDL	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 <code>interpolate</code> performs the Brownian interpolation.

Optional Input Arguments

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:

- 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:

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 <code>interpolate</code> 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), <code>interpolate</code> bases the interpolation only on the state information specified in Paths. If you set Refine to TRUE, <code>interpolate</code> 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	<p>Function or cell array of functions that indicates a sequence of background processes or state vector adjustments of the form</p> $X_t = P(t, X_t)$ <p>The <code>interpolate</code> 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.</p> <p>If you specify more than one processing function, <code>interpolate</code> 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.</p> <p>If you do not specify a processing function, <code>interpolate</code> makes no adjustments and performs no processing.</p>
-----------	---

Output Arguments

XT	<p>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 <code>Paths</code> time series array.</p>
T	<p>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.</p>

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:

$$\begin{aligned}dX_{1t} &= 0.3dt + 0.2dW_{1t} - 0.1dW_{2t} \\dX_{2t} &= 0.4dt + 0.1dW_{1t} - 0.2dW_{2t} \\E[dW_{1t}dW_{2t}] &= \rho dt = 0.5dt\end{aligned}$$

- 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):

```

rng default % make output reproducible
dt = 1/250; % 1 trading day = 1/250 years
[X,T] = simulate(obj,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 = interpolate(obj,t,X,'Times',T);

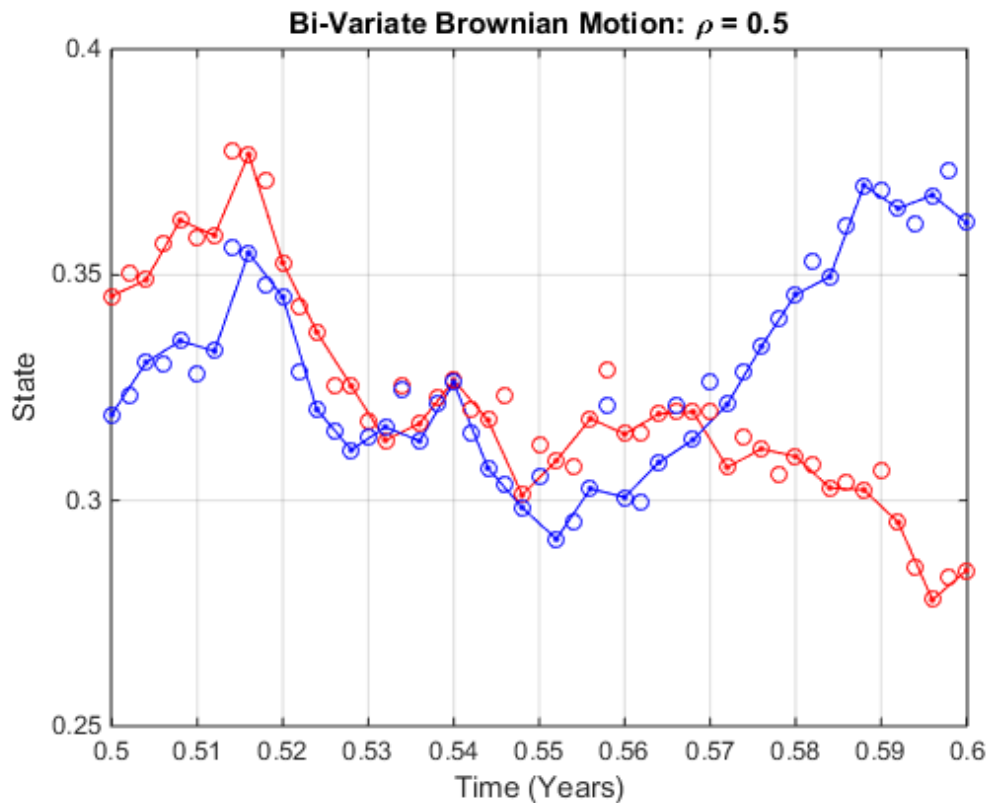
```

- b** Plot both the simulated and interpolated values:

```

plot(T,X(:,1),'.-r',T,X(:,2),'.-b')
grid on;
hold on;
plot(t,x(:,1),'or',t,x(:,2),'ob')
hold off;
xlabel('Time (Years)')
ylabel('State')
title('Bi-Variate Brownian Motion: \rho = 0.5')
axis([0.4999 0.6001 0.25 0.4])

```



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.

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:

```
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);

rng default; % make output reproducible
dt    = 1/250; % 1 trading day = 1/250 years
[X,T] = simulate(obj,250,'DeltaTime',dt);

n      = 125; % index of simulated state near middle
times  = (T(n):(dt/10):T(n + 1));
nTrials = 25000; % # of Trials at each time
```

- 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 = interpolate(obj,t(ones(nTrials,1)),...
        X,'Times',T);
    average(i) = mean(x(:,1));
    variance(i) = var(x(:,1));
end
```

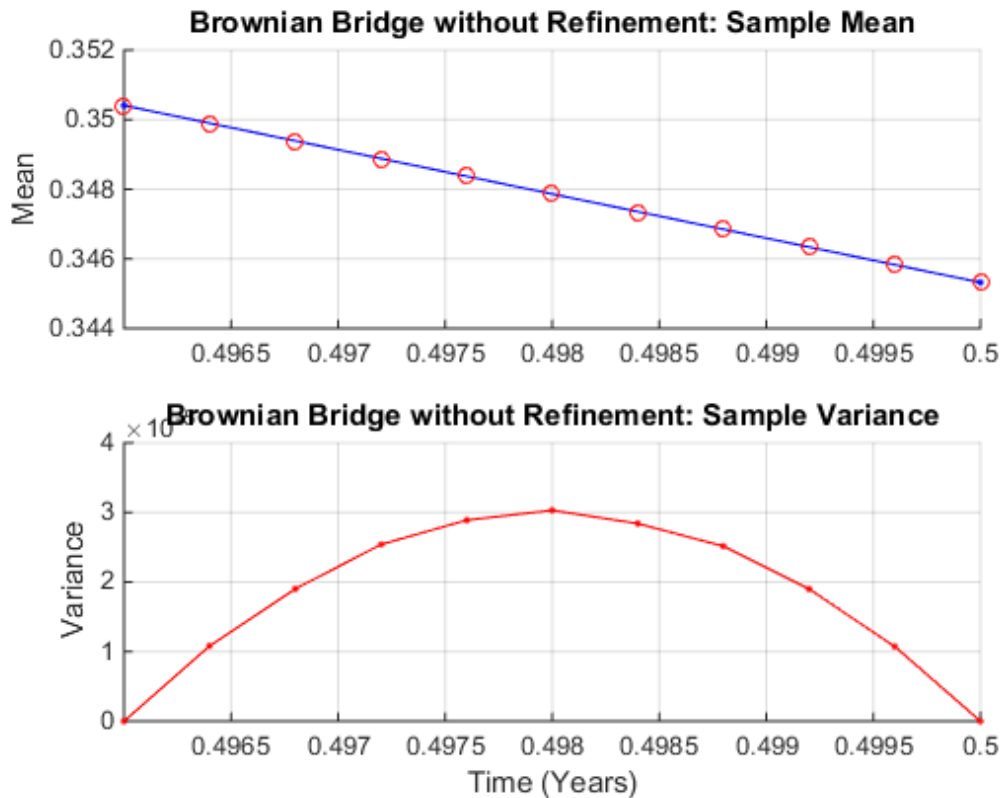
- 3 Plot the sample mean and variance of each state variable:

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 cluster 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.

More About

Algorithms

- 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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

simulate | sde

irr

Internal rate of return

Syntax

```
Return = irr(CashFlow)
[Return, AllRates] = irr(CashFlow)
```

Description

`Return = irr(CashFlow)` calculates the internal rate of return for a series of periodic cash flows.

`[Return, AllRates] = irr(CashFlow)` calculates the internal rate of return and a vector of all internal rates for a series of periodic cash flows.

Input Arguments

CashFlow

A vector containing a stream of periodic cash flows. The first entry in `CashFlow` is the initial investment. If `CashFlow` is a matrix, `irr` handles each column of `CashFlow` as a separate cash-flow stream.

Output Arguments

Return

An internal rate of return associated to `CashFlow`. If `CashFlow` is a matrix, then `Return` is a vector whose entry `j` is an internal rate of return for column `j` in `CashFlow`.

AllRates

A vector containing all the internal rates of return associated with `CashFlow`. If `CashFlow` is a matrix, then `AllRates` is also a matrix, with the same number of

columns as `CashFlow` and one less row. Also, column `j` in `AllRates` contains all the rates of return associated to column `j` in `CashFlow` (including complex-valued rates).

Definitions

`irr` uses the following conventions:

- If one or more internal rate of returns (warning if multiple) are strictly positive rates, `Return` sets to the minimum.
- If no strictly positive rate of returns, but one or multiple (warning if multiple) returns are nonpositive rates, `Return` sets to the maximum.
- If no real-valued rates exist, `Return` sets to `NaN` (no warnings).

Examples

Find the internal rate of return for a simple investment with a unique positive rate of return. The initial investment is \$100,000 and the following cash flows represent the yearly income from the investment.

- Year 1 — \$10,000
- Year 2 — \$20,000
- Year 3 — \$30,000
- Year 4 — \$40,000
- Year 5 — \$50,000

Calculate the internal rate of return on the investment:

```
Return = irr([-100000 10000 20000 30000 40000 50000])
```

This returns:

```
Return =  
0.1201
```

If the cash flow payments were monthly, then the resulting rate of return is multiplied by 12 for the annual rate of return.

Find the internal rate of return for multiple rates of return. The project has the following cash flows and a market rate of 10%.

```
CashFlow = [-1000 6000 -10900 5800]
```

Use `irr` with a single output argument:

```
Return = irr(CashFlow)
```

A warning appears and `irr` returns a 100% rate of return. The 100% rate on the project looks attractive:

```
Warning: Multiple rates of return
```

```
> In irr at 166
```

```
Return =
```

```
1.0000
```

Use `irr` with two output arguments:

```
[Return, AllRates] = irr(CashFlow)
```

This returns:

```
>> [Return, AllRates] = irr(CashFlow)
```

```
Return =
```

```
1.0000
```

```
AllRates =
```

```
-0.0488
```

```
1.0000
```

```
2.0488
```

The rates of return in `AllRates` are -4.88%, 100%, and 204.88%. Though some rates are lower and some higher than the market rate, based on the work of Hazen, any rate gives a consistent recommendation on the project. However, you can use a present value analysis in these kinds of situations. To check the present value of the project, use `pvvar`:

```
PV = pvvar(CashFlow,0.10)
```

This returns:

PV =

-196.0932

The second argument is the 10% market rate. The present value is -196.0932, negative, so the project is undesirable.

More About

- “Interest Rates/Rates of Return” on page 2-16

References

Brealey and Myers, *Principles of Corporate Finance*, McGraw-Hill Higher Education, Chapter 5, 2003.

Hazen G., “A New Perspective on Multiple Internal Rates of Return,” *The Engineering Economist*, Vol. 48-1, 2003, pp. 31-51.

See Also

effrr | mirr | nomrr | xirr | pvvar

isbusday

True for dates that are business days

Syntax

```
Busday = isbusday(Date, Holiday, Weekend)
```

Arguments

Date	Date(s) being checked. Enter as a vector of serial date numbers or date strings. Date can contain multiple dates, but they must all be in the same format. Dates are assumed to be whole date numbers or date stamps with no fractional or time values.
Holiday	(Optional) Vector of holidays and nontrading-day dates. All dates in Holiday must be the same format: either serial date numbers or date strings. (Using date numbers improves performance.) The holidays function supplies the default vector.
Weekend	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), then Weekend = [1 0 0 0 0 0 1].

Description

`Busday = isbusday(Date, Holiday, Weekend)` returns logical true (1) if **Date** is a business day and logical false (0) otherwise.

Examples

Example 1:

```
Busday = isbusday('16 jun 2001')
```

```

Busday =
    0

Date = ['15 feb 2001'; '16 feb 2001'; '17 feb 2001'];
Busday = isbusday(Date)

Busday =
    1
    1
    0

```

Example 2: Set June 21, 2003 (a Saturday) as a business day.

```

Weekend = [1 0 0 0 0 0 0];
isbusday('June 21, 2003', [], Weekend)

ans =
    1

```

Note: If the second argument, `Holiday`, is empty (`[]`), the default `Holidays` vector (generated with `holidays` and then associated to the NYSE calendar) is used. To consider a calendar without holidays (except for weekends as controlled by third input), use a serial (or text date) out of your range. For example `0-Jan-0` or the value `0` are with high probabilities out of the range. This value overwrites the default calendar allowing you to remove any specific country holiday:

```

isbusday(datenum('06092010','ddmmyyy'),0)

ans =
    1

```

See Also

`busdate` | `fbusdate` | `holidays` | `lbusdate`

iscompatible

Structural equality

Syntax

```
iscomp = iscompatible(tsobj_1, tsobj_2)
```

Arguments

<code>tsobj_1, tsobj_2</code>	A pair of financial time series objects.
-------------------------------	--

Description

`iscomp = iscompatible(tsobj_1, tsobj_2)` returns 1 if both financial time series objects `tsobj_1` and `tsobj_2` have the same dates and data series names. It returns 0 if any component is different.

`iscomp = 1` indicates that the two objects contain the same number of data points and equal number of data series. However, the values contained in the data series can be different.

Note Data series names are case-sensitive.

See Also

`isequal`

isequal

Multiple object equality

Syntax

```
iseq = isequal(tsojb_1, tsojb_2, ...)
```

Arguments

<code>tsojb_1 ...</code>	A list of financial time series objects.
--------------------------	--

Description

`iseq = isequal(tsojb_1, tsojb_2, ...)` returns 1 if all listed financial time series objects have the same dates, data series names, and values contained in the data series. It returns 0 if any of those components is different.

Note Data series names are case-sensitive.

`iseq = 1` implies that each object contains the same number of dates and the same data. Only the descriptions can differ.

See Also

`eq` | `iscompatible`

isempty

True for empty financial time series objects

Syntax

```
tf = isempty(fts)
```

Arguments

<code>fts</code>	Financial time series object.
------------------	-------------------------------

Description

`isempty` for financial times series objects is based on the MATLAB `isempty` function. See `isempty` in the MATLAB documentation.

`tf = isempty(fts)` returns true (1) if `fts` is an empty financial time series object and false (0) otherwise. An empty financial times series object has no elements, that is, `length(fts) = 0`.

See Also

`nanmax` | `nanmean` | `nanmin` | `nanstd` | `nanvar` | `nanmedian`

isfield

Check whether string is field name

Syntax

```
F = isfield(tsoobj, name)
```

Description

`F = isfield(tsoobj, name)` returns true (1) if `name` is the name of a data series in `tsoobj`. Otherwise, `isfield` returns false (0).

See Also

`fieldnames` | `getfield` | `setfield`

issorted

Check whether dates and times are monotonically increasing

Syntax

```
monod = issorted(tsobj)
```

Arguments

tsobj	Financial time series object
-------	------------------------------

Description

`monod = issorted(tsobj)` returns `1` if the dates and times in `tsobj` are monotonically increasing or `0` if they are not.

See Also

`sortfts`

kagi

Kagi chart

Syntax

kagi(X)

Arguments

X	M-by-2 matrix where the first column contains date numbers and the second column is the asset price.
---	--

Description

kagi(X) plots asset price with respect to dates.

Examples

Create a Kagi Chart

This example shows how to generate a Kagi chart for asset X that is an M-by-2 matrix of date numbers and asset prices.

X = [...

```

733299.00      41.99;...
733300.00      42.14;...
733303.00      41.93;...
733304.00      41.98;...
733305.00      41.75;...
733306.00      41.61;...
733307.00      42.29;...
733310.00      42.19;...

```

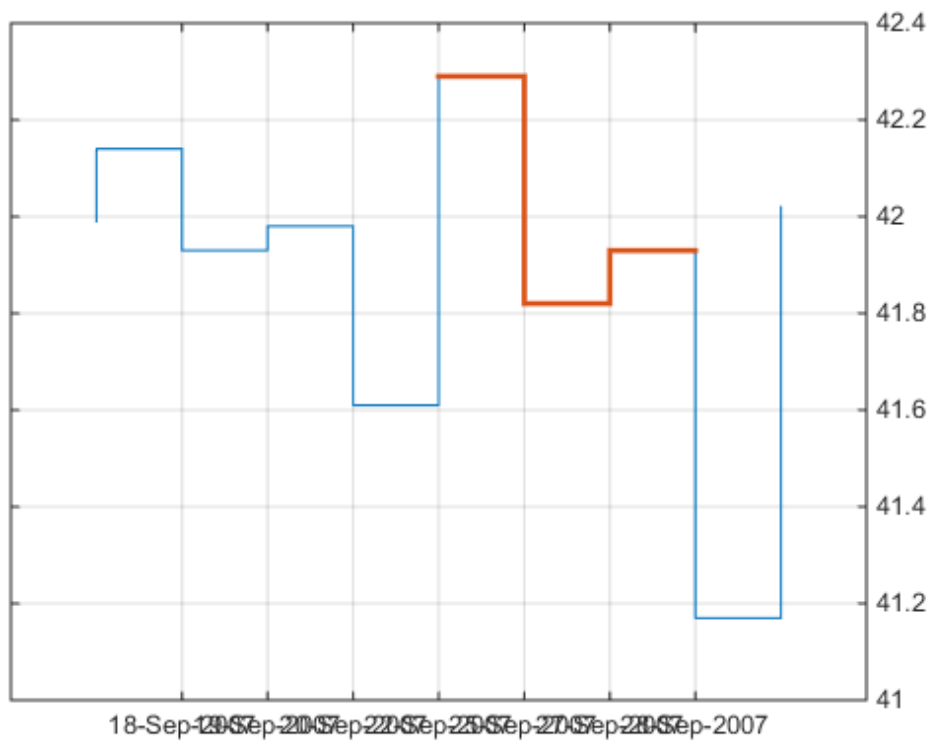
733311.00	41.82;...
733312.00	41.93;...
733313.00	41.81;...
733314.00	41.37;...
733317.00	41.17;...
733318.00	42.02]

kagi(X)

X =

1.0e+05 *

7.3330	0.0004
7.3330	0.0004
7.3330	0.0004
7.3330	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3332	0.0004
7.3332	0.0004



See Also

[bolling](#) | [candle](#) | [highlow](#) | [linebreak](#) | [movavg](#) | [pointfig](#) | [priceandvol](#) | [renko](#) | [volarea](#)

lagts

Lag time series object

Syntax

```
newfts = lagts(oldfts)
```

```
newfts = lagts(oldfts, lagperiod)
```

```
newfts = lagts(oldfts, lagperiod, padmode)
```

Arguments

<code>oldfts</code>	Financial time series object
<code>lagperiod</code>	Number of lag periods expressed in the frequency of the time series object
<code>padmode</code>	Data padding value

Description

`lagts` delays a financial time series object by a specified time step.

`newfts = lagts(oldfts)` delays the data series in `oldfts` by one time series date entry and returns the result in the object `newfts`. The end will be padded with zeros, by default.

`newfts = lagts(oldfts, lagperiod)` shifts time series values to the right on an increasing time scale. `lagts` delays the data series to happen at a later time. `lagperiod` is the number of lag periods expressed in the frequency of the time series object `oldfts`. For example, if `oldfts` is a daily time series, `lagperiod` is specified in days. `lagts` pads the data with zeros (default).

`newfts = lagts(oldfts, lagperiod, padmode)` lets you pad the data with an arbitrary value, NaN, or Inf rather than zeros by setting `padmode` to the desired value.

See Also
leadts

lbusdate

Last business date of month

Syntax

```
Date = lbusdate(Year, Month, Holiday, Weekend)
```

Arguments

Year	Enter as four-digit integer.
Month	Enter as integer from 1 through 12.
Holiday	(Optional) Vector of holidays and nontrading-day dates. All dates in Holiday must be the same format: either serial date numbers or date strings. (Using date numbers improves performance.) The holidays function supplies the default vector.
Weekend	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), then Weekend = [1 0 0 0 0 0 1].

Description

`Date = lbusdate(Year, Month, Holiday, Weekend)` returns the serial date number for the last business date of the given year and month. **Holiday** specifies nontrading days.

Year and **Month** can contain multiple values. If one contains multiple values, the other must contain the same number of values or a single value that applies to all. For example, if **Year** is a 1-by-n vector of integers, then **Month** must be a 1-by-n vector of integers or a single integer. **Date** is then a 1-by-n vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

Example 1.

```
Date = lbusdate(2001, 5)
```

```
Date =
```

```
731002
```

```
datestr(Date)
```

```
ans =
```

```
31-May-2001
```

Example 2. You can indicate that Saturday is a business day by appropriately setting the `Weekend` argument.

```
Weekend = [1 0 0 0 0 0 0];
```

May 31, 2003, is a Saturday. Use `lbusdate` to check that this Saturday is actually the last business day of the month.

```
Date = datestr(lbusdate(2003, 5, [], Weekend))
```

```
Date =
```

```
31-May-2003
```

See Also

`busdate` | `eomdate` | `fbusdate` | `holidays` | `isbusday`

leadts

Lead time series object

Syntax

```
newfts = leadts(oldfts)
```

```
newfts = leadts(oldfts, leadperiod)
```

```
newfts = leadts(oldfts, leadperiod, padmode)
```

Arguments

<code>oldfts</code>	Financial time series object.
<code>leadperiod</code>	Number of lead periods expressed in the frequency of the time series object.
<code>padmode</code>	Data padding value.

Description

`leadts` advances a financial time series object by a specified time step.

`newfts = leadts(oldfts)` advances the data series in `oldfts` by one time series date entry and returns the result in the object `newfts`. The end will be padded with zeros, by default.

`newfts = leadts(oldfts, leadperiod)` shifts time series values to the left on an increasing time scale. `leadts` advances the data series to happen at an earlier time. `leadperiod` is the number of lead periods expressed in the frequency of the time series object `oldfts`. For example, if `oldfts` is a daily time series, `leadperiod` is specified in days. `leadts` pads the data with zeros (default).

`newfts = leadts(oldfts, leadperiod, padmode)` lets you pad the data with an arbitrary value, NaN, or Inf rather than zeros by setting `padmode` to the desired value.

See Also
lagts

length

Get number of dates (rows)

Syntax

```
lenfts = length(tsoj)
```

Description

`lenfts = length(tsoj)` returns the number of dates (rows) in the financial time series object `tsoj`. This is the same as issuing `lenfts = size(tsoj, 1)`.

See Also

`size` | `length`

linebreak

Line break chart

Syntax

`linebreak(X)`

Arguments

X	M -by-2 matrix where the first column contains date numbers and the second column is the asset price.
---	---

Description

`linebreak(X)` plots asset price with respect to dates.

Examples

Create a Line Break Chart

This example shows how to generate a line break chart for asset X that is an M-by-2 matrix of date numbers and asset prices.

`X = [...`

```

733299.00      41.99;...
733300.00      42.14;...
733303.00      41.93;...
733304.00      41.98;...
733305.00      41.75;...
733306.00      41.61;...
733307.00      42.29;...
733310.00      42.19;...

```

733311.00	41.82;...
733312.00	41.93;...
733313.00	41.81;...
733314.00	41.37;...
733317.00	41.17;...
733318.00	42.02]

linebreak(X)

X =

1.0e+05 *

7.3330	0.0004
7.3330	0.0004
7.3330	0.0004
7.3330	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3331	0.0004
7.3332	0.0004
7.3332	0.0004



See Also

[bolling](#) | [kagi](#) | [movavg](#) | [candle](#) | [highlow](#) | [pointfig](#) | [priceandvol](#) | [renko](#) | [volarea](#)

llov

Lowest low

Syntax

```
llv = llov(data)
```

```
llv = llov(data, nperiods, dim)
```

```
llvts = llov(tsobj, nperiods)
```

```
llvts = llov(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

<code>data</code>	Data series matrix.
<code>nperiods</code>	(Optional) Number of periods. Default = 14.
<code>dim</code>	Dimension.
<code>tsobj</code>	Financial time series object.
<code>ParameterName</code>	The valid parameter name is: <ul style="list-style-type: none">• <code>LowName</code>: low prices series name
<code>ParameterValue</code>	The parameter value is a string that represents the valid parameter name.

Description

`llv = llov(data)` generates a vector of lowest low values for the past 14 periods from the matrix `data`.

`llv = llov(data, nperiods, dim)` generates a vector of lowest low values for the past `nperiods` periods. `dim` indicates the direction in which the lowest low is to be searched. If you input `[]` for `nperiods`, the default is 14.

`llvts = l1ow(tsobj, nperiods)` generates a vector of lowest low values from `tsobj`, a financial time series object. `tsobj` must include at least the series `LOW`. The output `llvts` is a financial time series object with the same dates as `tsobj` and data series named `LowestLow`. If `nperiods` is specified, `l1ow` generates a financial time series object of lowest low values for the past `nperiods` periods.

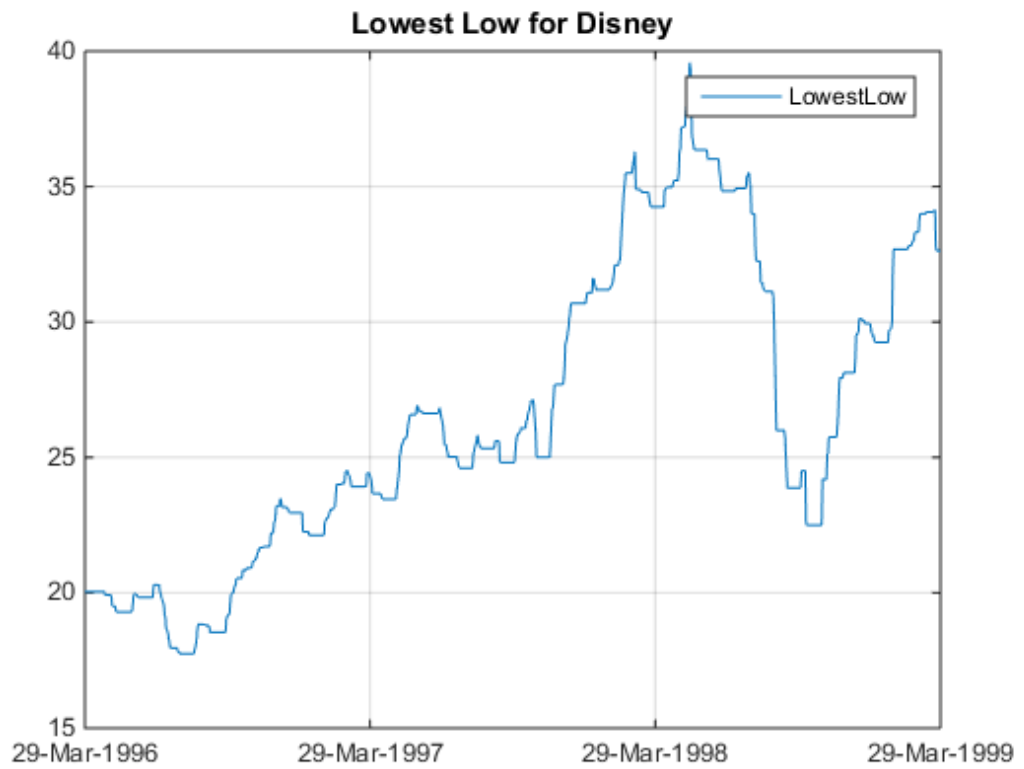
`llvts = l1ow(tsobj, nperiods, ParameterName, ParameterValue)` specifies the name for the required data series when it is different from the default name. The parameter value is a string that represents the valid parameter name.

Examples

Compute the Lowest Low Price

This example shows how to compute the lowest low price for Disney stock and plot the results.

```
load disney.mat
dis_LLow = l1ow(dis);
plot(dis_LLow)
title('Lowest Low for Disney')
```



See Also
hhigh

log

Natural logarithm

Syntax

```
newfts = log(tsobj)
```

Description

`newfts = log(tsobj)` calculates the natural logarithm (log base e) of the data series in a financial time series object `tsobj`. It returns another time series object `newfts` containing the natural logarithms.

See Also

`exp` | `log2` | `log10`

log10

Common logarithm

Syntax

```
newfts = log10(tsobj)
```

Description

`newfts = log10(tsobj)` calculates the common logarithm (base 10) of all the data in the data series of the financial time series object `tsobj` and returns the result in the object `newfts`.

See Also

`exp` | `log` | `log2`

log2

Base 2 logarithm

Syntax

```
newfts = log2(tsobj)
```

Description

`newfts = log2(tsobj)` calculates the base 2 logarithm of the data series in a financial time series object `tsobj`. It returns another time series object `newfts` containing the logarithms.

See Also

`exp` | `log` | `log10`

lpm

Compute sample lower partial moments of data

Syntax

`lpm(Data)`

`lpm(Data, MAR)`

`lpm(Data, MAR, Order)`

`Moment = lpm(Data, MAR, Order)`

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES observations of NUMSERIES asset returns.
MAR	(Optional) Scalar minimum acceptable return (default MAR = 0). This is a cutoff level of return such that all returns above MAR contribute nothing to the lower partial moment.
Order	(Optional) Either a scalar or a NUMORDERS-vector of nonnegative integer moment orders. If no order specified, default Order = 0, which is the shortfall probability. Although this function will work for noninteger orders and, in some cases, for negative orders, this falls outside customary usage.

Description

Given NUMSERIES assets with NUMSAMPLES returns in a NUMSAMPLES-by-NUMSERIES matrix `Data`, a scalar minimum acceptable return `MAR`, and one or more nonnegative moment orders in a NUMORDERS vector `Order`, `lpm` computes lower partial moments relative to `MAR` for each asset in a NUMORDERS x NUMSERIES matrix `Moment`.

The output `Moment` is a NUMORDERS x NUMSERIES matrix of lower partial moments with NUMORDERS `Orders` and NUMSERIES series, that is, each row contains lower partial moments for a given order.

Note: To compute upper partial moments, just reverse the signs of both Data and MAR (do not reverse the sign of the output). This function computes sample lower partial moments from data. To compute expected lower partial moments for multivariate normal asset returns with a specified mean and covariance, use `elpm`. With `lpm`, you can compute various investment ratios such as Omega ratio, Sortino ratio, and Upside Potential ratio, where:

- $\text{Omega} = \text{lpm}(-\text{Data}, -\text{MAR}, 1) / \text{lpm}(\text{Data}, \text{MAR}, 1)$
 - $\text{Sortino} = (\text{mean}(\text{Data}) - \text{MAR}) / \text{sqrt}(\text{lpm}(\text{Data}, \text{MAR}, 2))$
 - $\text{Upside} = \text{lpm}(-\text{Data}, -\text{MAR}, 1) / \text{sqrt}(\text{lpm}(\text{Data}, \text{MAR}, 2))$
-

Examples

See “Sample Lower Partial Moments” on page 7-12.

References

Vijay S. Bawa, "Safety-First, Stochastic Dominance, and Optimal Portfolio Choice," *Journal of Financial and Quantitative Analysis*, Vol. 13, No. 2, June 1978, pp. 255-271.

W. V. Harlow, "Asset Allocation in a Downside-Risk Framework," *Financial Analysts Journal*, Vol. 47, No. 5, September/October 1991, pp. 28-40.

W. V. Harlow and K. S. Rao, "Asset Pricing in a Generalized Mean-Lower Partial Moment Framework: Theory and Evidence," *Journal of Financial and Quantitative Analysis*, Vol. 24, No. 3, September 1989, pp. 285-311.

Frank A. Sortino and Robert van der Meer, "Downside Risk," *Journal of Portfolio Management*, Vol. 17, No. 5, Spring 1991, pp. 27-31.

See Also

`elpm`

lweekdate

Date of last occurrence of weekday in month

Syntax

```
LastDate = lweekdate(Weekday, Year, Month, NextDay)
```

Arguments

Weekday	Weekday whose date you seek. Enter as an integer from 1 through 7:	
	1	Sunday
	2	Monday
	3	Tuesday
	4	Wednesday
	5	Thursday
	6	Friday
	7	Saturday
Year	Year. Enter as a four-digit integer.	
Month	Month. Enter as an integer from 1 through 12.	
NextDay	(Optional) Weekday that must occur after Weekday in the same week. Enter as an integer from 0 through 7, where 0 = ignore (default) and 1 through 7 are the same as for Weekday .	

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if **Year** is a 1-by-n vector of integers, then **Month** must be a 1-by-n vector of integers or a single integer. **LastDate** is then a 1-by-n vector of date numbers.

Description

`LastDate = lweekdate(Weekday, Year, Month, NextDay)` returns the serial date number for the last occurrence of `Weekday` in the given year and month and in a week that also contains `NextDay`.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

Example 1. To find the last Monday in June 2001

```
LastDate = lweekdate(2, 2001, 6); datestr>LastDate)
```

```
ans =
```

```
25-Jun-2001
```

Example 2. To find the last Monday in a week that also contains a Friday in June 2001

```
LastDate = lweekdate(2, 2001, 6, 6); datestr>LastDate)
```

```
ans =
```

```
25-Jun-2001
```

Example 3. To find the last Monday in May for 2001, 2002, and 2003

```
Year = [2001:2003];
```

```
LastDate = lweekdate(2, Year, 5)
```

```
LastDate =
```

```
730999    731363    731727
```

```
datestr>LastDate)
```

```
ans =
```

```
28-May-2001
```

```
27-May-2002
```

```
26-May-2003
```

See Also

eomdate | lbusdate | nweekdate

m2xdate

MATLAB serial date number to Excel serial date number

Syntax

DateNum = m2xdate(MATLABDateNumber, Convention)

Arguments

MATLABDateNumber	A vector or scalar of MATLAB serial date numbers.
Convention	<p>(Optional) Excel date system. A vector or scalar. When <code>Convention = 0</code> (default), the Excel 1900 date system is in effect. When <code>Convention = 1</code>, the Excel 1904 date system is used.</p> <p>In the Excel 1900 date system, the Excel serial date number 1 corresponds to January 1, 1900 A.D. In the Excel 1904 date system, date number 0 is January 1, 1904 A.D.</p> <p>Due to a software limitation in Excel software, the year 1900 is considered a leap year. As a result, all DATEVALUE's reported by Excel software between Jan. 1, 1900 and Feb. 28, 1900 (inclusive) differs from the values reported by 1. For example:</p> <ul style="list-style-type: none"> • In Excel software, Jan. 1, 1900 = 1 • In MATLAB, Jan. 1, 1900 – 693960 (for 1900 date system) = 2 <pre>datenum('Jan 1, 1900') - 693960 ans = 2</pre>

Vector arguments must have consistent dimensions.

Description

`DateNum = m2xdate(MATLABDateNumber, Convention)` converts MATLAB serial date numbers to Excel serial date numbers. MATLAB date numbers start with 1 = January 1, 0000 A.D., hence there is a difference of 693960 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with Spreadsheet Link™ EX software.

Examples

Convert MATLAB Serial Date Numbers Using 1900 Date System

This example shows how to convert MATLAB serial date numbers using the 1900 date system. Given MATLAB date numbers for Christmas 2001 through 2004, convert them to Excel date numbers in the 1900 system.

```
DateNum = datenum(2001:2004, 12, 25);  
ExDate = m2xdate(DateNum)
```

```
ExDate =
```

```
    37250    37615    37980    38346
```

Convert MATLAB Serial Date Numbers Using 1904 Date System

This example shows how to convert MATLAB serial date numbers using the 1904 date system. Given MATLAB date numbers for Christmas 2001 through 2004, convert them to Excel date numbers in the 1904 system.

```
DateNum = datenum(2001:2004, 12, 25);  
ExDate = m2xdate(DateNum, 1)
```

```
ExDate =
```

```
    35788    36153    36518    36884
```

See Also

`datenum` | `datestr` | `x2mdate`

macd

Moving Average Convergence/Divergence (MACD)

Syntax

```
[macdvec, nineperma] = macd(data)
```

```
[macdvec, nineperma] = macd(data, dim)
```

```
macdts = macd(tsobj, series_name)
```

Arguments

<code>data</code>	Data matrix
<code>dim</code>	Dimension. Default = 1 (column orientation).
<code>tsobj</code>	Financial time series object
<code>series_name</code>	Data series name

Description

`[macdvec, nineperma] = macd(data)` calculates the Moving Average Convergence/Divergence (MACD) line, `macdvec`, from the data matrix, `data`, as well as the nine-period exponential moving average, `nineperma`, from the MACD line.

When the two lines are plotted, they can give you an indication of whether to buy or sell a stock, when an overbought or oversold condition is occurring, and when the end of a trend might occur.

The MACD is calculated by subtracting the 26-period (7.5%) exponential moving average from the 12-period (15%) moving average. The 9-day (20%) exponential moving average of the MACD line is used as the *signal* line. For example, when the MACD and the 20% moving average line have just crossed and the MACD line falls below the other line, it is time to sell.

`[macdvec, nineperma] = macd(data, dim)` lets you specify the orientation direction for the input. If the input data is a matrix, you need to indicate whether each row is a set of observations (`dim = 2`) or each column is a set of observations (`dim = 1`, the default).

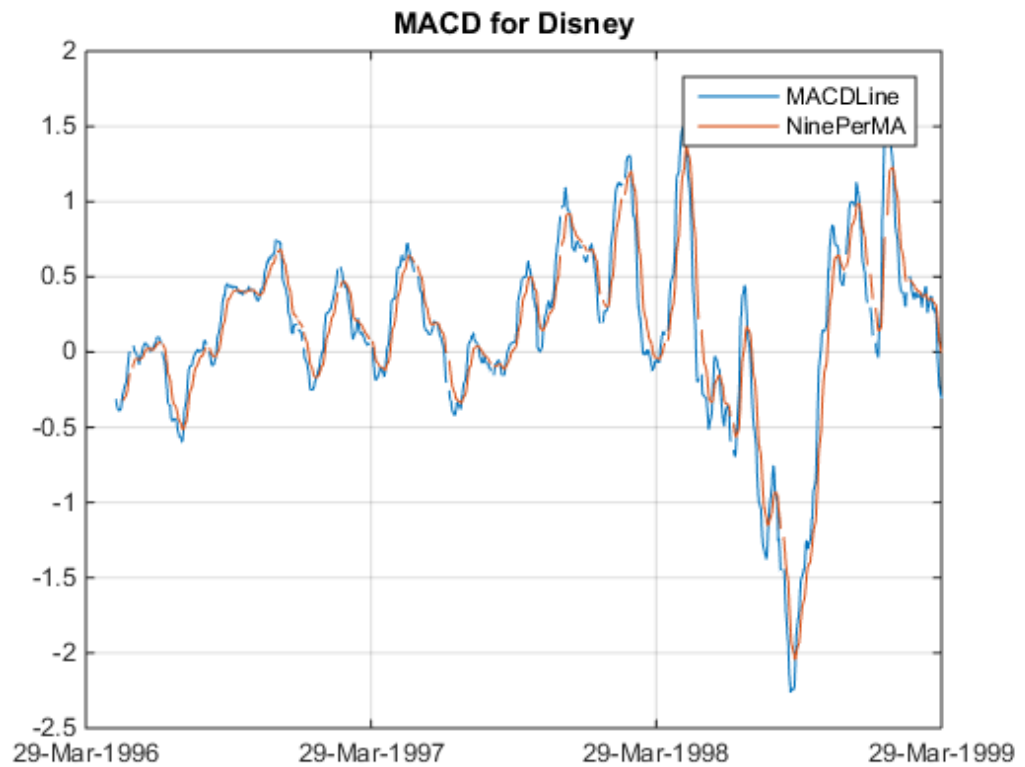
`macdts = macd(tsoobj, series_name)` calculates the MACD line from the financial time series `tsoobj`, as well as the nine-period exponential moving average from the MACD line. The MACD is calculated for the closing price series in `tsoobj`, presumed to have been named `Close`. The result is stored in the financial time series object `macdts`. The `macdts` object has the same dates as the input object `tsoobj` and contains only two series, named `MACDLine` and `NinePerMA`. The first series contains the values representing the MACD line and the second is the nine-period exponential moving average of the MACD line.

Examples

Compute the Moving Average Convergence/Divergence (MACD)

This example shows how to compute the MACD for Disney stock and plot the results.

```
load disney.mat
dis_CloseMACD = macd(dis);
dis_OpenMACD = macd(dis, 'OPEN');
plot(dis_CloseMACD);
plot(dis_OpenMACD);
title('MACD for Disney')
```

References

Achelis, Steven B., *Technical Analysis From A To Z*, Second Printing, McGraw-Hill, 1995, pp. 166-168.

See Also

adline | willad

max

Maximum value

Syntax

```
tymax = max(tsobj)
```

Description

`tymax = max(tsobj)` finds the maximum value in each data series in the financial time series object `tsobj` and returns it in a structure `tymax`. The `tymax` structure contains field name(s) identical to the data series name(s).

Note `tymax` returns only the values and does not return the dates associated with the values. The maximum values are not necessarily from the same date.

See Also

`min`

maxdrawdown

Compute maximum drawdown for one or more price series

Syntax

MaxDD = maxdrawdown(Data)

MaxDD = maxdrawdown(Data, Format)

[MaxDD, MaxDDIndex] = maxdrawdown(Data, Format)

Arguments

Data	T-by-N matrix with T samples of N total return price series (also known as total equity).
Format	(Optional) MATLAB string indicating format of data. Possible values are:
	'return' (default): Maximum drawdown in terms of maximum percentage drop from a peak.
	'arithmetic': Maximum drawdown of an arithmetic Brownian motion with drift (differences of data from peak to trough) using the equation $dX(t) = \mu dt + \sigma dW(t).$
	'geometric': Maximum drawdown of a geometric Brownian motion with drift (differences of log of data from peak to trough) using the equation $dS(t) = \mu_0 S(t) dt + \sigma_0 S(t) dW(t)$

Description

MaxDD = maxdrawdown(Data, *Format*) computes maximum drawdown for each series in an N-vector MaxDD and identifies start and end indexes of maximum drawdown periods for each series in a 2 x N matrix MaxDDIndex.

To summarize the outputs of maxdrawdown:

- MaxDD is a 1-by-N vector with maximum drawdown for each of N time series.
- MaxDDIndex is a 2-by-N vector of start and end indexes for each maximum drawdown period for each total equity time series, where the first row contains the start indexes and the second row contains the end indexes of each maximum drawdown period.

Notes

- Drawdown is the percentage drop in total returns from the start to the end of a period. If the total equity time series is increasing over an entire period, drawdown is 0. Otherwise, it is a positive number. Maximum drawdown is an ex-ante proxy for downside risk that computes the largest drawdown over all intervals of time that can be formed within a specified interval of time.
 - Maximum drawdown is sensitive to quantization error.
-

Examples

See “Maximum Drawdown” on page 7-15.

References

Christian S. Pederson and Ted Rudholm-Alfvén, "Selecting a Risk-Adjusted Shareholder Performance Measure," *Journal of Asset Management*, Vol. 4, No. 3, 2003, pp. 152-172.

See Also

emaxdrawdown

mean

Arithmetic average

Syntax

```
tsmean = mean(tsobj)
```

Description

`tsmean = mean(tsobj)` computes the arithmetic mean of all data in all series in `tsobj` and returns it in a structure `tsmean`. The `tsmean` structure contains field name(s) identical to the data series name(s).

See Also

`peravg` | `tsmovavg`

medprice

Median price

Syntax

```
mprc = medprice(highp, lowp)
```

```
mprc = medprice([highp lowp])
```

```
mprcts = medprice(tsobj)
```

```
mprcts = medprice(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

highp	High price (vector)
lowp	Low price (vector)
tsobj	Financial time series object
ParameterName	Valid parameter names are: <ul style="list-style-type: none">• HighName: high prices series name• LowName: low prices series name
ParameterValue	Parameter values are the strings that represent the valid parameter names.

Description

`mprc = medprice(highp, lowp)` calculates the median prices `mprc` from the high (`highp`) and low (`lowp`) prices. The median price is the average of the high and low price for each period.

`mprc = medprice([highp lowp])` accepts a two-column matrix as the input rather than two individual vectors. The columns of the matrix represent the high and low prices, in that order.

`mprcts = medprice(tsobj)` calculates the median prices of a financial time series object `tsobj`. The object must minimally contain the series `High` and `Low`. The median price is the average of the high and low price each period. `mprcts` is a financial time series object with the same dates as `tsobj` and the data series `MedPrice`.

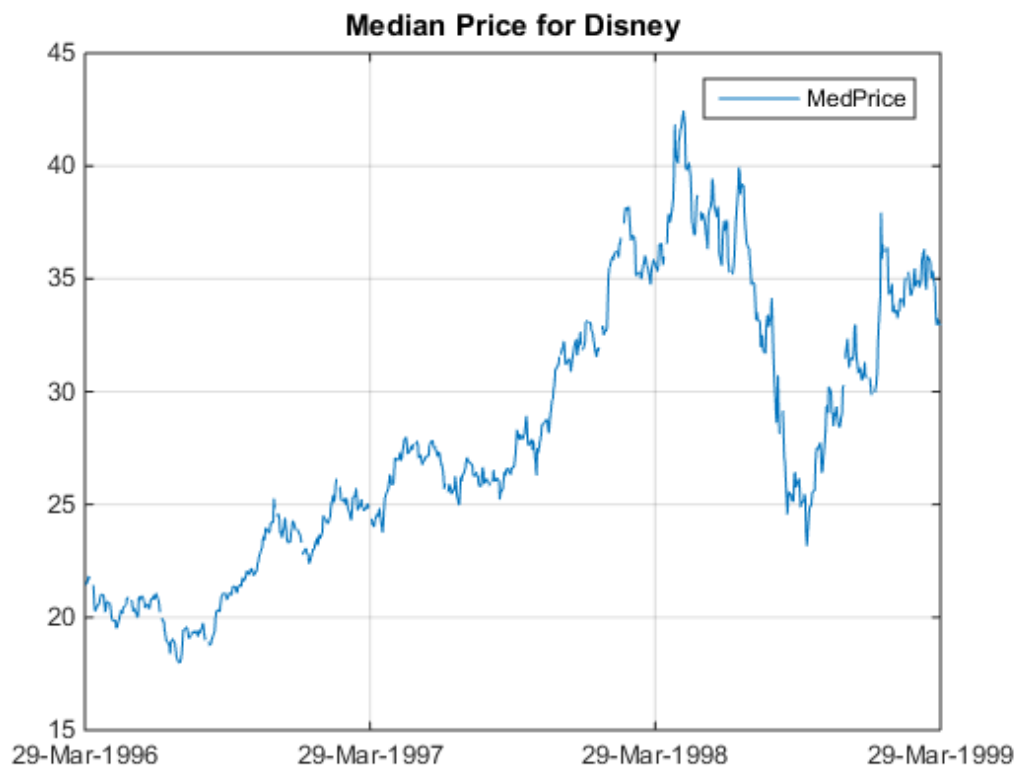
`mprcts = medprice(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Median Price

This example shows how to compute the median price for Disney stock and plot the results.

```
load disney.mat
dis_MedPrice = medprice(dis);
plot(dis_MedPrice)
title('Median Price for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 177 -178.

See Also

wclose | typprice

merge

Merge multiple financial time series objects

Syntax

```
newfts = merge(fts1, fts2)
```

```
newfts = merge(fts1, fts2, ..., ftsx)
```

```
newfts = merge(fts1, fts2, ..., ftsx, 'PARAM1', VALUE1, 'PARAM2',  
VALUE2, ...)
```

Arguments

fts1, fts2, ...	Comma-separated list of financial time series objects to merge. Note Multiple Financial Time Series objects can be merged at once. The merged objects must appear in a comma separated list before the optional inputs. The order of the inputs is significant.
'DataSetMethod'	(Optional) Merge <i>method</i> . Valid merge values are: 'union': (Default) Returns the combined values of all merged objects. 'intersection': Returns the values common to all merged objects. RefObj: Maps all values to a reference time contained in a Financial Time Series object (RefObj) or vector of date numbers.
'DataSetMethod'	(Optional) Merge <i>method</i> . Valid merge values are: 'closest': (Default) Returns data based on the order of the inputs. However, the first missing data point (NaN value) of

	a date will be replaced by the closest non-NaN data point that appears on the same date of subsequent merged objects. 'order': Returns data based strictly on the order of the inputs.
'SortColumns'	(Optional) Sorts columns. Valid merge values are: True/1: Sorts the columns based on the headers (series names). The headers are sorted in alphabetical order. False/0: Columns are not sorted.

Description

`newfts = merge(fts1, fts2, ..., ftsx, 'PARAM1', VALUE1, 'PARAM2', VALUE2', ...)` merges multiple financial time series objects. The optional parameter and value pair argument specifies the values contained in the output financial time series object `ftsout`.

Examples

Create Three Financial Time Series Objects and Merge into a Single Object

Create three financial time series objects and merged into a new time series object `t123`.

```

dates = {'jan-01-2001'; 'jan-02-2001'; 'jan-03-2001'; ...
        'jan-04-2001'; 'jan-06-2001'};
data = [1; 1; 1; 1; 1];
t1 = fints(dates, data);

dates = {'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001';
        'jan-05-2001'};
data = [2; 2; 2; 2];
t2 = fints(dates, data);

dates = {'jan-03-2001'; 'jan-04-2001'; 'jan-05-2001';
        'jan-06-2001'};
data = [3; 3; 3; 3];
t3 = fints(dates, data);

t123 = merge(t1, t2, t3)

```

```
t123 =
  desc:  ||  ||
  freq:  Unknown (0)

  'dates: (6)'   'series1: (6)'
  '01-Jan-2001' [           1]
  '02-Jan-2001' [           1]
  '03-Jan-2001' [           1]
  '04-Jan-2001' [           1]
  '05-Jan-2001' [           2]
  '06-Jan-2001' [           1]
```

If you change the order of input time series, the output may contain different data when duplicate dates exist. Here, for example, is the result of using the same three time series defined above but with the order changed.

```
merge(t3, t2, t1)
```

```
ans =
  desc:  ||  ||
  freq:  Unknown (0)

  'dates: (6)'   'series1: (6)'
  '01-Jan-2001' [           1]
  '02-Jan-2001' [           2]
  '03-Jan-2001' [           3]
  '04-Jan-2001' [           3]
  '05-Jan-2001' [           3]
  '06-Jan-2001' [           3]
```

`t123` contains all 1's except on `'05-Jan-2001'` because `t1` appears first in the list of inputs and takes precedence. The same logic can be applied to `t321`. By changing the order of inputs, you can overwrite old financial time series data with new data by placing the new time series ahead of the old one in the list of inputs to the `merge` function.

Merge Financial Time Series Objects with Different Headers (Series Names)

Merge time series objects with different headers into a new time series object `t45`.

```
dates = {'jan-01-2001'; 'jan-02-2001'; 'jan-03-2001'; ...
'jan-04-2001'; 'jan-06-2001'};
data = [1; 1; 1; 1; 1];
t4 = fints(dates, data, 'ts4');

dates = {'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001'; 'jan-05-2001'};
data = [2; 2; 2; 2];
t5 = fints(dates, data, 'ts5');

t45 = merge(t4, t5)

t45 =

desc:  ||
freq: Unknown (0)

'dates: (6)'   'ts4: (6)'   'ts5: (6)'
'01-Jan-2001' [         1]   [         NaN]
'02-Jan-2001' [         1]   [         2]
'03-Jan-2001' [         1]   [         2]
'04-Jan-2001' [         1]   [         2]
'05-Jan-2001' [         NaN]  [         2]
'06-Jan-2001' [         1]   [         NaN]
```

Merge Two Financial Index Series and Keep Intersecting Dates

Merge two index series into the final merged object (t12) and keep the intersecting dates.

```
dates = {'jan-01-2001'; 'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001'; 'jan-06-2001'};
data = [1; 1; 1; 1; 1];
t1 = fints(dates, data, 'A')

dates = {'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001'; 'jan-05-2001'};
data = [2; 2; 2; 2];
t2 = fints(dates, data, 'B')

t12 = merge(t1, t2, 'DateSetMethod', 'Intersection')

t1 =

desc: (none)
freq: Unknown (0)
```

```
'dates: (5)'      'A: (5)'
'01-Jan-2001'    [      1]
'02-Jan-2001'    [      1]
'03-Jan-2001'    [      1]
'04-Jan-2001'    [      1]
'06-Jan-2001'    [      1]
```

t2 =

```
desc: (none)
freq: Unknown (0)

'dates: (4)'      'B: (4)'
'02-Jan-2001'    [      2]
'03-Jan-2001'    [      2]
'04-Jan-2001'    [      2]
'05-Jan-2001'    [      2]
```

t12 =

```
desc: ||
freq: Unknown (0)

'dates: (3)'      'A: (3)'      'B: (3)'
'02-Jan-2001'    [      1]    [      2]
'03-Jan-2001'    [      1]    [      2]
'04-Jan-2001'    [      1]    [      2]
```

See Also

horzcat | vertcat

min

Minimum value

Syntax

```
tmin = min(tobj)
```

Description

`tmin = min(tobj)` finds the minimum value in each data series in the financial time series object `tobj` and returns it in the structure `tmin`. The `tmin` structure contains field name(s) identical to the data series name(s).

Note `tmin` returns only the values and does not return the dates associated with the values. The minimum values are not necessarily from the same date.

See Also

`max`

minus

Financial time series subtraction

Syntax

```
newfts = tsoj_1 - tsoj_2
```

```
newfts = tsoj - array
```

```
newfts = array - tsoj
```

Arguments

<code>tsoj_1, tsoj_2</code>	A pair of financial time series objects .
<code>array</code>	A scalar value or array with the number of rows equal to the number of dates in <code>tsoj</code> and the number of columns equal to the number of data series in <code>tsoj</code> .

Description

`minus` is an element-by-element subtraction of the components.

`newfts = tsoj_1 - tsoj_2` subtracts financial time series objects. If an object is to be subtracted from another object, both objects must have the same dates and data series names, although the order need not be the same. The order of the data series, when one financial time series object is subtracted from another, follows the order of the first object.

`newfts = tsoj - array` subtracts an array element by element from a financial time series object.

`newfts = array - tsoj` subtracts a financial time series object element by element from an array.

See Also

`rdivide` | `plus` | `times`

minute

Minute of date or time

Syntax

```
Minute = minute(Date)
```

Description

`Minute = minute(Date)` returns the minute given a serial date number or a date string.

Examples

```
Minute = minute(731204.5591223380)
```

or

```
Minute = minute('19-dec-2001, 13:25:08.17')
```

returns

```
Minute =
```

```
25
```

See Also

`datevec` | `hour` | `second`

mirr

Modified internal rate of return

Syntax

Return = mirr(CashFlow, FinRate, Reinvest)

Arguments

CashFlow	Vector of cash flows. The first entry is the initial investment.
FinRate	Finance rate for negative cash flow values. Enter as a decimal fraction.
Reinvest	Reinvestment rate for positive cash flow values, as a decimal fraction.

Description

Return = mirr(CashFlow, FinRate, Reinvest) calculates the modified internal rate of return for a series of periodic cash flows. This function calculates only positive rates of return; for nonpositive rates of return, Return = 0.

Examples

This cash flow represents the yearly income from an initial investment of \$100,000. The finance rate is 9% and the reinvestment rate is 12%.

Year 1	\$20,000
Year 2	(\$10,000)
Year 3	\$30,000
Year 4	\$38,000
Year 5	\$50,000

To calculate the modified internal rate of return on the investment

```
Return = mirr([-100000 20000 -10000 30000 38000 50000], 0.09,...  
0.12)
```

returns

```
Return =  
0.0832 (8.32%)
```

References

Brealey and Myers, *Principles of Corporate Finance*, Chapter 5

See Also

annurate | effrr | irr | nomrr | pvvar | xirr

month

Month of date

Syntax

```
[MonthNum, MonthString] = month(Date)
```

```
[MonthNum, MonthString] = month(Date, F)
```

Description

`[MonthNum, MonthString] = month(Date)` returns the month in numeric and string form given a serial date number or a date string.

`[MonthNum, MonthString] = month(Date, F)` returns the day of the of the month, given a serial date number or date string, in a specified date format.

Examples

```
[MonthNum, MonthString] = month(730368)
```

or

```
[MonthNum, MonthString] = month('05-Sep-1999')
```

returns

```
MonthNum =
```

```
9
```

```
MonthString =
```

```
Sep
```

You can also use the `F` argument to designate a country-specific date format:

```
[MonthNum, MonthString] = month('1999/05/09', 'yyyy/dd/mm')
```

returns

hmiMonthNum =

9

MonthString =

Sep

See Also

datevec | day | year

months

Number of whole months between dates

Syntax

```
MyMonths = months(StartDate, EndDate, EndMonthFlag)
```

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.
EndMonthFlag	(Optional) end-of-month flag. If StartDate and EndDate are end-of-month dates and EndDate has fewer days than StartDate, EndMonthFlag = 1 (default) treats EndDate as the end of a whole month, while EndMonthFlag = 0 does not.

Description

`MyMonths = months(StartDate, EndDate, EndMonthFlag)` returns the number of whole months between `StartDate` and `EndDate`. If `EndDate` is earlier than `StartDate`, `MyMonths` is negative. Enter dates as serial date numbers or date strings.

Any input argument can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n -row character array of date strings, then `EndDate` must be an n -row character array of date strings or a single date. `MyMonths` is then an n -by-1 vector of numbers.

Examples

```
MyMonths = months('may 31 2000', 'jun 30 2000', 1)
```

```
MyMonths =  
    1  
  
MyMonths = months('may 31 2000', 'jun 30 2000', 0)  
  
MyMonths =  
    0  
  
Dates = ['mar 31 2002'; 'apr 30 2002'; 'may 31 2002'];  
MyMonths = months(Dates, 'jun 30 2002')  
  
MyMonths =  
    3  
    2  
    1
```

See Also
yearfrac

movavg

Leading and lagging moving averages chart

Syntax

movavg(Asset, Lead, Lag, Alpha)

[Short, Long] = movavg(Asset, Lead, Lag, Alpha)

Arguments

Asset	Security data, a vector of time-series prices.
Lead	Number of samples to use in leading average calculation. A positive integer. Lead must be less than or equal to Lag .
Lag	Number of samples to use in the lagging average calculation. A positive integer.
Alpha	<p>(Optional) Control parameter that determines the type of moving averages. 0 = simple moving average (default), 0.5 = square root weighted moving average, 1 = linear moving average, 2 = square weighted moving average, and so on. To calculate the exponential moving average, set Alpha = 'e'.</p> <hr/> <p>Note: When Alpha = 'e', the value of the moving average depends on all previous data points (due to the iterative calculation). In this case, the Lead and Lag parameters are used to calculate the weighting factor for their respective averages (which is different from the number of samples).</p>

Description

movavg(Asset, Lead, Lag, Alpha) plots leading and lagging moving averages.

[Short, Long] = movavg(Asset, Lead, Lag, Alpha) returns the leading Short and lagging Long moving average data without plotting it.

Examples

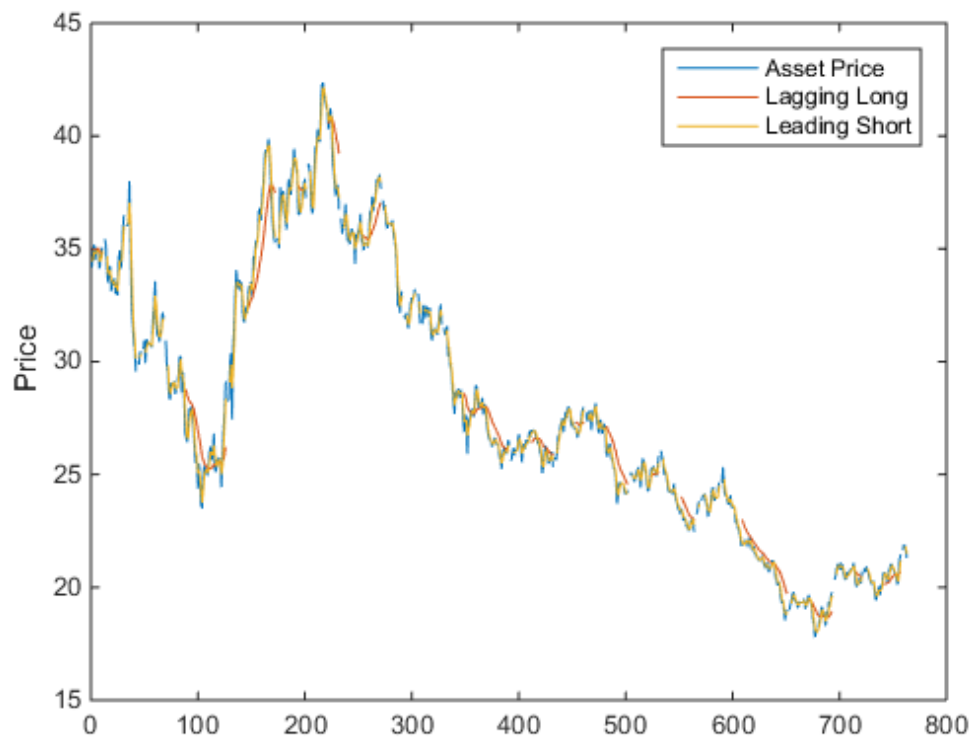
Compute the Moving Average for DIS Closing Prices

Load the DIS closing prices using `disney.mat`.

```
load disney.mat
```

Use `movavg` to plot the leading and lagging moving averages for DIS.

```
movavg(dis_CLOSE,3,20,1);ylabel('Price')  
legend('Asset Price','Lagging Long','Leading Short')
```

See Also

[bolling](#) | [candle](#) | [dateaxis](#) | [highlow](#) | [pointfig](#)

mrdivide

Financial time series matrix division

Syntax

```
newfts = tsobj_1 / tsobj_2
```

```
newfts = tsobj / array
```

```
newfts = array / tsobj
```

Arguments

<code>tsobj_1, tsobj_2</code>	A pair of financial time series objects.
<code>array</code>	A scalar value or array with number of rows equal to the number of dates in <code>tsobj</code> and number of columns equal to the number of data series in <code>tsobj</code> .

Description

The `mrdivide` method divides element by element the components of one financial time series object by the components of the other. You can also divide the whole object by an array or divide a financial time series object into an array.

If an object is to be divided by another object, both objects must have the same dates and data series names, although the order need not be the same. The order of the data series, when an object is divided by another object, follows the order of the first object.

`newfts = tsobj_1 / tsobj_2` divides financial time series objects element by element.

`newfts = tsobj / array` divides a financial time series object element by element by an array.

`newfts = array / tsobj` divides an array element by element by a financial time series object.

For financial time series objects, the `mrdivide` operation is identical to the `rdivide` operation.

See Also

`minus` | `plus` | `rdivide` | `times`

mtimes

Financial time series matrix multiplication

Syntax

```
newfts = tsoj_1 * tsoj_2
```

```
newfts = tsoj * array
```

```
newfts = array * tsoj
```

Arguments

<code>tsoj_1, tsoj_2</code>	A pair of financial time series objects.
<code>array</code>	A scalar value or array with number of rows equal to the number of dates in <code>tsoj</code> and number of columns equal to the number of data series in <code>tsoj</code> .

Description

The `mtimes` method multiplies element by element the components of one financial time series object by the components of the other. You can also multiply the entire object by an array.

If an object is to be multiplied by another object, both objects must have the same dates and data series names, although the order need not be the same. The order of the data series, when an object is multiplied by another object, follows the order of the first object.

`newfts = tsoj_1 * tsoj_2` multiplies financial time series objects element by element.

`newfts = tsoj * array` multiplies a financial time series object element by element by an array.

`newfts = array * tsoj` and `newfts = array / tsoj` multiplies an array element by element by a financial time series object.

For financial time series objects, the `mtimes` operation is identical to the `times` operation.

See Also

`minus` | `mrdivide` | `plus` | `times`

mvnrfish

Fisher information matrix for multivariate normal or least-squares regression

Syntax

```
Fisher = mvnrfish(Data, Design, Covariance, MatrixFormat,
CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. If a data sample has missing values, represented as NaNs, the sample is ignored.
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Covariance	NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the residuals of the regression.
MatrixFormat	<p>(Optional) String that identifies parameters to be included in the Fisher information matrix:</p> <ul style="list-style-type: none"> • <code>full</code> - Default format. Compute the full Fisher information matrix for both model and covariance parameter estimates.

	<ul style="list-style-type: none"> • paramonly - Compute only components of the Fisher information matrix associated with the model parameter estimates.
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • 'full' - Default method. The covariance matrix is a full matrix. • 'diagonal' - The covariance matrix is a diagonal matrix.

Description

`Fisher = mvnrfish(Data, Design, Covariance, MatrixFormat, CovarFormat)` computes a Fisher information matrix based on current maximum likelihood or least-squares parameter estimates.

`Fisher` is a `TOTALPARAMS`-by-`TOTALPARAMS` Fisher information matrix. The size of `TOTALPARAMS` depends on `MatrixFormat` and on current parameter estimates. If `MatrixFormat = 'full'`,

$$\text{TOTALPARAMS} = \text{NUMPARAMS} + \text{NUMSERIES} * (\text{NUMSERIES} + 1) / 2$$

If `MatrixFormat = 'paramonly'`,

$$\text{TOTALPARAMS} = \text{NUMPARAMS}$$

Note `mvnrfish` operates slowly if you calculate the full Fisher information matrix.

Examples

See “Multivariate Normal Linear Regression” on page 9-2.

See Also

`mvnrstd` | `mvnrml`

mvnrml

Multivariate normal regression (ignore missing data)

Syntax

```
[Parameters, Covariance, Resid, Info] = mvnrml(Data, Design,
MaxIterations, TolParam, TolObj, Covar0, CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. If a data sample has missing values, represented as NaNs, the sample is ignored. (Use <code>ecmmvnrml</code> to handle missing data.)
Design	<p>Matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
MaxIterations	(Optional) Maximum number of iterations for the estimation algorithm. Default value is 100.
TolParam	(Optional) Convergence tolerance for estimation algorithm based on changes in model parameter estimates. Default value is <code>sqrt(eps)</code> which is about 1.0e-8 for double

	precision. The convergence test for changes in model parameters is
	$\ Param_k - Param_{k-1}\ < TolParam \times (1 + \ Param_k\)$
	where Param represents the output Parameters , and iteration $k = 2, 3, \dots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both $TolParam \leq 0$ and $TolObj \leq 0$, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.
TolObj	<p>(Optional) Convergence tolerance for estimation algorithm based on changes in the objective function. Default value is eps # 3/4 which is about 1.0e-12 for double precision. The convergence test for changes in the objective function is</p> $ Obj_k - Obj_{k-1} < TolObj \times (1 + Obj_k)$ <p>for iteration $k = 2, 3, \dots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both $TolParam \leq 0$ and $TolObj \leq 0$, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.</p>
Covar0	(Optional) NUMSERIES -by- NUMSERIES matrix that contains a user-supplied initial or known estimate for the covariance matrix of the regression residuals.
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • 'full' - Default method. Compute the full covariance matrix. • 'diagonal' - Force the covariance matrix to be a diagonal matrix.

Description

[Parameters, Covariance, Resid, Info] = `mvnrml`(Data, Design, MaxIterations, TolParam, TolObj, Covar0, CovarFormat) estimates a multivariate normal regression model without missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

for samples $k = 1, \dots, \text{NUMSAMPLES}$.

`mvnrml` estimates a `NUMPARAMS`-by-1 column vector of model parameters called `Parameters`, and a `NUMSERIES`-by-`NUMSERIES` matrix of covariance parameters called `Covariance`.

`mvnrml`(Data, Design) with no output arguments plots the log-likelihood function for each iteration of the algorithm.

To summarize the outputs of `mvnrml`:

- `Parameters` is a `NUMPARAMS`-by-1 column vector of estimates for the parameters of the regression model.
- `Covariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the regression model's residuals.
- `Resid` is a `NUMSAMPLES`-by-`NUMSERIES` matrix of residuals from the regression. For any row with missing values in `Data`, the corresponding row of residuals is represented as all NaN missing values, since this routine ignores rows with NaN values.

Another output, `Info`, is a structure that contains additional information from the regression. The structure has these fields:

- `Info.Obj` – A variable-extent column vector, with no more than `MaxIterations` elements, that contains each value of the objective function at each iteration of the estimation algorithm. The last value in this vector, `Obj(end)`, is the terminal estimate of the objective function. If you do maximum likelihood estimation, the objective function is the log-likelihood function.
- `Info.PrevParameters` – `NUMPARAMS`-by-1 column vector of estimates for the model parameters from the iteration just before the terminal iteration.
- `Info.PrevCovariance` – `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance parameters from the iteration just before the terminal iteration.

Notes

mvnrmlc does not accept an initial parameter vector, because the parameters are estimated directly from the first iteration onward.

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES = 1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES > 1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

These points concern how `Design` handles missing data:

- Although `Design` should not have NaN values, ignored samples due to NaN values in `Data` are also ignored in the corresponding `Design` array.
- If `Design` is a 1-by-1 cell array, which has a single `Design` matrix for each sample, no NaN values are permitted in the array. A model with this structure must have `NUMSERIES ≥ NUMPARAMS` with `rank(Design{1}) = NUMPARAMS`.
- Two functions for handling missing data, `ecmmvnrmlc` and `ecmlsrmlc`, are stricter about the presence of NaN values in `Design`.

Use the estimates in the optional output structure `Info` for diagnostic purposes.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

References

Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

Xiao-Li Meng and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

See Also

ecmmvnrmlc | mvnrstd | mvnrobj

mvnrobj

Log-likelihood function for multivariate normal regression without missing data

Syntax

Objective = mvnrobj(Data, Design, Parameters, Covariance, CovarFormat)

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. If a data sample has missing values, represented as NaNs, the sample is ignored. (Use <code>ecmmvnrml</code> to handle missing data.)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Parameters	NUMPARAMS-by-1 column vector of estimates for the parameters of the regression model.
Covariance	NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the residuals of the regression.
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • <code>'full'</code> - Default method. The covariance matrix is a full matrix.

- 'diagonal' - The covariance matrix is a diagonal matrix.

Description

`Objective = mvnrojb(Data, Design, Parameters, Covariance, CovarFormat)` computes the log-likelihood function based on current maximum likelihood parameter estimates without missing data. `Objective` is a scalar that contains the log-likelihood function.

Notes

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES = 1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES > 1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Although `Design` should not have NaN values, ignored samples due to NaN values in `Data` are also ignored in the corresponding `Design` array.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

See Also

`ecmmvnrml` | `ecmmvnrojb` | `mvnrml`

mvnrstd

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters, StdCovariance] = mvnrstd(Data, Design, Covariance, CovarFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. If a data sample has missing values, represented as NaNs, the sample is ignored. (Use <code>ecmmvnrml</code> to handle missing data.)
Design	<p>A matrix or a cell array that handles two model structures:</p> <ul style="list-style-type: none"> • If <code>NUMSERIES = 1</code>, <code>Design</code> is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series. • If <code>NUMSERIES ≥ 1</code>, <code>Design</code> is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values. <p>If <code>Design</code> has a single cell, it is assumed to have the same <code>Design</code> matrix for each sample. If <code>Design</code> has more than one cell, each cell contains a <code>Design</code> matrix for each sample.</p>
Covariance	NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the regression residuals.
CovarFormat	<p>(Optional) String that specifies the format for the covariance matrix. The choices are:</p> <ul style="list-style-type: none"> • <code>'full'</code> - Default method. The covariance matrix is a full matrix. • <code>'diagonal'</code> - The covariance matrix is a diagonal matrix.

Description

[StdParameters, StdCovariance] = mvnrstd(Data, Design, Covariance, CovarFormat) evaluates standard errors for a multivariate normal regression model without missing data. The model has the form

$$Data_k \sim N(\text{Design}_k \times \text{Parameters}, \text{Covariance})$$

for samples $k = 1, \dots, \text{NUMSAMPLES}$.

mvnrstd computes two outputs:

- StdParameters is a NUMPARAMS-by-1 column vector of standard errors for each element of Parameters, the vector of estimated model parameters.
- StdCovariance is a NUMSERIES-by-NUMSERIES matrix of standard errors for each element of Covariance, the matrix of estimated covariance parameters.

Note mvnrstd operates slowly when you calculate the standard errors associated with the covariance matrix Covariance.

Notes

You can configure Design as a matrix if NUMSERIES = 1 or as a cell array if NUMSERIES ≥ 1.

- If Design is a cell array and NUMSERIES = 1, each cell contains a NUMPARAMS row vector.
- If Design is a cell array and NUMSERIES > 1, each cell contains a NUMSERIES-by-NUMPARAMS matrix.

Examples

See “Multivariate Normal Regression” on page 9-16, “Least-Squares Regression” on page 9-16, “Covariance-Weighted Least Squares” on page 9-17, “Feasible Generalized Least Squares” on page 9-18, and “Seemingly Unrelated Regression” on page 9-19.

References

Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

See Also

ecmmvnrmlc | ecmmvnrstd | mvnrmlc

nancov

Covariance ignoring NaNs

Syntax

```
c = nancov(X)
```

```
c = nancov(..., 'pairwise')
```

Arguments

X	Financial times series object.
Y	Financial times series object.

Description

`nancov` for financial times series objects is based on the Statistics Toolbox function `nancov`. See `nancov` in the Statistics Toolbox documentation.

`c = nancov(X)`, if `X` is a financial time series object with one series and returns the sample variance of the values in `X`, treating NaNs as missing values. For a financial time series object containing more than one series, where each row is an observation and each series a variable, `nancov(X)` is the covariance matrix computing using rows of `X` that do not contain any NaN values. `nancov(X, Y)`, where `X` and `Y` are financial time series objects with the same number of elements, is equivalent to `nancov([X(:) Y(:)])`.

`nancov(X)` or `nancov(X, Y)` normalizes by $(N-1)$ if $N > 1$, where N is the number of observations after removing missing values. This makes `nancov` the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For $N = 1$, `cov` normalizes by N .

`nancov(X, 1)` or `nancov(X, Y, 1)` normalizes by N and produces the second moment matrix of the observations about their mean. `nancov(X, Y, 0)` is the same as `nancov(X, Y)`, and `nancov(X, 0)` is the same as `nancov(X)`.

`c = nancov(..., 'pairwise')` computes $c(i, j)$ using rows with no NaN values in columns i or j . The result may not be a positive definite matrix. `c = nancov(..., 'complete')` is the default, and it omits rows with any NaN values, even if they are not in column i or j . The mean is removed from each column before calculating the result.

Examples

To generate random data having nonzero covariance between column 4 and the other columns:

```
x = randn(30, 4);           % uncorrelated data
x(:, 4) = sum(x, 2);       % introduce correlation
x(2, 3) = NaN;            % introduce one missing value
f = fints('today:today+29', x); % create a fints object using x
c = nancov(f);           % compute sample covariance
```

`c =`

```
    1.6898    -0.0005     0.3612     1.9143
   -0.0005     1.0833    -0.5513     0.6059
    0.3612    -0.5513     1.0369     0.7570
    1.9143     0.6059     0.7570     4.4895
```

See Also

`cov` | `nanvar` | `var`

nanmax

Maximum ignoring NaNs

Syntax

```
m = nanmax(X)
```

```
[m,ndx] = nanmax(X)
```

```
m = nanmax(X,Y)
```

```
[m,ndx] = nanmax(X,[],DIM)
```

Arguments

X	Financial times series object.
Y	Financial times series object or scalar.
DIM	Dimension of X.

Description

`nanmax` for financial times series objects is based on the Statistics Toolbox function `nanmax`. See `nanmax` in the Statistics Toolbox documentation.

`m = nanmax(X)` returns the maximum of a financial time series object `X` with NaNs treated as missing. `m` is the largest non-NaN element in `X`.

`[m,ndx] = nanmax(X)` returns the indices of the maximum values in `X`. If the values along the first nonsingleton dimension contain multiple maximal elements, the index of the first one is returned.

`m = nanmax(X,Y)` returns an array the same size as `X` and `Y` with the largest elements taken from `X` or `Y`. Only `Y` can be a scalar double.

`[m,ndx] = nanmax(X,[],DIM)` operates along the dimension `DIM`.

Examples

To compute nanmax for the following dates:

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
f.series3(2) = nan;
```

```
[nmax, maxidx] = nanmax(f)
```

```
nmax =
     4     5     6
```

```
maxidx =
     3     2     1
```

See Also

[max](#) | [nanmean](#) | [nanmin](#) | [nanstd](#) | [nanvar](#) | [nanmedian](#)

nanmean

Mean ignoring NaNs

Syntax

```
m = nanmean(X)
```

```
m = nanmean(X,DIM)
```

Arguments

X	Financial times series object.
DIM	Dimension along which the operation is conducted.

Description

`nanmean` for financial times series objects is based on the Statistics Toolbox function `nanmean`. See `nanmean` in the Statistics Toolbox documentation.

`m = nanmean(X)` returns the sample mean of a financial time series object `X`, treating NaNs as missing values. `m` is a row vector containing the mean value of the non-NaN elements in each series.

`m = nanmean(X,DIM)` takes the mean along dimension `DIM` of `X`.

Examples

To compute `nanmean` for the following dates:

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'};  
f = fints(dates, magic(3));  
f.series1(1) = nan;  
f.series2(3) = nan;  
f.series3(2) = nan;
```

```
nmean = nanmean(f)
```

```
nmean =
```

```
    3.5000    3.0000    4.0000
```

See Also

[mean](#) | [nanmax](#) | [nanmin](#) | [nanstd](#) | [nansum](#) | [nanvar](#)

nanmedian

Median ignoring NaNs

Syntax

```
m = nanmedian(X)
```

```
m = nanmedian(X,DIM)
```

Arguments

X	Financial times series object.
DIM	Dimension along which the operation is conducted.

Description

`nanmedian` for financial times series objects is based on the Statistics Toolbox function `nanmedian`. See `nanmedian` in the Statistics Toolbox documentation.

`m = nanmedian(X)` returns the sample median of a financial time series object `X`, treating NaNs as missing values. `m` is a row vector containing the median value of non-NaN elements in each column.

`m = nanmedian(X,DIM)` takes the median along the dimension `DIM` of `X`.

Examples

To compute `nanmedian` for the following dates:

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'; '04-Jan-2007'};  
f = fints(dates, magic(4));  
f.series1(1) = nan;  
f.series2(2) = nan;  
f.series3([1 3]) = nan;  
  
nmedian = nanmedian(f)
```



```
nanmedian =  
    5.0000    7.0000   12.5000   10.0000
```

See Also

[mean](#) | [nanmax](#) | [nanmin](#) | [nanstd](#) | [nansum](#) | [nanvar](#)

nanmin

Minimum ignoring NaNs

Syntax

```
m = nanmin(X)
```

```
[m,ndx] = nanmin(X)
```

```
m = nanmin(X,Y)
```

```
[m,ndx] = nanmin(X,[],DIM)
```

Arguments

X	Financial times series object.
Y	Financial times series object or scalar.
DIM	Dimension along which the operation is conducted.

Description

`nanmin` for financial times series objects is based on the Statistics Toolbox function `nanmin`. See `nanmin` in the Statistics Toolbox documentation.

`m = nanmin(X)` returns the minimum of a financial time series object `X` with NaNs treated as missing. `m` is the smallest non-NaN element in `X`.

`[m,ndx] = nanmin(X)` returns the indices of the minimum values in `X`. If the values along the first nonsingleton dimension contain multiple elements, the index of the first one is returned.

`m = nanmin(X,Y)` returns an array the same size as `X` and `Y` with the smallest elements taken from `X` or `Y`. Only `Y` can be a scalar double.

`[m,ndx] = nanmin(X, [], DIM)` operates along the dimension `DIM`.

Examples

To compute `nanmin` for the following dates:

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
f.series3(2) = nan;
```

```
[nmin, minidx] = nanmin(f)
```

```
nmin =
     3     1     2
minidx =
     2     1     3
```

See Also

[mean](#) | [nanmax](#) | [nanstd](#) | [nanvar](#)

nanstd

Standard deviation ignoring NaNs

Syntax

`y = nanstd(X)`

`y = nanstd(X,1)`

`y = nanstd(X,FLAG,DIM)`

Arguments

X	Financial times series object.
FLAG	Normalization flag.
DIM	Dimension along which the operation is conducted.

Description

`nanstd` for financial times series objects is based on the Statistics Toolbox function `nanstd`. See `nanstd` in the Statistics Toolbox documentation.

`y = nanstd(X)` returns the sample standard deviation of the values in a financial time series object `X`, treating NaNs as missing values. `y` is the standard deviation of the non-NaN elements of `X`.

`nanstd` normalizes `y` by $(N - 1)$, where N is the sample size. This is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples and data are missing at random.

`y = nanstd(X, 1)` normalizes by N and produces the square root of the second moment of the sample about its mean. `nanstd(X, 0)` is the same as `nanstd(X)`.

`y = nanstd(X, flag, dim)` takes the standard deviation along the dimension `dim` of `X`. Set the value of `flag` to 0 to normalize the result by $n - 1$; set the value of `flag` to 1 to normalize the result by n .

Examples

To compute `nanstd` for the following dates:

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'};  
f = fints(dates, magic(3));  
f.series1(1) = nan;  
f.series2(3) = nan;  
f.series3(2) = nan;
```

```
nstd = nanstd(f)
```

```
nstd =
```

```
    0.71    2.83    2.83
```

See Also

`nanmax` | `nanmean` | `nanmin` | `nanvar` | `std` | `nanmedian`

nansum

Sum ignoring NaNs

Syntax

```
y = nansum(X)
```

```
y = nansum(X,DIM)
```

Arguments

X	Financial time series object.
DIM	Dimension along which the operation is conducted.

Description

`nansum` for financial times series objects is based on the Statistics Toolbox function `nansum`. See `nansum` in the Statistics Toolbox documentation.

`y = nansum(X)` returns the sum of a financial time series object `X`, treating NaNs as missing values. `y` is the sum of the non-NaN elements in `X`.

`y = nansum(X,DIM)` takes the sum along dimension `DIM` of `X`.

Examples

To compute `nansum` for the following dates:

```
dates = {'01-Jan-2007'; '02-Jan-2007'; '03-Jan-2007'};  
f = fints(dates, magic(3));  
f.series1(1) = nan;  
f.series2(3) = nan;  
f.series3(2) = nan;
```

```
nsum = nansum(f)
```

```
nsum =  
    7    6    8
```

See Also

nanmax | nanmean | nanmin | nanstd | nanvar | nanmedian

nanvar

Variance ignoring NaNs

Syntax

`y = nanvar(X)`

`y = nanvar(X,1)`

`y = nanvar(X,W)`

`y = nanvar(X,W,DIM)`

Arguments

X	Financial times series object.
W	Weight vector.
DIM	Dimension along which the operation is conducted.

Description

`nanvar` for financial times series objects is based on the Statistics Toolbox function `nanvar`. See `nanvar` in the Statistics Toolbox documentation.

`y = nanvar(X)` returns the sample variance of the values in a financial time series object `X`, treating NaNs as missing values. `y` is the variance of the non-NaN elements of each series in `X`.

`nanvar` normalizes `y` by $N - 1$ if $N > 1$, where N is the sample size of the non-NaN elements. This is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples, and data are missing at random. For $N = 1$, `y` is normalized by N .

`y = nanvar(X,1)` normalizes by N and produces the second moment of the sample about its mean. `nanvar(X,0)` is the same as `nanvar(X)`.

`y = nanvar(X,W)` computes the variance using the weight vector `W`. The length of `W` must equal the length of the dimension over which `nanvar` operates, and its non-NaN elements must be nonnegative. Elements of `X` corresponding to NaN elements of `W` are ignored.

`y = nanvar(X,W,DIM)` takes the variance along dimension `DIM` of `X`.

Examples

To compute `nanvar`:

```
f = fints((today:today+1)', [4 -2 1; 9 5 7])
f.series1(1) = nan;
f.series3(2) = nan;
```

```
nvar = nanvar(f)
```

```
nvar =
      0    24.5000      0
```

See Also

`nanmax` | `nanmean` | `nanmin` | `nanstd` | `var` | `nanmedian`

negvalidx

Negative volume index

Syntax

```
nvi = negvalidx(closep, tvolume, initnvi)
```

```
nvi = negvalidx([closep tvolume], initnvi)
```

```
nvits = negvalidx(tsobj)
```

```
vits = negvalidx(tsobj, initnvi, ParameterName, ParameterValue,  
...)
```

Arguments

closep	Closing price (vector).
tvolume	Volume traded (vector).
initnvi	(Optional) Initial value for negative volume index (Default = 100).
tsobj	Financial time series object.
ParameterName	Valid parameter names are: <ul style="list-style-type: none">• CloseName: closing prices series name• VolumeName: volume traded series name
ParameterValue	Parameter values are the strings that represent the valid parameter names.

Description

`nvi = negvalidx(closep, tvolume, initnvi)` calculates the negative volume index from a set of stock closing prices (`closep`) and volume traded (`tvolume`) data. `nvi`

is a vector representing the negative volume index. If `initnvi` is specified, `negvolidx` uses that value instead of the default (100).

`nvi = negvolidx([closep tvolume], initnvi)` accepts a two-column matrix, the first column representing the closing prices (`closep`) and the second representing the volume traded (`tvolume`). If `initnvi` is specified, `negvolidx` uses that value instead of the default (100).

`nvits = negvolidx(tsoobj)` calculates the negative volume index from the financial time series object `tsoobj`. The object must contain, at least, the series `Close` and `Volume`. The `nvits` output is a financial time series object with dates similar to `tsoobj` and a data series named `NVI`. The initial value for the negative volume index is arbitrarily set to 100.

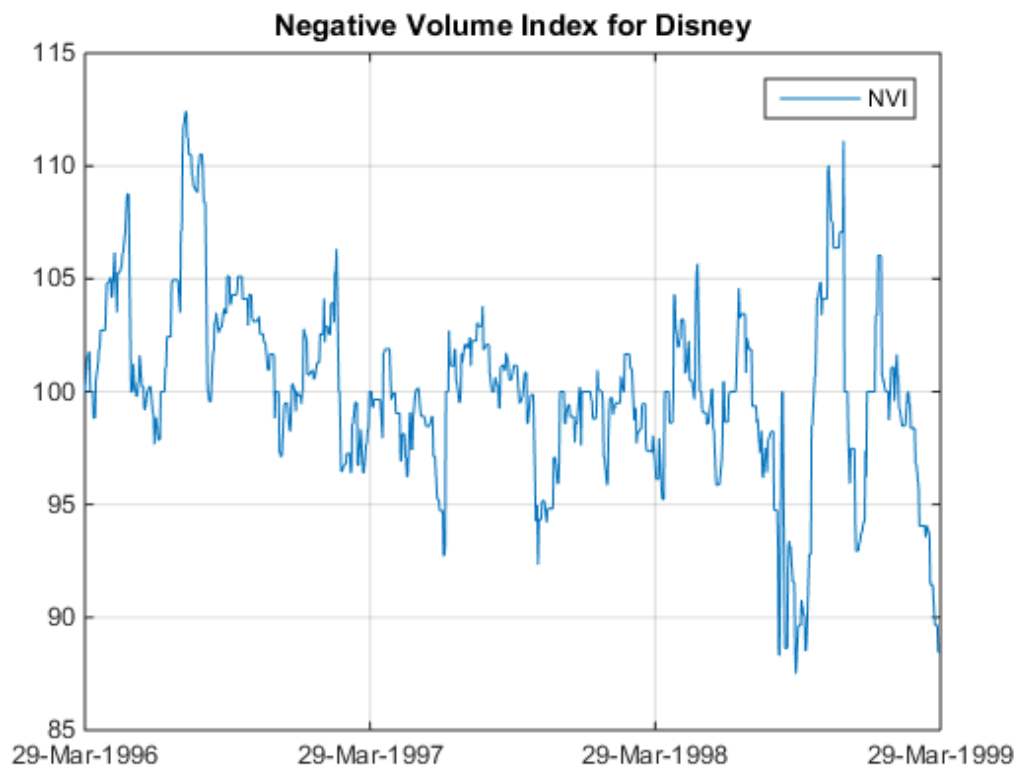
`nvits = negvolidx(tsoobj, initnvi, ParameterName, ParameterValue, ...)` accepts parameter name/ parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Negative Volume Index

This example shows how to compute the negative volume index for Disney stock and plot the results.

```
load disney.mat
dis_NegVol = negvolidx(dis);
plot(dis_NegVol)
title('Negative Volume Index for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 193 - 194.

See Also

onbalvol | posvolidx

nomrr

Nominal rate of return

Syntax

Return = nomrr(Rate, NumPeriods)

Arguments

Rate	Effective annual percentage rate. Enter as a decimal fraction.
NumPeriods	Number of compounding periods per year, an integer.

Description

Return = nomrr(Rate, NumPeriods) calculates the nominal rate of return.

Examples

Calculate the Nominal Rate of Return

This example shows how to calculate the nominal rate of return based on an effective annual percentage rate of 9.38% compounded monthly.

Return = nomrr(0.0938, 12)

Return =

0.0900

See Also

effrr | irr | mirr | taxedrr | xirr

nweekdate

Date of specific occurrence of weekday in month

Syntax

Date = nweekdate(n, Weekday, Year, Month, Same)

Arguments

n	Nth occurrence of the weekday in a month. Enter as integer from 1 through 5.	
Weekday	Weekday whose date you seek. Enter as integer from 1 through 7.	
	1	Sunday
	2	Monday
	3	Tuesday
	4	Wednesday
	5	Thursday
	6	Friday
	7	Saturday
Year	Year. Enter as a four-digit integer.	
Month	Month. Enter as an integer from 1 through 12.	
Same	(Optional) Weekday that must occur in the same week with Weekday. Enter as an integer from 0 through 7, where 0 = ignore (default) and 1 through 7 are as for Weekday.	

Description

Date = nweekdate(n, Weekday, Year, Month, Same) returns the serial date number for the specific occurrence of the weekday in the given year and month, and in a week that also contains the weekday Same.

If n is larger than the last occurrence of `Weekday`, `Date = 0`.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by- n vector of integers, then `Month` must be a 1-by- n vector of integers or a single integer. `Date` is then a 1-by- n vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

To find the first Thursday in May 2001

```
Date = nweekdate(1, 5, 2001, 5); datestr(Date)
```

```
ans =
```

```
03-May-2001
```

To find the first Thursday in a week that also contains a Wednesday in May 2001

```
Date = nweekdate(2, 5, 2001, 5, 4); datestr(Date)
```

```
ans =
```

```
10-May-2001
```

To find the third Monday in February for 2001, 2002, and 2003

```
Year = [2001:2003];
```

```
Date = nweekdate(3, 2, Year, 2)
```

```
Date =
```

```
730901    731265    731629
```

```
datestr(Date)
```

```
ans =
```

```
19-Feb-2001
```

```
18-Feb-2002
```

```
17-Feb-2003
```

See Also

`fbusdate` | `lbusdate` | `lweekdate`

nyseclosures

New York Stock Exchange closures from 1885 to 2050

Syntax

```
[Closures, SatTransition] = nyseclosures(StartDate, EndDate,  
WorkWeekFormat)
```

Description

`[Closures, SatTransition] = nyseclosures(StartDate, EndDate, WorkWeekFormat)` returns a vector of serial date numbers corresponding to market closures between `StartDate` and `EndDate`, inclusive. If you do not specify `StartDate` and `EndDate`,

`Closures` output contains all known or anticipated closures from January 1, 1885 to December 31, 2050. By default, `WorkWeekFormat` argument uses the 'Implicit' value.

`SatTransition` output is the date of transition for the New York Stock Exchange from a 6-day workweek to a 5-day workweek. The date for this transition is September 29, 1952 and this date returns the serial date number 713226.

Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures from 1885 to 1952 are based on a 6-day workweek. `nyseclosures` contains all holiday and special non-trading days for the New York Stock Exchange from 1885 through 2050 based on a six-day work week (always closed on Sundays). Use `WorkWeekFormat` to modify the list of dates.

Input Arguments

StartDate

First date of a specified date range that is a date string or a serial date number.

Default: January 1, 1885 (start of the default date range)

EndDate

Last date of a specified date range that is a date string or a serial date number. If specified, the **EndDate** must be a date after the **StartDate**.

Default: December 31, 2050 (end of the default date range)

WorkWeekFormat

Specifies method to handle the workweek. The default is 'Implicit'. This function accepts the first letter for each method as input and is not case sensitive. Acceptable values are:

- 'Modern' — 5-day workweek with all Saturday trading days removed
- 'Implicit' — 6-day workweek until 1952 and 5-day week afterward (no need to exclude Saturdays)
- 'Archaic' — 6-day workweek throughout and Saturdays treated as closures after 1952

Output Arguments

Closures

A vector of serial date numbers corresponding to market closures between the dates **StartDate** and **EndDate**, inclusive

SatTransition

The date of transition for the New York Stock Exchange from a 6-day workweek to a 5-day workweek.

Definitions

holidays is based on a modern 5-day workweek and contains all holidays and special nontrading days for the New York Stock Exchange from January 1, 1885 to December 31, 2050. Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures for the period from 1885 to 2050 should include Saturday trading days. To capture these dates, use the function **nyseclosures**. The results from

holidays and nyseclosures are identical if the WorkWeekFormat in nyseclosures is 'modern'.

Examples

Find the NYSE closures for 1899:

```
datestr(nyseclosures('1-jan-1899','31-dec-1899'),'dd-mmm-yyyy ddd')
```

This returns:

```
ans =
```

```
02-Jan-1899 Mon
11-Feb-1899 Sat
13-Feb-1899 Mon
22-Feb-1899 Wed
31-Mar-1899 Fri
29-May-1899 Mon
30-May-1899 Tue
03-Jul-1899 Mon
04-Jul-1899 Tue
04-Sep-1899 Mon
29-Sep-1899 Fri
30-Sep-1899 Sat
07-Nov-1899 Tue
25-Nov-1899 Sat
30-Nov-1899 Thu
25-Dec-1899 Mon
```

Find the NYSE closure dates using the 'Archaic' value for WorkWeekFormat:

```
datestr(nyseclosures('1-sep-1952','31-oct-1952','a'),1)
```

This returns:

```
ans =
```

```
01-Sep-1952
06-Sep-1952
13-Sep-1952
20-Sep-1952
27-Sep-1952
```

04-Oct-1952
11-Oct-1952
13-Oct-1952
18-Oct-1952
25-Oct-1952

The exchange was closed on Saturdays for much of 1952 before the official transition to a 5-day workweek.

See Also

[busdate](#) | [createholidays](#) | [fbusdate](#) | [isbusday](#) | [lbusdate](#) | [holidays](#)

onbalvol

On-Balance Volume (OBV)

Syntax

```
obv = onbalvol(closep, tvolume)
```

```
obv = onbalvol([closep tvolume])
```

```
obvts = onbalvol(tsoobj)
```

```
obvts = onbalvol(tsoobj, ParameterName, ParameterValue, ...)
```

Arguments

closep	Closing price (vector)
tvolume	Volume traded
tsoobj	Financial time series object

Description

`obv = onbalvol(closep, tvolume)` calculates the On-Balance Volume (OBV) from the stock closing price (`closep`) and volume traded (`tvolume`) data.

`obv = onbalvol([closep tvolume])` accepts a two-column matrix representing the closing price (`closep`) and volume traded (`tvolume`), in that order.

`obvts = onbalvol(tsoobj)` calculates the OBV from the stock data in the financial time series object `tsoobj`. The object must minimally contain series names `Close` and `Volume`. The `obvts` output is a financial time series object with the same dates as `tsoobj` and a series named `OnBalVol`.

`obvts = onbalvol(tsoobj, ParameterName, ParameterValue, ...)` accepts parameter name/ parameter value pairs as input. These pairs specify the name(s) for the

required data series if it is different from the expected default name(s). Valid parameter names are

- **CloseName**: closing prices series name
- **VolumeName**: volume traded series name

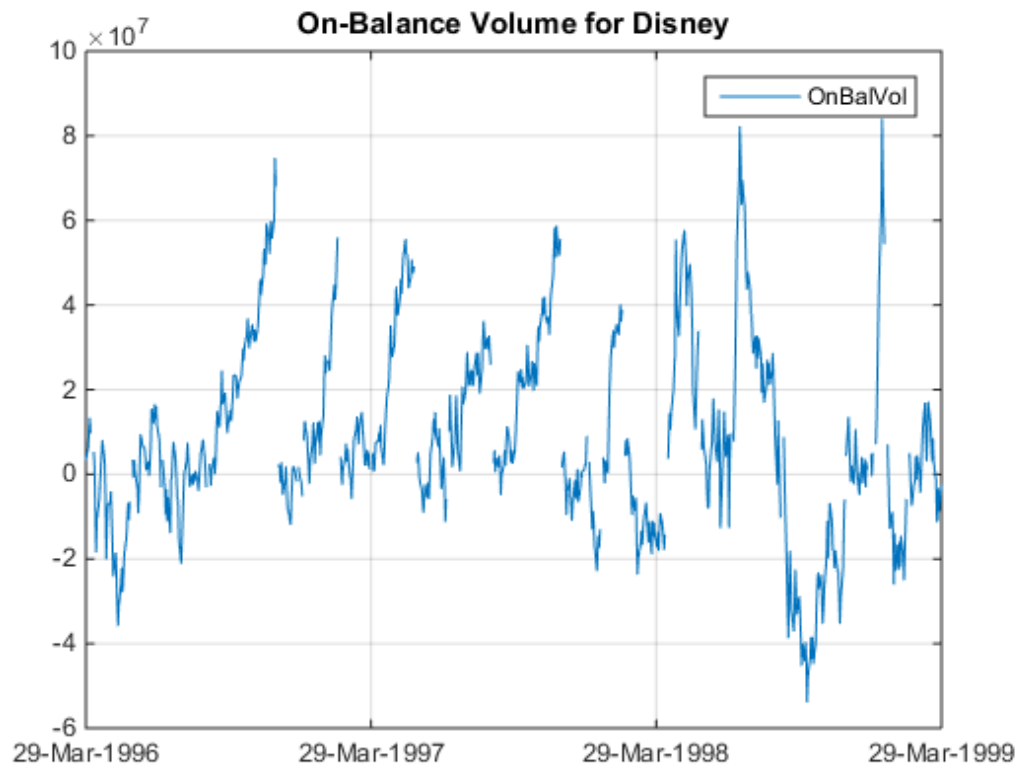
Parameter values are the strings that represent the valid parameter names.

Examples

Calculate the On-Balance Volume (OBV)

This example shows how to calculate the OBV for Disney stock and plot the results.

```
load disney.mat
dis_OnBalVol = onbalvol(dis);
plot(dis_OnBalVol)
title('On-Balance Volume for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 207 - 209.

See Also

negvolidx

opprofit

Option profit

Syntax

```
Profit = opprofit(AssetPrice, Strike, Cost, PosFlag, OptType)
```

Arguments

AssetPrice	Asset price.
Strike	Strike or exercise price.
Cost	Cost of the option.
PosFlag	Option position. 0 = long, 1 = short.
OptType	Option type. 0 = call option, 1 = put option.

Description

`Profit = opprofit(AssetPrice, Strike, Cost, PosFlag, OptType)` returns the profit of an option.

Examples

Calculate the Profit of an Option

This example shows how to return the profit of an option. For example, consider buying (going long on) a call option with a strike price of \$90 on an underlying asset with a current price of \$100 for a cost of \$4.

```
Profit = opprofit(100, 90, 4, 0, 0)
```

```
Profit =
```


6

See Also

binprice | blsprice

payadv

Periodic payment given number of advance payments

Syntax

Payment = payadv(Rate, NumPeriods, PresentValue, FutureValue, Advance)

Arguments

Rate	Lending or borrowing rate per period. Enter as a decimal fraction. Must be greater than or equal to 0.
NumPeriods	Number of periods in the life of the instrument.
PresentValue	Present value of the instrument.
FutureValue	Future value or target value to be attained after NumPeriods periods.
Advance	Number of advance payments. If the payments are made at the beginning of the period, add 1 to Advance.

Description

Payment = payadv(Rate, NumPeriods, PresentValue, FutureValue, Advance) returns the periodic payment given a number of advance payments.

Examples

Compute the Periodic Payment

This example shows how to compute the periodic payment, given a number of advance payments. For example, the present value of a loan is \$1000.00 and it will be paid in full in 12 months. The annual interest rate is 10% and three payments are made at closing time.

```
Payment = payadv(0.1/12, 12, 1000, 0, 3)
```

```
Payment =
```

```
85.9389
```

See Also

[amortize](#) | [payodd](#) | [payper](#)

payodd

Payment of loan or annuity with odd first period

Syntax

`Payment = payodd(Rate, NumPeriods, PresentValue, FutureValue, Days)`

Arguments

<code>rate</code>	Interest rate per period. Enter as a decimal fraction.
<code>NumPeriods</code>	Number of periods in the life of the instrument.
<code>PresentValue</code>	Present value of the instrument.
<code>FutureValue</code>	Future value or target value to be attained after <code>NumPeriods</code> periods.
<code>Days</code>	Actual number of days until the first payment is made.

Description

`Payment = payodd(Rate, NumPeriods, PresentValue, FutureValue, Days)`
returns the payment for a loan or annuity with an odd first period.

Examples

Compute the Payment for a Loan or Annuity With an Odd First Period

This example shows how to return the payment for a loan or annuity with an odd first period. For example, consider a two-year loan for \$4000 that has an annual interest rate of 11% and the first payment will be made in 36 days.

`Payment = payodd(0.11/12, 24, 4000, 0, 36)`

`Payment =`

186.7731

See Also

amortize | payadv | payper

payper

Periodic payment of loan or annuity

Syntax

Payment = payper(Rate, NumPeriods, PresentValue, FutureValue, Due)

Arguments

Rate	Interest rate per period. Enter as a decimal fraction.
NumPeriods	Number of payment periods in the life of the instrument.
PresentValue	Present value of the instrument.
FutureValue	(Optional) Future value or target value to be attained after NumPeriods periods. Default = 0.
Due	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

Payment = payper(Rate, NumPeriods, PresentValue, FutureValue, Due)
returns the periodic payment of a loan or annuity.

Examples

Compute the Periodic Payment of a Loan or Annuity

This example shows how to find the monthly payment for a three-year loan of \$9000 with an annual interest rate of 11.75%.

Payment = payper(0.1175/12, 36, 9000, 0, 0)

Payment =

297.8553

See Also

amortize | fvfix | payadv | payodd | pvfix

payuni

Uniform payment equal to varying cash flow

Syntax

```
Series = payuni(CashFlow, Rate)
```

Arguments

CashFlow	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
Rate	Periodic interest rate. Enter as a decimal fraction.

Description

`Series = payuni(CashFlow, Rate)` returns the uniform series value of a varying cash flow.

Examples

This cash flow represents the yearly income from an initial investment of \$10,000. The annual interest rate is 8%.

Year 1	\$2000
Year 2	\$1500
Year 3	\$3000
Year 4	\$3800
Year 5	\$5000

To calculate the uniform series value

```
Series = payuni([-10000 2000 1500 3000 3800 5000], 0.08)
```


returns

Series =

429.63

See Also

fvfix | fvvar | irr | pvfix | pvvar

pcalims

Linear inequalities for individual asset allocation

As an alternative to `pcalims`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Syntax

```
[A,b] = pcalims(AssetMin, AssetMax, NumAssets)
```

Arguments

<code>AssetMin</code>	Scalar or <code>NASSETS</code> vector of minimum allocations in each asset. NaN indicates no constraint.
<code>AssetMax</code>	Scalar or <code>NASSETS</code> vector of maximum allocations in each asset. NaN indicates no constraint.
<code>NumAssets</code>	(Optional) Number of assets. Default = length of <code>AssetMin</code> or <code>AssetMax</code> .

Description

`[A,b] = pcalims(AssetMin, AssetMax, NumAssets)` specifies the lower and upper bounds of portfolio allocations in each of `NumAssets` available asset investments.

`A` is a matrix and `b` is a vector such that $A * \text{PortWts}' \leq b$, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

If `pcalims` is called with fewer than two output arguments, the function returns `A` concatenated with `b` `[A,b]`.

Examples

Set the minimum weight in every asset to 0 (no short-selling), and set the maximum weight of IBM stock to 0.5 and CSCO to 0.8, while letting the maximum weight in INTC float.

Asset	IBM	INTC	CSCO
Min. Wt.	0	0	0
Max. Wt.	0.5		0.8

```
AssetMin = 0
AssetMax = [0.5 NaN 0.8]
[A,b] = pcalims(AssetMin, AssetMax)
```

```
A =
     1     0     0
     0     0     1
    -1     0     0
     0    -1     0
     0     0    -1
```

```
b =
    0.5000
    0.8000
         0
         0
         0
```

Portfolio weights of 50% in IBM and 50% in INTC satisfy the constraints.

Set the minimum weight in every asset to 0 and the maximum weight to 1.

Asset	IBM	INTC	CSCO
Min. Wt.	0	0	0
Max. Wt.	1	1	1

```
AssetMin = 0
AssetMax = 1
NumAssets = 3
```

```
[A,b] = pcalims(AssetMin, AssetMax, NumAssets)
```

```
A =
```

```
    1    0    0
    0    1    0
    0    0    1
   -1    0    0
    0   -1    0
    0    0   -1
```

```
b =
```

```
    1
    1
    1
    0
    0
    0
```

Portfolio weights of 50% in IBM and 50% in INTC satisfy the constraints.

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

pcgcomp | pcglims | pcpvval | portcons | portopt | portstats

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

pcgcomp

Linear inequalities for asset group comparison constraints

As an alternative to `pcgcomp`, use the `Portfolio` object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using `Portfolio` objects, see “Portfolio Object Workflow”.

Syntax

```
[A,b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)
```

Arguments

GroupA	Number of groups (NGROUPS) by number of assets (NASSETS) specifications of groups to compare. Each row specifies a group. For a specific group, $\text{Group}(i, j) = 1$ if the group contains asset j ; otherwise, $\text{Group}(i, j) = 0$.
GroupB	
AtoBmin	Scalar or NGROUPS-long vectors of minimum and maximum ratios of allocations in GroupA to allocations in GroupB. NaN indicates no constraint between the two groups. Scalar bounds are applied to all group pairs. The total number of assets allocated to GroupA divided by the total number of assets allocated to GroupB is $\geq \text{AtoBmin}$ and $\leq \text{AtoBmax}$.
AtoBmax	

Description

`[A,b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)` specifies that the ratio of allocations in one group to allocations in another group is at least `AtoBmin` to 1 and at most `AtoBmax` to 1. Comparisons can be made between an arbitrary number of group pairs `NGROUPS` comprising subsets of `NASSETS` available investments.

A is a matrix and b a vector such that $A * \text{PortWts}' \leq b$, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

If `pcgcomp` is called with fewer than two output arguments, the function returns A concatenated with b [A, b].

Examples

Asset	INTC	XOM	RD
Region	North America	North America	Europe
Sector	Technology	Energy	Energy

Group	Min. Exposure	Max. Exposure
North America	0.30	0.75
Europe	0.10	0.55
Technology	0.20	0.50
Energy	0.20	0.80

Make the North American energy sector compose exactly 20% of the North American investment.

```
%          INTC  XOM  RD
GroupA = [  0   1   0 ]; % North American Energy
GroupB = [  1   1   0 ]; % North America
```

```
AtoBmin = 0.20;
AtoBmax = 0.20;
```

```
[A, b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)
```

```
A =
```

```
    0.2000    -0.8000     0
   -0.2000     0.8000     0
```

```
b =
```

```
    0
    0
```

Portfolio weights of 40% for INTC, 10% for XOM, and 50% for RD satisfy the constraints.

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

`pcalims` | `pcglims` | `pcpval` | `portcons` | `portopt`

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

pcglims

Linear inequalities for asset group minimum and maximum allocation

As an alternative to `pcglims`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Syntax

```
[A,b] = pcglims(Groups, GroupMin, GroupMax)
```

Arguments

Groups	Number of groups (NGROUPS) by number of assets (NASSETS) specification of which assets belong to which group. Each row specifies a group. For a specific group, $\text{Group}(i, j) = 1$ if the group contains asset j ; otherwise, $\text{Group}(i, j) = 0$.
GroupMin	Scalar or NGROUPS-long vectors of minimum and maximum
GroupMax	combined allocations in each group. NaN indicates no constraint. Scalar bounds are applied to all groups.

Description

`[A,b] = pcglims(Groups, GroupMin, GroupMax)` specifies minimum and maximum allocations to groups of assets. An arbitrary number of groups, NGROUPS, comprising subsets of NASSETS investments, is allowed.

A is a matrix and b a vector such that $A * \text{PortWts}' \leq b$, where PortWts is a 1-by-NASSETS vector of asset allocations.

If `pcglims` is called with fewer than two output arguments, the function returns A concatenated with b `[A,b]`.

Examples

Asset	INTC	XOM	RD
Region	North America	North America	Europe
Sector	Technology	Energy	Energy

Group	Min. Exposure	Max. Exposure
North America	0.30	0.75
Europe	0.10	0.55
Technology	0.20	0.50
Energy	0.50	0.50

Set the minimum and maximum investment in various groups.

```
%
Groups = [   INTC  XOM  RD ; % North America
           0   0   1 ; % Europe
           1   0   0 ; % Technology
           0   1   1 ]; % Energy
```

```
GroupMin = [0.30
            0.10
            0.20
            0.50];
```

```
GroupMax = [0.75
            0.55
            0.50
            0.50];
```

```
[A,b] = pcglims(Groups, GroupMin, GroupMax)
```

```
A =
```

```
-1   -1   0
 0    0  -1
-1    0   0
 0   -1  -1
 1    1   0
 0    0   1
```

```
    1    0    0
    0    1    1
```

b =

```
-0.3000
-0.1000
-0.2000
-0.5000
 0.7500
 0.5500
 0.5000
 0.5000
```

Portfolio weights of 50% in INTC, 25% in XOM, and 25% in RD satisfy the constraints.

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

`pcalims` | `pcgcomp` | `pcpval` | `portcons` | `portopt`

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

pcpval

Linear inequalities for fixing total portfolio value

As an alternative to `pcpval`, use the `Portfolio` object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using `Portfolio` objects, see “Portfolio Object Workflow”.

Syntax

```
[A,b] = pcpval(PortValue, NumAssets)
```

Arguments

PortValue	Scalar total value of asset portfolio (sum of the allocations in all assets). <code>PortValue = 1</code> specifies weights as fractions of the portfolio and return and risk numbers as rates instead of value.
NumAssets	Number of available asset investments.

Description

`[A,b] = pcpval(PortValue, NumAssets)` scales the total value of a portfolio of `NumAssets` assets to `PortValue`. All portfolio weights, bounds, return, and risk values except `ExpReturn` and `ExpCovariance` (see `portopt`) are in terms of `PortValue`.

`A` is a matrix and `b` a vector such that $A * \text{PortWts}' \leq b$, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

If `pcpval` is called with fewer than two output arguments, the function returns `A` concatenated with `b` `[A,b]`.

Examples

Scale the value of a portfolio of three assets = 1, so all return values are rates and all weight values are in fractions of the portfolio.

```
PortValue = 1;  
NumAssets = 3;
```

```
[A,b] = pcpval(PortValue, NumAssets)
```

```
A =
```

```
    1    1    1  
   -1   -1   -1
```

```
b =
```

```
    1  
   -1
```

Portfolio weights of 40%, 10%, and 50% in the three assets satisfy the constraints.

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

`pcalims` | `pcgcomp` | `pcglims` | `portcons` | `portopt`

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

peravg

Periodic average of FINTS object

Syntax

```
avgfts = peravg(tsobj)
```

```
avgfts = peravg(tsobj, numperiod)
```

```
avgfts = peravg(tsobj, daterange)
```

Arguments

<code>tsobj</code>	Financial time series object
<code>numperiod</code>	(Optional) Integer specifying the number of data points over which each periodic average should be averaged
<code>daterange</code>	(Optional) Time period over which the data is averaged

Description

`peravg` calculates periodic averages of a financial time series object. Periodic averages are calculated from the values per period defined. If the period supplied is a string, it is assumed as a range of date string. If the period is entered as numeric, the number represents the number of data points (financial time series periods) to be included in a period for the calculation. For example, if you enter `'01/01/98::01/01/99'` as the period input argument, `peravg` returns the average of the time series between those dates, inclusive. However, if you enter the number `5` as the period input, `peravg` returns a series of averages from the time series data taken `5` date points (financial time series periods) at a time.

`avgfts = peravg(tsobj, numperiod)` returns a structure `avgfts` that contains the periodic (per `numperiod` periods) average of the financial time series object. `avgfts` has field names identical to the data series names of `tsobj`.

`avgfts = peravg(tsobj, daterange)` returns a structure `avgfts` that contains the periodic (as specified by `daterange`) average of the financial time series object. `avgfts` has field names identical to the data series names of `tsobj`.

Note: `peravg` calculates periodic averages of a FINTS object. Periodic averages are calculated from the values per period defined. If the period supplied is a string, it is assumed as a range of date strings. If the period is entered as numeric, the number represents the number of data points to be included in a period for the calculation.

Examples

If you enter `01-Jan-2001::03-Jan-2001` as the period input argument, `peravg` returns the average of the time series between those dates, inclusive. However, if you enter the number `5` as the period input, `peravg` returns a series of averages from the time series data, taken `5` date points at a time.

```
%% Create the FINTS object %%
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
         '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates, 1), 1), times]);
data = [(1:6)', 2*(1:6)'];
myFts = fints(dates_times, data, {'Data1', 'Data2'}, 1, 'My first FINTS')

%% Create the FINTS object %%
[p, pFts] = peravg(myFts, 3)

p =

    Data1: [2 5]
    Data2: [4 10]

pFts =

    desc: My first FINTS
    freq: Daily (1)

    'dates: (2)'    'times: (2)'    'Data1: (2)'    'Data2: (2)'
    '02-Jan-2001'  '11:00'        [         2]    [         4]
    '03-Jan-2001'  '12:00'        [         5]    [        10]

[p, pFts] = peravg(myFts, '01-Jan-2001 12:00::03-Jan-2001 11:00')

p =
```

```
Data1: 3.5000  
Data2: 7
```

```
pFts =
```

```
desc: My first FINTS  
freq: Daily (1)
```

```
'dates: (1)'    'times: (1)'    'Data1: (1)'    'Data2: (1)'  
'03-Jan-2001'  '11:00'        [ 3.5000]       [ 7]
```

See Also

[mean](#) | [tsmovavg](#) | [mean](#)

periodicreturns

Periodic total returns from total return prices

Syntax

```
TotalReturn = periodicreturns(TotalReturnPrices)
```

```
TotalReturn = periodicreturns(TotalReturnPrices, Period)
```

Arguments

TotalReturnPrices	Number of observations (NUMOBS) by number of assets (NASSETS + 1) matrix of total return prices for a given security. Column 1 contains MATLAB serial date numbers. The remaining columns contain total return price data.
Period	(Optional) Periodicity flag used to compute total returns: 'd' = daily values (default) 'w' = weekly values 'm' = monthly values n = rolling return periodic values, where n is an integer

Description

`TotalReturn = periodicreturns(TotalReturnPrices)` calculates the daily total returns from a daily total return price series.

`TotalReturn = periodicreturns(TotalReturnPrices, Period)` calculates the total returns for a periodicity you specify from a daily total return price series.

`TotalReturn` is a NUMOBS-by-NASSETS + 1 matrix containing month-end dates and return values. Each row represents an observation. Column 1 contains month-end dates in MATLAB serial date number format. The remaining columns contain monthly return values.

Note: Although input returns can have dates in either ascending or descending order, output total returns in `TotalReturn` have dates in ascending order, with the earliest date in the first row `TotalReturn`, and the most recent date in the last row of `TotalReturn`.

See Also

`totalreturnprice`

plot

Plot data series

Syntax

```
plot(tsobj)
```

```
hp = plot(tsobj)
```

```
plot(tsobj, linefmt)
```

```
hp = plot(tsobj, linefmt)
```

```
plot(..., volumename, bar)
```

```
hp = plot(..., volumename, bar)
```

Arguments

<code>tsobj</code>	Financial time series object.
<code>linefmt</code>	(Optional) Line format.
<code>volumename</code>	(Optional) Specifies which data series is the volume series. <code>volumename</code> must be the exact data series name for the volume column (case sensitive).
<code>bar</code>	(Optional) <ul style="list-style-type: none">• <code>bar = 0</code> (default). Plot volume as a line.• <code>bar = 1</code>. Plot volume as a bar chart. The width of each bar is the same as the default in <code>bar</code>, <code>barh</code>.

Description

`plot(tsobj)` plots the data series contained in the object `tsobj`. Each data series will be a line. `plot` automatically generates a legend and dates on the *x*-axis. Grid is turned on by default. `plot` uses the default color order as if plotting a matrix.

The `plot` command automatically creates subplots when multiple time series are encountered, and they differ greatly on their decimal scales. For example, subplots are generated if one time series data set is in the 10s and another is in the 10,000s.

`hp = plot(tsobj)` additionally returns the handle(s) to the object(s) inside the plot figure. If there are multiple lines in the plot, `hp` is a vector of multiple handles.

`plot(tsobj, linefmt)` plots the data series in `tsobj` using the line format specified. For a list of possible line formats, see `plot` in the MATLAB documentation. The plot legend is not generated, but the dates on the *x*-axis and the plot grid are. The specified line format is applied to all data series; that is, all data series will have the same line type.

`hp = plot(tsobj, linefmt)` plots the data series in `tsobj` using the format specified. The plot legend is not generated, but the dates on the *x*-axis and the plot grid are. The specified line format is applied to all data series, that is, all data series can have the same line type. If there are multiple lines in the plot, `hp` is a vector of multiple handles.

`plot(..., volumename, bar)` additionally specifies which data series is the volume. The volume is plotted in a subplot below the other data series. If `bar = 1`, the volume is plotted as a bar chart. Otherwise, a line plot is used.

`hp = plot(..., volumename, bar)` returns handles for each line. If `bar = 1`, the handle to the patch for the bars is also returned.

Note To turn the legend off, enter `legend off` at the MATLAB command line. Once you turn it off, the legend is essentially deleted. To turn it back on, recreate it using the `legend` command as if you are creating it for the first time. To turn the grid off, enter `grid off`. To turn it back on, enter `grid on`.

See Also

`candle` | `highlow` | `chartfts` | `grid` | `legend` | `plot`

plotFrontier

Class: Portfolio

Plot efficient frontier

Syntax

```
[prsk,pret] = plotFrontier(obj)
[prsk,pret] = plotFrontier(obj,varargin)
```

Description

[prsk,pret] = plotFrontier(obj) plots the efficient frontier.

[prsk,pret] = plotFrontier(obj,varargin) plots the efficient frontier with multiple types of input methods. There are four ways to use `plotFrontier`:

- Method 1 — Given a Portfolio object `obj`, estimate the efficient frontier with default number of 10 portfolios on the frontier.
- Method 2 — Given a Portfolio object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier.
- Method 3 — Given a Portfolio object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios. This method assumes that you provide valid efficient portfolios as input.
- Method 4 — Given a Portfolio object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier. This method assumes that you provide valid inputs for efficient portfolio risks and returns.

Note: `plotFrontier` handles multiple input formats as described above. Given an asset universe with `NumAssets` assets and an efficient frontier with `NumPorts` portfolios, remember that portfolio weights are `NumAsset`-by-`NumPorts` matrices and that portfolio risks and returns are `NumPorts` column vectors.

Tips

You can also use dot notation to plot the efficient frontier.

```
[prsk, pret] = obj.plotFrontier;
```

Input Arguments

obj

Portfolio object [Portfolio].

varargin

(Optional) **varargin** can be **NumPorts**, **PortRisk**, **PortReturn**, or **PortWeights** depending on which of the four input methods you use:

- Method 1 — Given a Portfolio object **obj**, estimate the efficient frontier with the default number of 10 portfolios on the frontier:

```
[prsk, pret] = plotFrontier(obj)
```

- Method 2 — Given a Portfolio object **obj**, estimate the efficient frontier with a specified number of portfolios **NumPorts** on the frontier:

```
[prsk, pret] = plotFrontier(obj, NumPorts)
```

- Method 3 — Given a Portfolio object **obj** with estimated efficient portfolios in **PortWeights**, plot the efficient frontier with those portfolios:

```
[prsk, pret] = plotFrontier(obj, PortWeights)
```

This method assumes that you provide valid efficient portfolios as inputs.

- Method 4 — Given a Portfolio object **obj** with estimated portfolio risks (**PortRisk**) and returns (**PortReturn**), plot the efficient frontier:

```
[prsk, pret] = plotFrontier(obj, PortRisk, PortReturn)
```

This method assumes that you provide valid efficient portfolio risks and returns as inputs.

Output Arguments

prsk

Estimated efficient portfolio risks (standard deviation of returns).

pret

Estimated efficient portfolio returns.

Note:

- If the Portfolio object has a name in the **Name** property, the name is displayed as the title of the plot. Otherwise, the plot is just labeled “Efficient Frontier.”
 - If the Portfolio object has an initial portfolio in the **InitPort** property, the initial portfolio is plotted and labeled.
 - If portfolio risks and returns are inputs, make sure that risks come first in the calling sequence. In addition, if portfolio risks and returns are not sorted in ascending order, this method performs the sort. On output, the sorted moments are returned.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

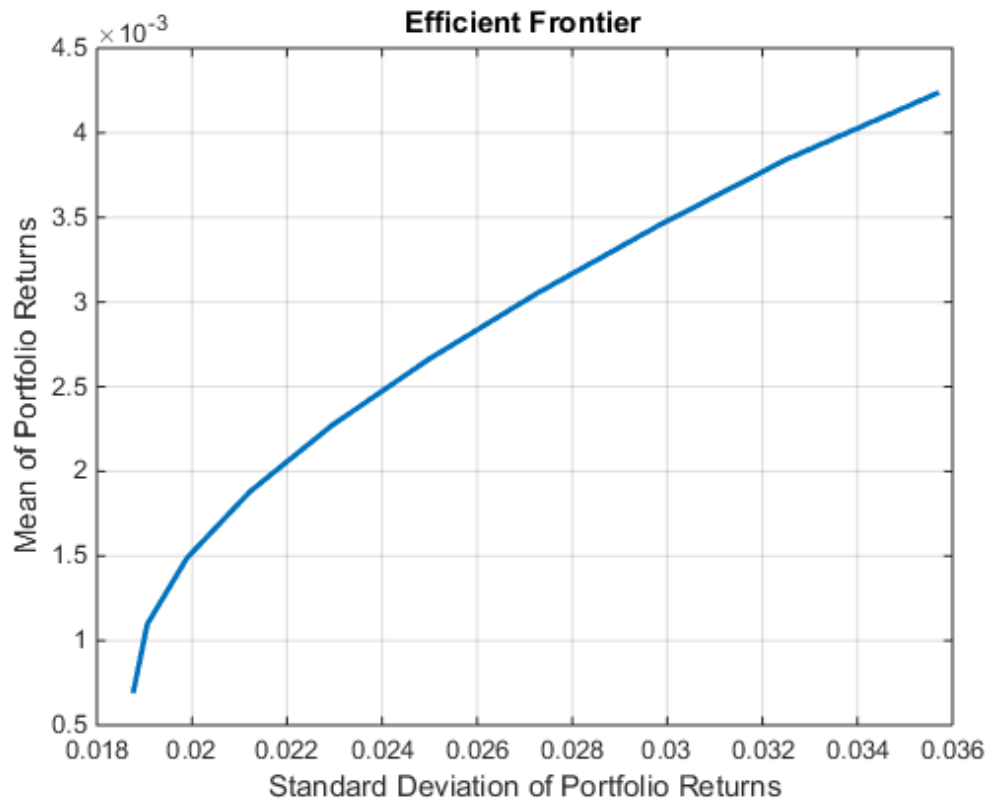
Examples

Plot the Efficient Frontier for the Portfolio Object

Given a portfolio `p`, plot the efficient frontier.

```
load CAPMuniverse
```

```
p = Portfolio('AssetList',Assets(1:12));  
p = estimateAssetMoments(p, Data(:,1:12),'missingdata',true);  
p = setDefaultConstraints(p);  
plotFrontier(p);
```



- “Plotting the Efficient Frontier” on page 4-98

See Also
Portfolio

plotFrontier

Class: PortfolioCVaR

Plot single efficient frontier for PortfolioCVaR object

Syntax

```
[prsk,pret] = plotFrontier(obj)
[prsk,pret] = plotFrontier(obj,varargin)
```

Description

`[prsk,pret] = plotFrontier(obj)` plots a single efficient frontier for a PortfolioCVaR object.

`[prsk,pret] = plotFrontier(obj,varargin)` to plot a single efficient frontier with multiple types of input methods. There are four ways to use `plotFrontier`:

- Method 1 — Given a CVaR portfolio object `obj`, estimate the efficient frontier with default number of 10 portfolios on the frontier.

```
[prsk, pret] = plotFrontier(obj)
```

- Method 2 — Given a CVaR portfolio object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier.

```
[prsk, pret] = plotFrontier(obj, NumPorts)
```

- Method 3 — Given a CVaR portfolio object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios. This method assumes that you provide valid efficient portfolios as input.

```
[prsk, pret] = plotFrontier(obj, PortWeights)
```

- Method 4 — Given a CVaR portfolio object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier. This method assumes that you provide valid efficient portfolio risks and returns as inputs.

```
[prsk, pret] = plotFrontier(obj, PortRisk, PortReturn);
```

Note: `plotFrontier` handles multiple input formats as described above. Given an asset universe with `NumAssets` assets and an efficient frontier with `NumPorts` portfolios, remember that portfolio weights are `NumAsset`-by-`NumPorts` matrices and that portfolio risks and returns are `NumPorts` column vectors.

Tips

You can also use dot notation to plot a single efficient frontier for a `PortfolioCVaR` object.

```
[prsk, pret] = obj.plotFrontier;
```

Input Arguments

obj

CVaR portfolio object [`PortfolioCVaR`].

varargin

(Optional) `varargin` can be `NumPorts`, `PortRisk`, `PortReturn`, or `PortWeights` depending on which of the four input methods you use:

- Method 1 — Given a CVaR portfolio object `obj`, estimate the efficient frontier using the default number of 10 portfolios on the frontier:

```
[prsk, pret] = plotFrontier(obj)
```

- Method 2 — Given a CVaR portfolio object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier:

```
[prsk, pret] = plotFrontier(obj, NumPorts)
```

- Method 3 — Given a CVaR portfolio object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios:

```
[prsk, pret] = plotFrontier(obj, PortWeights)
```

This method assumes that you provide valid efficient portfolios as input.

- Method 4 — Given a CVaR portfolio object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier:

```
[prsk, pret] = plotFrontier(obj, PortRisk, PortReturn)
```

This method assumes that you provide valid efficient portfolio risks and returns as inputs.

Output Arguments

prsk

Estimated efficient portfolio risks (conditional value-at-risk) [vector].

pret

Estimated efficient portfolio returns [vector].

Note:

- If the CVaR portfolio object has a name in the **Name** property, the name is displayed as the title of the plot. Otherwise, the plot is just labeled “Efficient Frontier.”
 - If the CVaR portfolio object has an initial portfolio in the **InitPort** property, the initial portfolio is plotted and labeled.
 - If portfolio risks and returns are inputs, ensure that risks come first in the calling sequence. In addition, if portfolio risks and returns are not sorted in ascending order, this method performs the sort. On output, the sorted moments are returned.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Plot the Efficient Frontier for the PortfolioCVaR Object

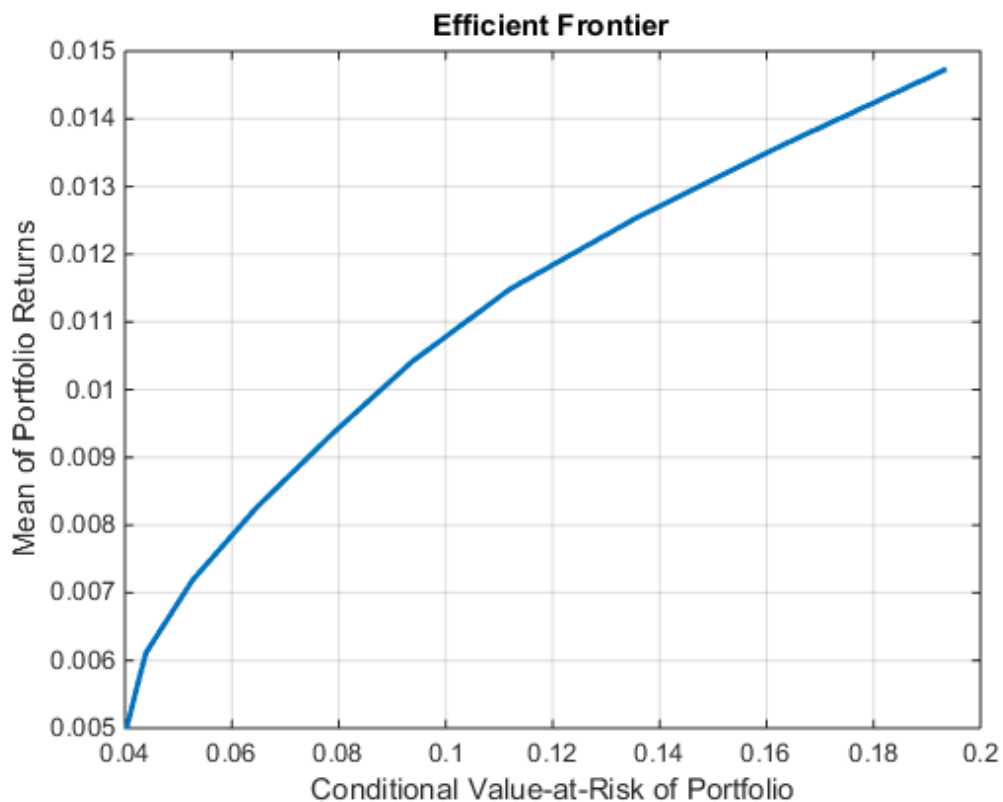
Given a PortfolioCVaR `p`, plot the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```



- “Plotting the Efficient Frontier” on page 5-98

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 5-23

plotFrontier

Class: PortfolioMAD

Plot single efficient frontier for PortfolioMAD object

Syntax

```
[prsk,pret] = plotFrontier(obj)
[prsk,pret] = plotFrontier(obj,varargin)
```

Description

`[prsk,pret] = plotFrontier(obj)` plots a single efficient frontier for a PortfolioMAD object.

`[prsk,pret] = plotFrontier(obj,varargin)` to plot a single efficient frontier with multiple types of input methods. There are four ways to use `plotFrontier`:

- Method 1 — Given a PortfolioMAD object `obj`, estimate the efficient frontier with default number of 10 portfolios on the frontier.

```
[prsk, pret] = plotFrontier(obj)
```

- Method 2 — Given a PortfolioMAD object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier.

```
[prsk, pret] = plotFrontier(obj, NumPorts)
```

- Method 3 — Given a PortfolioMAD object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios. This method assumes that you provide valid efficient portfolios as input.

```
[prsk, pret] = plotFrontier(obj, PortWeights)
```

- Method 4 — Given a PortfolioMAD object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier. This method assumes that you provide valid efficient portfolio risks and returns as inputs.

```
[prsk, pret] = plotFrontier(obj, PortRisk, PortReturn);
```

Note: `plotFrontier` handles multiple input formats as described above. Given an asset universe with `NumAssets` assets and an efficient frontier with `NumPorts` portfolios, remember that portfolio weights are `NumAsset`-by-`NumPorts` matrices and that portfolio risks and returns are `NumPorts` column vectors.

Tips

You can also use dot notation to plot a single efficient frontier for a `PortfolioMAD` object.

```
[prsk, pret] = obj.plotFrontier;
```

Input Arguments

obj

MAD portfolio object [`PortfolioMAD`].

varargin

(Optional) `varargin` can be `NumPorts`, `PortRisk`, `PortReturn`, or `PortWeights` depending on which of the four input methods you use:

- Method 1 — Given a `PortfolioMAD` object `obj`, estimate the efficient frontier using the default number of 10 portfolios on the frontier:

```
[prsk, pret] = plotFrontier(obj)
```

- Method 2 — Given a `PortfolioMAD` object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier:

```
[prsk, pret] = plotFrontier(obj, NumPorts)
```

- Method 3 — Given a `PortfolioMAD` object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios:

```
[prsk, pret] = plotFrontier(obj, PortWeights)
```

This method assumes that you provide valid efficient portfolios as input.

- Method 4 — Given a `PortfolioMAD` object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier:

```
[prsk, pret] = plotFrontier(obj, PortRisk, PortReturn)
```

This method assumes that you provide valid efficient portfolio risks and returns as inputs.

Output Arguments

prsk

Estimated efficient portfolio risks (conditional value-at-risk) [vector].

pret

Estimated efficient portfolio returns [vector].

Note:

- If the MAD portfolio object has a name in the **Name** property, the name is displayed as the title of the plot. Otherwise, the plot is just labeled “Efficient Frontier.”
 - If the MAD portfolio object has an initial portfolio in the **InitPort** property, the initial portfolio is plotted and labeled.
 - If portfolio risks and returns are inputs, ensure that risks come first in the calling sequence. In addition, if portfolio risks and returns are not sorted in ascending order, this method performs the sort. On output, the sorted moments are returned.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Plot Efficient Frontier for PortfolioMAD Object

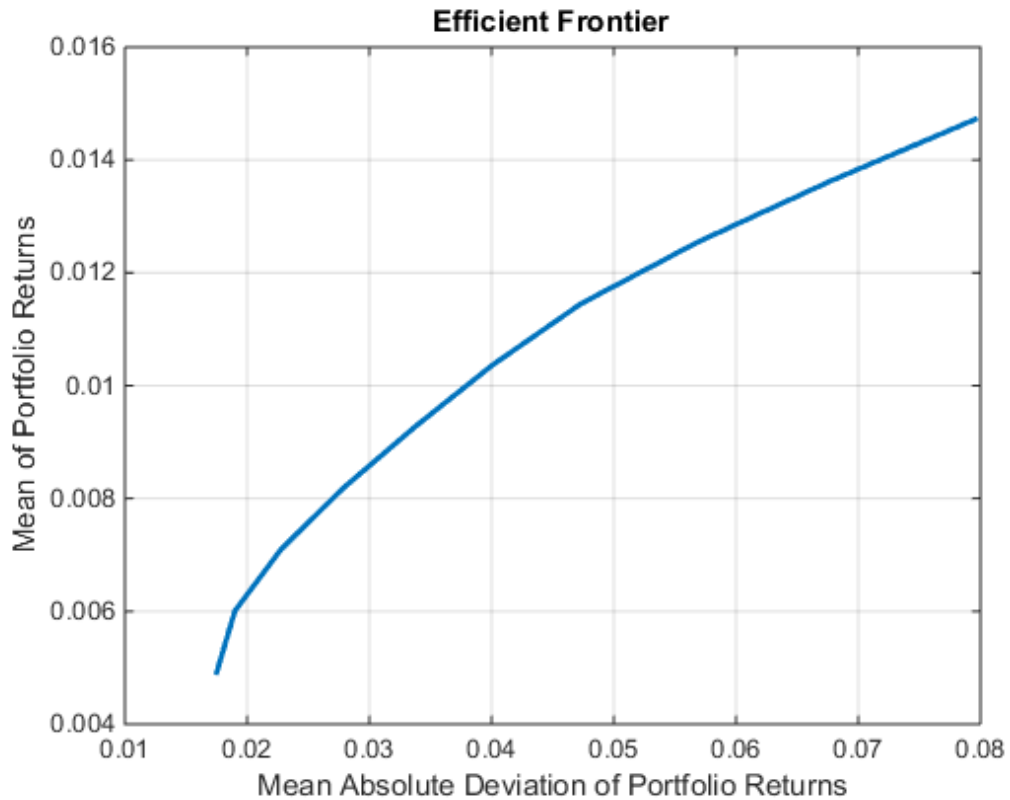
Given a PortfolioMAD `p`, plot the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

plotFrontier(p);
```

- “Plotting the Efficient Frontier” on page 6-97

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

More About

- “Estimating Efficient Portfolios and Frontiers” on page 6-23

plus

Financial time series addition

Syntax

```
newfts = tsobj_1 + tsobj_2
```

```
newfts = tsobj + array
```

```
newfts = array + tsobj
```

Arguments

<code>tsobj_1, tsobj_2</code>	A pair of financial time series objects.
<code>array</code>	A scalar value or array with the number of rows equal to the number of dates in <code>tsobj</code> and the number of columns equal to the number of data series in <code>tsobj</code> .

Description

`plus` is an element-by-element addition of the components.

`newfts = tsobj_1 + tsobj_2` adds financial time series objects. If an object is to be added to another object, both objects must have the same dates and data series names, although the order need not be the same. The order of the data series, when one financial time series object is added to another, follows the order of the first object.

`newfts = tsobj + array` adds an array element by element to a financial time series object.

`newfts = array + tsobj` adds a financial time series object element by element to an array.

See Also

`minus` | `rdivide` | `times`

pointfig

Point and figure chart

Syntax

```
pointfig(Asset)
```

Description

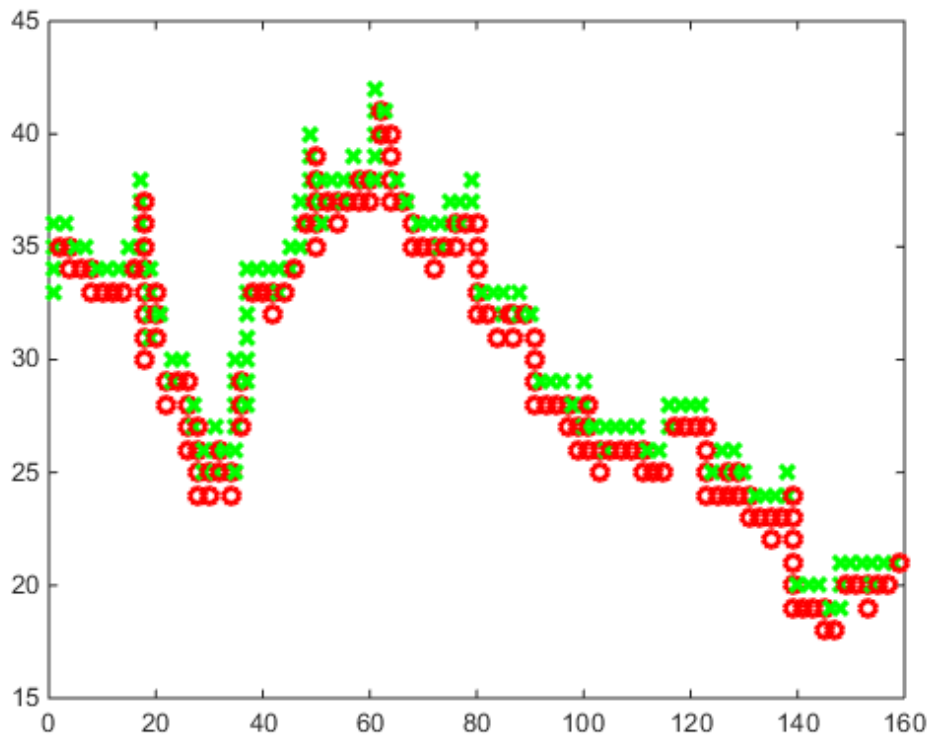
`pointfig(Asset)` plots a point and figure chart for a vector of price data `Asset`. Upward price movements are plotted as X's and downward price movements are plotted as O's.

Examples

Create a Point and Figure Chart for an Equity's Closing Prices

Using the price data for the equity DIS, plot a point and figure chart for the closing prices.

```
load disney;  
pointfig(dis_CLOSE)
```



See Also

[bolling](#) | [candle](#) | [dateaxis](#) | [highlow](#) | [movavg](#)

portalloc

Optimal capital allocation to efficient frontier portfolios

Syntax

```
[RiskyRisk, RiskyReturn, RiskyWts, RiskyFraction, OverallRisk,
OverallReturn] = portalloc(PortRisk, PortReturn, PortWts,
RisklessRate, BorrowRate, RiskAversion)
```

Arguments

PortRisk	Standard deviation of each risky asset efficient frontier portfolio. A number of portfolios (NPORTS) by 1 vector.
PortReturn	Expected return of each risky asset efficient frontier portfolio. An NPORTS-by-1 vector.
PortWts	Weights allocated to each asset. An NPORTS by number of assets (NASSETS) matrix of weights allocated to each asset. Each row represents an efficient frontier portfolio of risky assets. Total of all weights in a portfolio is 1.
RisklessRate	Risk-free lending rate. A decimal number.
BorrowRate	(Optional) Borrowing rate. A decimal number. If borrowing is not desired, or not an option, set to NaN (default).
RiskAversion	(Optional) Coefficient of investor's degree of risk aversion. Higher numbers indicate greater risk aversion. Typical coefficients range between 2.0 and 4.0 (Default = 3). Note: Consider that a less risk-averse investor would be expected to accept much greater risk and, consequently, a more risk-averse investor would accept less risk for a given level of return. Therefore, making the RiskAversion argument higher reflects the risk-return tradeoff in the data.

Description

[RiskyRisk, RiskyReturn, RiskyWts, RiskyFraction, OverallRisk, OverallReturn] = portalloc(PortRisk, PortReturn, PortWts, RisklessRate, BorrowRate, RiskAversion) computes the optimal risky portfolio, and the optimal allocation of funds between the risky portfolio and the risk-free asset.

RiskyRisk is the standard deviation of the optimal risky portfolio.

RiskyReturn is the expected return of the optimal risky portfolio.

RiskyWts is a 1-by-NASSETS vector of weights allocated to the optimal risky portfolio. The total of all weights in the portfolio is 1.

RiskyFraction is the fraction of the complete portfolio allocated to the risky portfolio.

OverallRisk is the standard deviation of the optimal overall portfolio.

OverallReturn is the expected rate of return of the optimal overall portfolio.

portalloc generates a plot of the optimal capital allocation if you invoke it without output arguments.

Examples

Compute the Optimal Risky Portfolio

This example shows how to compute the optimal risky portfolio by generating the efficient frontier from the asset data and then finding the optimal risky portfolio and allocate capital. The risk-free investment return is 8%, and the borrowing rate is 12%.

```
ExpReturn = [0.1 0.2 0.15];  
  
ExpCovariance = [0.005   -0.010   0.004  
                -0.010   0.040   -0.002  
                0.004   -0.002   0.023];  
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn,...  
ExpCovariance);  
  
RisklessRate = 0.08;  
BorrowRate   = 0.12;  
RiskAversion  = 3;
```

```
[RiskyRisk, RiskyReturn, RiskyWts, RiskyFraction, ...  
OverallRisk, OverallReturn] = portalloc(PortRisk, PortReturn, ...  
PortWts, RisklessRate, BorrowRate, RiskAversion)
```

```
RiskyRisk =
```

```
0.1283
```

```
RiskyReturn =
```

```
0.1788
```

```
RiskyWts =
```

```
0.0265    0.6023    0.3712
```

```
RiskyFraction =
```

```
1.1898
```

```
OverallRisk =
```

```
0.1527
```

```
OverallReturn =
```

```
0.1899
```

References

Bodie, Kane, and Marcus, *Investments*, Second Edition, Chapters 6 and 7.

See Also

frontcon | portrand | portstats

portalpha

Compute risk-adjusted alphas and returns for one or more assets

Syntax

```
portalpha(Asset, Benchmark)
```

```
portalpha(Asset, Benchmark, Cash)
```

```
portalpha(Asset, Benchmark, Cash, Choice)
```

```
Alpha = portalpha(Asset, Benchmark, Cash, Choice)
```

```
[Alpha, RAReturn] = portalpha(Asset, Benchmark, Cash, Choice)
```

Arguments

Asset	NUMSAMPLES x NUMSERIES matrix with NUMSAMPLES observations of asset returns for NUMSERIES asset return series.
Benchmark	NUMSAMPLES vector of returns for a benchmark asset. The periodicity must be the same as the periodicity of Asset. For example, if Asset is monthly data, then Benchmark should be monthly returns.
Cash	(Optional) Either a scalar return for a riskless asset or a vector of asset returns to be a proxy for a “riskless” asset. In either case, the periodicity must be the same as the periodicity of Asset. For example, if Asset is monthly data, then Cash must be monthly returns. If no value is supplied, the default value for Cash returns is 0.
Choice	(Optional) A number, string, or cell array of numbers or strings to indicate one or more measures to be computed from among a number of risk-adjusted alphas and return measures. The number of choices selected in Choice is NUMCHOICES. The current list of choices is given in the following table:

Code	Description
'xs'	Excess Return (no risk adjustment)
'sml'	Security Market Line
'capm'	Jensen's Alpha
'mm'	Modigliani & Modigliani
'gh1'	Graham-Harvey 1
'gh2'	Graham-Harvey 2
'all'	Compute all measures

Choice is specified by using the code from the table (for example, to select the Modigliani & Modigliani measure, `Choice = 'mm'`). A single choice is either a string or a scalar cell array with a single code from the table.

Multiple choices can be selected with a cell array of choice codes (for example, to select both Graham-Harvey measures, `Choice = { 'gh1' , 'gh2' }`). To select all choices, specify `Choice = 'all'`. If no value is supplied, the default choice is to compute the excess return with `Choice = 'xs'`. `Choice` is not case sensitive.

Description

Given `NUMSERIES` assets with `NUMSAMPLES` returns in a `NUMSAMPLES`-by-`NUMSERIES` matrix `Asset`, a `NUMSAMPLES` vector of `Benchmark` returns, and either a scalar `Cash` return or a `NUMSAMPLES` vector of `Cash` returns, compute risk-adjusted alphas and returns for one or more methods specified by `Choice`.

To summarize the outputs of `portalalpha`:

- `Alpha` is a `NUMCHOICES`-by-`NUMSERIES` matrix of risk-adjusted alphas for each series in `Asset` with each row corresponding to a specified measure in `Choice`.
- `RAReturn` is a `NUMCHOICES`-by-`NUMSERIES` matrix of risk-adjusted returns for each series in `Asset` with each row corresponding to a specified measure in `Choice`.

Note: NaN values in the data are ignored and, if NaNs are present, some results could be unpredictable. Although the alphas are comparable across measures, risk-adjusted returns depend on whether the **Asset** or **Benchmark** is levered or unlevered to match its risk with the alternative. If **Choice** = 'all', the order of rows in **Alpha** and **RAReturn** follows the order in the table. In addition, **Choice** = 'all' overrides all other choices.

Examples

See “Risk-Adjusted Return” on page 7-10.

References

John Lintner, "The Valuation of Risk Assets and the Selection of Risky Investments in Stocks Portfolios and Capital Budgets," *Review of Economics and Statistics*, Vol. 47, No. 1, February 1965, pp. 13-37.

John R. Graham and Campbell R. Harvey, "Market Timing Ability and Volatility Implied in Investment Newsletters' Asset Allocation Recommendations," *Journal of Financial Economics*, Vol. 42, 1996, pp. 397-421.

Franco Modigliani and Leah Modigliani, "Risk-Adjusted Performance: How to Measure It and Why," *Journal of Portfolio Management*, Vol. 23, No. 2, Winter 1997, pp. 45-54.

Jan Mossin, "Equilibrium in a Capital Asset Market," *Econometrica*, Vol. 34, No. 4, October 1966, pp. 768-783.

William F. Sharpe, "Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk," *Journal of Finance*, Vol. 19, No. 3, September 1964, pp. 425-442.

See Also

inforatio | sharpe

portcons

Portfolio constraints

As an alternative to `portcons`, use the `Portfolio` object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using `Portfolio` objects, see “Portfolio Object Workflow”.

Syntax

```
ConSet = portcons(varargin)
```

Description

Using linear inequalities, `portcons` generates a matrix of constraints for a portfolio of asset investments. The matrix `ConSet` is defined as $\text{ConSet} = [A \ b]$. `A` is a matrix and `b` a vector such that $A * \text{PortWts}' \leq b$ sets the value, where `PortWts` is a 1-by-number of assets (`NASSETS`) vector of asset allocations.

`ConSet = portcons('ConstType', Data1, ..., DataN)` creates a matrix `ConSet`, based on the constraint type `ConstType`, and the constraint parameters `Data1, ..., DataN`.

`ConSet = portcons('ConstType1', Data11, ..., Data21, ..., Data2N, ...)` creates a matrix `ConSet`, based on the constraint types `ConstTypeN`, and the corresponding constraint parameters `DataN1, ..., DataNN`.

Constraint Type	Description	Values
Default	All allocations are ≥ 0 ; no short selling allowed. Combined value of portfolio allocations normalized to 1.	<code>NumAssets</code> (required). Scalar representing number of assets in portfolio.

Constraint Type	Description	Values
PortValue	Fix total value of portfolio to PVal.	<p>PVal (required). Scalar representing total value of portfolio.</p> <p>NumAssets (required). Scalar representing number of assets in portfolio. See pcpval.</p>
AssetLims	Minimum and maximum allocation per asset.	<p>AssetMin (required). Scalar or vector of length NASSETS, specifying minimum allocation per asset.</p> <p>AssetMax (required). Scalar or vector of length NASSETS, specifying maximum allocation per asset.</p> <p>NumAssets (optional). See pcalims.</p>
GroupLims	Minimum and maximum allocations to asset group.	<p>Groups (required). NGROUPS-by-NASSETS matrix specifying which assets belong to each group.</p> <p>GroupMin (required). Scalar or a vector of length NGROUPS, specifying minimum combined allocations in each group.</p> <p>GroupMax (required). Scalar or a vector of length NGROUPS, specifying maximum combined allocations in each group.</p> <p>See pcglims.</p>

Constraint Type	Description	Values
GroupComparison	Group-to-group comparison constraints.	<p>GroupA (required). NGROUPS-by-NASSETS matrix specifying first group in the comparison.</p> <p>AtoBmin (required). Scalar or vector of length NGROUPS specifying minimum ratios of allocations in GroupA to allocations in GroupB.</p> <p>AtoBmax (required). Scalar or vector of length NGROUPS specifying maximum ratios of allocations in GroupA to allocations in GroupB.</p> <p>GroupB (required). NGROUPS-by-NASSETS matrix specifying second group in the comparison.</p> <p>See <code>pcgcomp</code>.</p>
Custom	Custom linear inequality constraints $A * PortWts' \leq b$.	<p>A (required). NCONSTRAINTS-by-NASSETS matrix, specifying weights for each asset in each inequality equation.</p> <p>b (required). Vector of length NCONSTRAINTS specifying the right hand sides of the inequalities.</p> <hr/> <p>Note: For more information using <code>CUSTOM</code>, see “Specifying Additional Constraints” on page 3-17.</p>

Examples

Constrain a portfolio of three assets:

Asset	IBM	HPQ	XOM
Group	A	A	B
Min. Wt.	0	0	0
Max. Wt.	0.5	0.9	0.8

```
NumAssets = 3;
PVal = 1; % Scale portfolio value to 1.
```

```
AssetMin = 0;
AssetMax = [0.5 0.9 0.8];
GroupA = [1 1 0];
GroupB = [0 0 1];
AtoBmax = 1.5 % Value of assets in Group A at most 1.5 times value
             % in group B.

ConSet = portcons('PortValue', PVal, NumAssets, 'AssetLims',...
                 AssetMin, AssetMax, NumAssets, 'GroupComparison', GroupA, NaN,...
                 AtoBmax, GroupB)
```

```
ConSet =

    1.0000    1.0000    1.0000    1.0000
   -1.0000   -1.0000   -1.0000   -1.0000
    1.0000         0         0    0.5000
         0    1.0000         0    0.9000
         0         0    1.0000    0.8000
   -1.0000         0         0         0
         0   -1.0000         0         0
         0         0   -1.0000         0
    1.0000    1.0000   -1.5000         0
```

For instance, one possible solution for portfolio weights that satisfy the constraints is 30% in IBM, 30% in HPQ, and 40% in XOM.

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

`pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portopt`

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

Portfolio class

Superclasses: AbstractPortfolio

Portfolio object for mean-variance portfolio optimization and analysis

Description

The Portfolio object implements mean-variance portfolio optimization and is derived from the abstract portfolio optimization class `AbstractPortfolio`. Portfolio objects implement all methods in the `AbstractPortfolio` class along with methods that are specific to mean-variance portfolio optimization.

The main workflow for portfolio optimization is to create an instance of a Portfolio object that completely specifies a portfolio optimization problem and to operate on the Portfolio object to obtain and analyze efficient portfolios. A mean-variance optimization problem is completely specified with the following three elements:

- A universe of assets with estimates for the prospective mean and covariance of asset total returns for a period of interest.
- A portfolio set that specifies the set of portfolio choices in terms of a collection of constraints.
- A model for portfolio return and risk, which, for mean-variance optimization, is either the gross or net mean of portfolio returns and the standard deviation of portfolio returns.

After you specify three elements in an unambiguous way, you can solve and analyze portfolio optimization problems. The simplest mean-variance portfolio optimization problem has:

- A mean and covariance of asset total returns
- Nonnegative weights for all portfolios that sum to 1 (the summation constraint is known as a budget constraint)
- Built-in models for portfolio return and risk that use the mean and covariance of asset total returns

Given mean and covariance of asset returns in the variables `AssetMean` and `AssetCovar`, this problem is completely specified by:

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, ...
```

```
'LowerBound', 0, 'Budget')
```

or equivalently by:

```
p = Portfolio;  
p = setAssetMoments(p, AssetMean, AssetCovar);  
p = setDefaultConstraints(p);
```

Construction

`p = Portfolio` constructs an empty Portfolio object for mean-variance portfolio optimization and analysis. You can then add elements to the Portfolio object using the supported add and set methods. For more information, see “Constructing the Portfolio Object”.

`p = Portfolio(Name, Value)` constructs a Portfolio object for mean-variance portfolio optimization and analysis with additional options specified by one or more Name, Value pair arguments. **Name** is a property name and **Value** is its corresponding value. **Name** must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as **Name1, Value1, . . . , NameN, ValueN**.

`p = Portfolio(p, Name, Value)` constructs a Portfolio object for mean-variance portfolio optimization and analysis using a previously constructed Portfolio object **p** with additional options specified by one or more Name, Value pair arguments.

Input Arguments

p

(Optional) Previously constructed Portfolio object (p).

Property Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1, Value1, . . . , NameN, ValueN**.

'AInequality'

Linear inequality constraint matrix ([] or [matrix]).

Default: []

'AssetCovar'

Covariance of asset returns ([] or square matrix).

Default: []

'AssetList'

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Default: []

'AssetMean'

Mean of asset returns ([] or vector).

Default: []

'bInequality'

Linear inequality constraint vector ([] or [vector]).

Default: []

'BuyCost'

Proportional purchase costs ([] or vector).

Default: []

'BuyTurnover'

Turnover constraint on purchases ([] or [scalar]).

Default: []

'GroupA'

Group A weights to be bounded by weights in group B ([] or [matrix]).

Default: []

'GroupB'

Group B weights ([] or [matrix]).

Default: []

'GroupMatrix'

Group membership matrix ([] or [matrix]).

Default: []

'InitPort'

Initial portfolio ([] or vector).

Default: []

'LowerBudget'

Lower-bound budget constraint ([] or [scalar]).

Default: []

'LowerGroup'

Lower-bound group constraint ([] or [vector]).

Default: []

'LowerRatio'

Minimum ratio of allocations between groups A and B ([] or [vector]).

Default: []

'Name'

Name for instance of the Portfolio object ([] or [string]).

Default: []

'NumAssets'

Number of assets in the universe ([] or [integer scalar]).

Default: []

'RiskFreeRate'

Risk-free rate ([] or scalar).

Default: []

'SellCost'

Proportional sales costs ([] or vector).

Default: []

'SellTurnover'

Turnover constraint on sales ([] or [scalar]).

Default: []

'Turnover'

Turnover constraint ([] or [scalar]).

Default: []

'UpperBound'

Upper-bound constraint ([] or [vector]).

Default: []

'UpperBudget'

Upper-bound budget constraint ([] or [scalar]).

Default: []

'UpperGroup'

Upper-bound group constraint ([] or [vector]).

Default: []

'UpperRatio'

Maximum ratio of allocations between groups A and B ([] or [vector]).

Default: []

Properties

The following properties are from the `Portfolio` class.

AssetCovar

Covariance of asset returns ([] or matrix).

Attributes:

<code>SetAccess</code>	<code>public</code>
<code>GetAccess</code>	<code>public</code>

AssetMean

Mean of asset returns ([] or vector).

Attributes:

<code>SetAccess</code>	<code>public</code>
<code>GetAccess</code>	<code>public</code>

BuyCost

Proportional purchase costs ([] or vector).

Attributes:

<code>SetAccess</code>	<code>public</code>
<code>GetAccess</code>	<code>public</code>

BuyTurnover

Turnover constraint on purchases ([] or [scalar]).

Attributes:

<code>SetAccess</code>	<code>public</code>
<code>GetAccess</code>	<code>public</code>

RiskFreeRate

Risk-free rate ([] or scalar).

Attributes:

SetAccess	public
GetAccess	public

SellCost

Proportional sales costs ([] or vector).

Attributes:

SetAccess	public
GetAccess	public

SellTurnover

Turnover constraint on sales ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

Turnover

Turnover constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

Properties

The following properties are inherited from the `AbstractPortfolio` class.

AEquality

Linear equality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

AInequality

Linear inequality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

AssetList

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Attributes:

SetAccess	public
GetAccess	public

bEquality

Linear equality constraint vector ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

bInequality

Linear inequality constraint vector ([] or [vector]).

Attributes:

SetAccess	public
-----------	--------

GetAccess public

GroupA

Group A weights to be bounded by group B ([] or [matrix]).

Attributes:

SetAccess public

GetAccess public

GroupB

Group B weights ([] or [matrix]).

Attributes:

SetAccess public

GetAccess public

GroupMatrix

Group membership matrix ([] or [matrix]).

Attributes:

SetAccess public

GetAccess public

InitPort

Initial portfolio ([] or vector).

Attributes:

SetAccess public

GetAccess public

LowerBound

Lower-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerBudget

Lower-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

LowerGroup

Lower-bound group constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerRatio

Minimum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

Name

Name for instance of the Portfolio object ([] or [string]).

Attributes:

SetAccess	public
GetAccess	public

NumAssets

Number of assets in the universe ([] or [integer scalar]).

Attributes:

SetAccess	public
GetAccess	public

UpperBound

Upper-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

UpperBudget

Upper-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

UpperGroup

Upper-bound group constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

UpperRatio

Maximum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

Inherited Methods

The following methods are inherited from the `AbstractPortfolio` class.

<code>addEquality</code>	Add equality constraints for portfolio weights to existing constraints.
<code>addGroupRatio</code>	Add group ratio constraints for portfolio weights to existing constraints.
<code>addGroups</code>	Add group constraints for portfolio weights to existing constraints.
<code>addInequality</code>	Add inequality constraints for portfolio weights to existing constraints.
<code>checkFeasibility</code>	Determine if portfolios are members of the set of feasible portfolios.
<code>estimateBounds</code>	Determine if set of feasible portfolios is nonempty and bounded.
<code>estimateFrontier</code>	Estimate portfolios on the entire efficient frontier.
<code>estimateFrontierByReturn</code>	Estimate portfolios on the efficient frontier with targeted returns or return proxies.
<code>estimateFrontierByRisk</code>	Estimate portfolios on the efficient frontier with targeted risks or risk proxies.
<code>estimateFrontierLimits</code>	Estimate portfolios at the extreme ends of the efficient frontier (minimum risk and maximum return).
<code>estimateMaxSharpeRatio</code>	Estimate efficient portfolio that maximizes the Sharpe ratio.
<code>estimatePortReturn</code>	Estimate return or return proxy for specified portfolios.
<code>estimatePortRisk</code>	Estimate risk or risk proxy for specified portfolios.
<code>getBounds</code>	Get lower and upper bounds from the object.
<code>getBudget</code>	Get lower and upper budget constraints from the object.

<code>getEquality</code>	Get equality constraint matrix and vector from the object.
<code>getGroupRatio</code>	Get base matrix, comparison matrix, and lower and upper bounds for group ratio constraints from the object.
<code>getGroups</code>	Get group matrix and lower and upper bounds for group constraints from the object.
<code>getInequality</code>	Get inequality constraint matrix and vector from the object.
<code>plotFrontier</code>	Plot efficient frontier and optionally obtain risks and returns for portfolios on the efficient frontier.
<code>setAssetList</code>	Set up a list of asset names and symbols to be associated with assets in the universe.
<code>setBounds</code>	Set up lower and upper bounds for portfolio weights.
<code>setBudget</code>	Set up lower and upper budget constraints for portfolio weights.
<code>setDefaultConstraints</code>	Set up default constraints for portfolio weights (nonnegative weights that must sum to 1).
<code>setEquality</code>	Set up equality constraints for portfolio weights.
<code>setGroupRatio</code>	Set up group ratio constraints for portfolio weights.
<code>setGroups</code>	Set up group constraints for portfolio weights.
<code>setInequality</code>	Set up inequality constraints for portfolio weights.
<code>setInitPort</code>	Set up initial portfolio weights.
<code>setOptions</code>	Set up hidden control properties in object (not implemented).
<code>setSolver</code>	Set up solver to estimate efficient portfolios.

Methods

<code>estimateAssetMoments</code>	Estimate mean and covariance of asset returns from price or return data.
<code>estimatePortMoments</code>	Estimate mean and standard deviation of portfolio returns for specified portfolios.
<code>getAssetMoments</code>	Get mean and covariance of asset returns from the object.
<code>getCosts</code>	Get purchase and sales proportional transaction costs from the object.
<code>getOneWayTurnover</code>	Get one-way portfolio turnover constraints.
<code>setAssetMoments</code>	Set up mean and covariance of asset returns.
<code>setCosts</code>	Set up purchase and sale proportional transaction costs for assets in the universe.
<code>setOneWayTurnover</code>	Set up one-way portfolio turnover constraints.
<code>setTurnover</code>	Set up average turnover constraints for portfolio weights.

Definitions

Mean-Variance Portfolio Optimization

For more information on the theory and definition of mean-variance optimization supported by portfolio optimization tools in Financial Toolbox, see “Portfolio Optimization Theory” on page 4-2.

Copy Semantics

Value. To learn how value classes affect copy operations, see “Copying Objects” in the MATLAB documentation.

Examples

Construct a Portfolio Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = Portfolio('AssetList',Assets(1:12));
p = estimateAssetMoments(p, Data(:,1:12),'missingdata',true);
p = setDefaultConstraints(p);
plotFrontier(p);

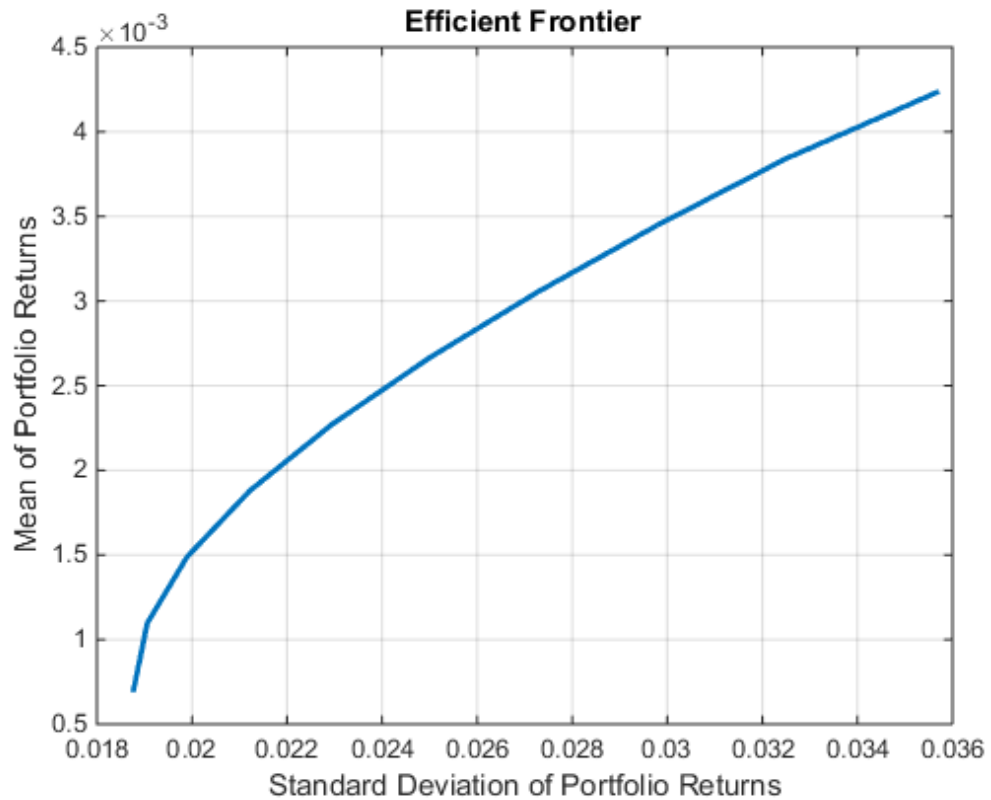
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

	Port1	Port2	Port3	Port4	Port5
AAPL	0.017926	0.058247	0.097816	0.12955	0
AMZN	0	0	0	0	0
CSCO	0	0	0	0	0
DELL	0.0041906	0	0	0	0
EBAY	0	0	0	0	0
GOOG	0.16144	0.35678	0.55228	0.75116	1
HPQ	0.052566	0.032302	0.011186	0	0
IBM	0.46422	0.36045	0.25577	0.11928	0
INTC	0	0	0	0	0
MSFT	0.29966	0.19222	0.082949	0	0
ORCL	0	0	0	0	0
YHOO	0	0	0	0	0



- “Constructing the Portfolio Object” on page 4-26
- “Class Attributes”
- “Property Attributes”

References

For a complete list of references for the Portfolio object and portfolio optimization tools, see “Portfolio Optimization”.

See Also

`plotFrontier`

More About

- “Portfolio Optimization Theory” on page 4-2
- “Portfolio Object” on page 4-17

PortfolioCVaR class

Superclasses: AbstractPortfolio

PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis

Description

The PortfolioCVaR object implements conditional value-at-risk (CVaR) portfolio optimization and is derived from the abstract portfolio optimization class `AbstractPortfolio`. This object implements all methods in the `AbstractPortfolio` class along with methods that are specific to CVaR portfolio optimization.

The main workflow for CVaR portfolio optimization is to create an instance of a `PortfolioCVaR` object that completely specifies a portfolio optimization problem and to operate on the `PortfolioCVaR` object to obtain and analyze efficient portfolios. A CVaR optimization problem is completely specified with these four elements:

- A universe of assets with scenarios of asset total returns for a period of interest, where scenarios comprise a collection of samples from the underlying probability distribution for asset total returns. This collection must be sufficiently large for asymptotic convergence of sample statistics. Note that asset return moments and related statistics are derived exclusively from the scenarios.
- A portfolio set that specifies the set of portfolio choices in terms of a collection of constraints.
- A model for portfolio return and risk proxies, which, for CVaR optimization, is either the gross or net mean of portfolio returns and the conditional value-at-risk of portfolio returns.
- A probability level that specifies the probability that a loss is less than or equal to the value-at-risk. Typical values are 0.9 and 0.95, which indicate 10% and 5% loss probabilities.

After these four elements have been specified in an unambiguous way, it is possible to solve and analyze CVaR portfolio optimization problems.

The simplest CVaR portfolio optimization problem has:

- Scenarios of asset total returns

- A requirement that all portfolios have nonnegative weights that sum to 1 (the summation constraint is known as a budget constraint)
- Built-in models for portfolio return and risk proxies that use scenarios of asset total returns
- A probability level of 0.95

Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified by:

```
p = PortfolioCVaR('Scenarios', AssetScenarios, 'LowerBound', 0, 'Budget', 1, ...
'ProbabilityLevel', 0.95);
```

or equivalently by:

```
p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);
```

To confirm that this is a valid portfolio optimization problem, the following method determines whether the set of PortfolioCVaR choices is bounded (a necessary condition for portfolio optimization).

```
[lb, ub, isbounded] = estimateBounds(p);
```

Given the problem specified in the PortfolioCVaR object `p`, the efficient frontier for this problem can be displayed with:

```
plotFrontier(p);
```

and efficient portfolios can be obtained with:

```
pwgt = estimateFrontier(p);
```

Construction

`p = PortfolioCVaR` constructs an empty PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis. You can then add elements to the PortfolioCVaR object using the supported add and set methods. For more information, see “Constructing the PortfolioCVaR Object”.

`p = PortfolioCVaR(Name, Value)` constructs a PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis with additional options specified by one or more Name, Value pair arguments. **Name** is a property name and **Value** is its

corresponding value. **Name** must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as **Name1, Value1, . . . , NameN, ValueN**.

p = PortfolioCVaR(p, Name, Value) constructs a PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis using a previously constructed PortfolioCVaR object **p** with additional options specified by one or more **Name, Value** pair arguments.

Input Arguments

p

(Optional) Previously constructed CVaR portfolio object (**p**).

Property Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name, Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1, Value1, . . . , NameN, ValueN**.

'AInequality'

Linear inequality constraint matrix ([] or [matrix]).

Default: []

'AssetList'

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Default: []

'bInequality'

Linear inequality constraint vector ([] or [vector]).

Default: []

'BuyCost'

Proportional purchase costs ([] or vector).

Default: []

'BuyTurnover'

Turnover constraint on purchases ([] or [scalar]).

Default: []

'GroupA'

Group A weights to be bounded by weights in group B ([] or [matrix]).

Default: []

'GroupB'

Group B weights ([] or [matrix]).

Default: []

'GroupMatrix'

Group membership matrix ([] or [matrix]).

Default: []

'InitPort'

Initial portfolio ([] or vector).

Default: []

'LowerBudget'

Lower-bound budget constraint ([] or [scalar]).

Default: []

'LowerGroup'

Lower-bound group constraint ([] or [vector]).

Default: []

'LowerRatio'

Minimum ratio of allocations between groups A and B ([] or [vector]).

Default: []

'Name'

Name for instance of the PortfolioCVaR object ([] or [string]).

Default: []

'NumAssets'

Number of assets in the universe ([] or [integer scalar]).

Default: []

'NumScenarios'

Number of scenarios ([] or [integer scalar]).

Note: Scenarios cannot be assigned directly to a PortfolioCVaR object. Scenarios must always be set through either the `PortfolioCVaR` constructor, the `setScenarios` method, or any of the scenario simulation methods. For more information, see “Constructing the PortfolioCVaR Object” on page 5-26.

Default: []

'ProbabilityLevel'

Probability level which is 1 minus the probability of losses greater than the value-at-risk ([] or [scalar]).

Default: []

'RiskFreeRate'

Risk-free rate ([] or scalar).

Default: []

'SellCost'

Proportional sales costs ([] or vector).

Default: []

'SellTurnover'

Turnover constraint on sales ([] or [scalar]).

Default: []

'Turnover'

Turnover constraint ([] or [scalar]).

Default: []

'UpperBound'

Upper-bound constraint ([] or [vector]).

Default: []

'UpperBudget'

Upper-bound budget constraint ([] or [scalar]).

Default: []

'UpperGroup'

Upper-bound group constraint ([] or [vector]).

Default: []

'UpperRatio'

Maximum ratio of allocations between groups A and B ([] or [vector]).

Default: []

Properties

The following properties are from the `PortfolioCVaR` class.

BuyCost

Proportional purchase costs ([] or vector).

Attributes:

SetAccess	public
GetAccess	public

BuyTurnover

Turnover constraint on purchases ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

NumScenarios

Number of scenarios ([] or [integer scalar]).

Attributes:

SetAccess	private
GetAccess	public

ProbabilityLevel

Value-at-risk probability level which is $1 - (\text{loss probability})$ ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

RiskFreeRate

Risk-free rate ([] or scalar).

Attributes:

SetAccess	public
GetAccess	public

SellCost

Proportional sales costs ([] or vector).

Attributes:

SetAccess	public
GetAccess	public

SellTurnover

Turnover constraint on sales ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

Turnover

Turnover constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

Properties

The following properties are inherited from the `AbstractPortfolio` class.

AEquality

Linear equality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

AInequality

Linear inequality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

AssetList

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Attributes:

SetAccess	public
GetAccess	public

bEquality

Linear equality constraint vector ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

bInequality

Linear inequality constraint vector ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

GroupA

Group A weights to be bounded by group B ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

GroupB

Group B weights ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

GroupMatrix

Group membership matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

InitPort

Initial portfolio ([] or vector).

Attributes:

SetAccess	public
GetAccess	public

LowerBound

Lower-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerBudget

Lower-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

LowerGroup

Lower-bound group constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerRatio

Minimum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

Name

Name for instance of the PortfolioCVaR object ([] or [string]).

Attributes:

SetAccess	public
GetAccess	public

NumAssets

Number of assets in the universe ([] or [integer scalar]).

Attributes:

SetAccess	public
GetAccess	public

UpperBound

Upper-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

UpperBudget

Upper-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

UpperGroup

Upper-bound group constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

UpperRatio

Maximum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

Inherited Methods

The following methods are inherited from the `AbstractPortfolio` class.

<code>addEquality</code>	Add equality constraints for portfolio weights to existing constraints.
<code>addGroupRatio</code>	Add group ratio constraints for portfolio weights to existing constraints.
<code>addGroups</code>	Add group constraints for portfolio weights to existing constraints.
<code>addInequality</code>	Add inequality constraints for portfolio weights to existing constraints.
<code>checkFeasibility</code>	Determine if portfolios are members of the set of feasible portfolios.
<code>estimateBounds</code>	Determine if set of feasible portfolios is nonempty and bounded.
<code>estimateFrontier</code>	Estimate portfolios on the entire efficient frontier.
<code>estimateFrontierByReturn</code>	Estimate portfolios on the efficient frontier with targeted returns or return proxies.
<code>estimateFrontierByRisk</code>	Estimate portfolios on the efficient frontier with targeted risks or risk proxies.
<code>estimateFrontierLimits</code>	Estimate portfolios at the extreme ends of the efficient frontier (minimum risk and maximum return).
<code>estimatePortReturn</code>	Estimate return or return proxy for specified portfolios.
<code>estimatePortRisk</code>	Estimate risk or risk proxy for specified portfolios.
<code>getBounds</code>	Get lower and upper bounds from the object.
<code>getBudget</code>	Get lower and upper budget constraints from the object.
<code>getEquality</code>	Get equality constraint matrix and vector from the object.

<code>getGroupRatio</code>	Get base matrix, comparison matrix, and lower and upper bounds for group ratio constraints from the object.
<code>getGroups</code>	Get group matrix and lower and upper bounds for group constraints from the object.
<code>getInequality</code>	Get inequality constraint matrix and vector from the object.
<code>plotFrontier</code>	Plot efficient frontier and optionally obtain risks and returns for portfolios on the efficient frontier.
<code>setAssetList</code>	Set up a list of asset names and symbols to be associated with assets in the universe.
<code>setBounds</code>	Set up lower and upper bounds for portfolio weights.
<code>setBudget</code>	Set up lower and upper budget constraints for portfolio weights.
<code>setDefaultConstraints</code>	Set up default constraints for portfolio weights (nonnegative weights that must sum to 1).
<code>setEquality</code>	Set up equality constraints for portfolio weights.
<code>setGroupRatio</code>	Set up group ratio constraints for portfolio weights.
<code>setGroups</code>	Set up group constraints for portfolio weights.
<code>setInequality</code>	Set up inequality constraints for portfolio weights.
<code>setInitPort</code>	Set up initial portfolio weights.
<code>setOptions</code>	Set up hidden control properties in object (not implemented).
<code>setSolver</code>	Set up solver to estimate efficient portfolios.

Methods

<code>estimatePortStd</code>	Estimate standard deviation of portfolio returns.
<code>estimatePortVaR</code>	Estimate value-at-risk for portfolio.

<code>estimateScenarioMoments</code>	Estimate mean and covariance of scenarios.
<code>simulateNormalScenariosByData</code>	Simulate multivariate normal asset return scenarios from data.
<code>simulateNormalScenariosByMoments</code>	Simulate multivariate normal asset return scenarios from a mean and covariance of asset returns.
<code>getCosts</code>	Get purchase and sales proportional transaction costs from the object.
<code>getOneWayTurnover</code>	Get one-way portfolio turnover constraints.
<code>getScenarios</code>	Obtain scenarios from PortfolioCVaR object.
<code>setCosts</code>	Set up purchase and sale proportional transaction costs for assets in the universe.
<code>setOneWayTurnover</code>	Set up one-way portfolio turnover constraints.
<code>setProbabilityLevel</code>	Set probability level for VaR and CVaR calculations.
<code>setScenarios</code>	Set asset returns scenarios by direct matrix.
<code>setTurnover</code>	Set up average turnover constraints for portfolio weights.

Definitions

Conditional Value-at-Risk Portfolio Optimization

For more information on the theory and definition of conditional value-at-risk optimization supported by portfolio optimization tools in Financial Toolbox, see “Portfolio Optimization Theory” on page 5-2.

Copy Semantics

Value. To learn how value classes affect copy operations, see “Copying Objects” in the MATLAB documentation.

Examples

Construct a PortfolioCVaR Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = PortfolioCVaR('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000, 'missingdata',true);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);

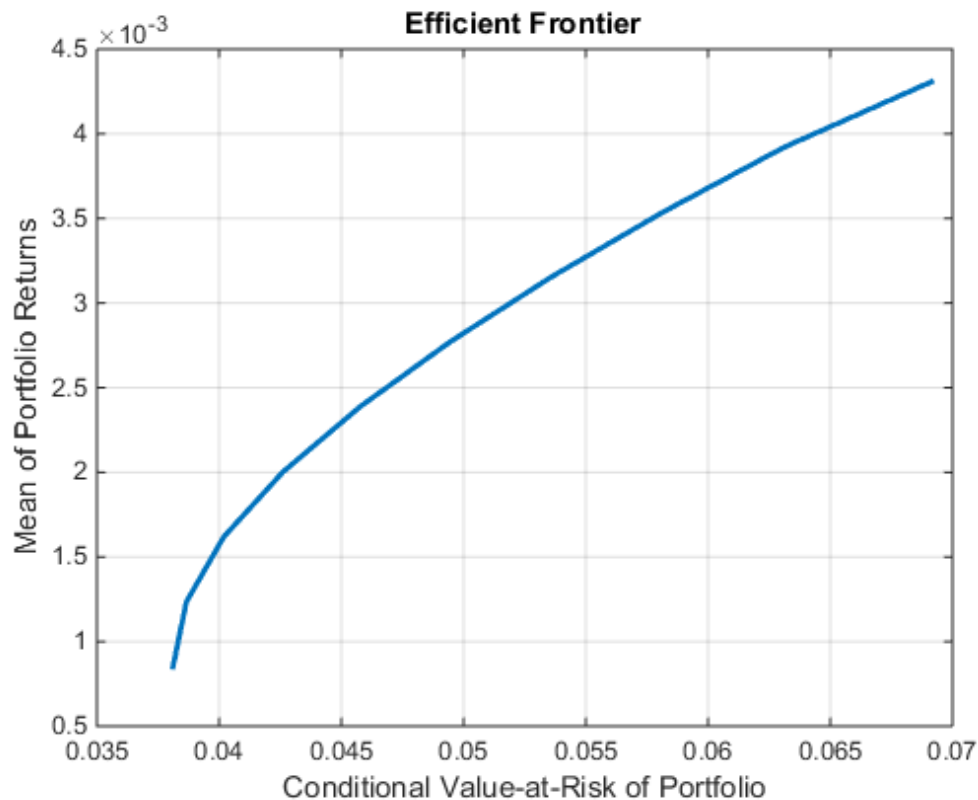
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames], 'obsnames', p.AssetList);

disp(Blotter);
```

	Port1	Port2	Port3	Port4	Port5
AAPL	0.010984	0.073246	0.11933	0.13068	1.5092e-14
AMZN	4.702e-38	3.0815e-33	4.702e-38	2.9398e-32	2.8997e-14
CSCO	0	4.6222e-33	1.1755e-38	1.2326e-32	4.1869e-14
DELL	0.022454	1.3193e-32	5.0693e-38	1.9706e-48	3.9048e-14
EBAY	0	0	4.1142e-38	5.2385e-32	1.3394e-15
GOOG	0.20335	0.38055	0.56242	0.75932	1
HPQ	0.041724	0.0099223	0	8.9376e-33	3.8894e-14
IBM	0.44482	0.36453	0.26282	0.11	3.7902e-14
INTC	0	0	1.4106e-37	0	3.8264e-14
MSFT	0.27667	0.17175	0.055435	0	4.0873e-14
ORCL	1.4694e-38	1.4448e-32	2.351e-38	8.4071e-33	3.7811e-14
YHOO	0	0	1.1755e-38	1.04e-32	3.535e-14



- “Constructing the PortfolioCVaR Object” on page 5-26
- “Class Attributes”
- “Property Attributes”

References

For a complete list of references for the PortfolioCVaR object and portfolio optimization tools, see “Portfolio Optimization”.

See Also

`plotFrontier`

More About

- “Portfolio Optimization Theory” on page 5-2
- “PortfolioCVaR Object” on page 5-17

PortfolioMAD class

Superclasses: AbstractPortfolio

PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis

Description

The PortfolioMAD object implements mean-absolute deviation portfolio optimization, where MAD stands for “mean-absolute deviation.” This type of portfolio optimization is usually called MAD portfolio optimization and is derived from the abstract portfolio optimization class `AbstractPortfolio`. This object implements all methods in the `AbstractPortfolio` class along with methods that are specific to MAD portfolio optimization.

The main workflow for MAD portfolio optimization is to create an instance of a PortfolioMAD object that completely specifies a portfolio optimization problem and to operate on the PortfolioMAD object to obtain and analyze efficient portfolios. A MAD optimization problem is completely specified with these three elements:

- A universe of assets with scenarios of asset total returns for a period of interest, where scenarios comprise a collection of samples from the underlying probability distribution for asset total returns. This collection must be sufficiently large for asymptotic convergence of sample statistics. Note that asset return moments and related statistics are derived exclusively from the scenarios.
- A portfolio set that specifies the set of portfolio choices in terms of a collection of constraints.
- A model for portfolio return and risk proxies, which, for MAD optimization, is either the gross or net mean of portfolio returns and the mean-absolute deviation of portfolio returns.

After these three elements have been specified unambiguously, it is possible to solve and analyze MAD portfolio optimization problems.

The simplest MAD portfolio optimization problem has:

- Scenarios of asset total returns
- A requirement that all portfolios have nonnegative weights that sum to 1 (the summation constraint is known as a budget constraint)

- Built-in models for portfolio return and risk proxies that use scenarios of asset total returns

Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified by:

```
p = PortfolioMAD('Scenarios', AssetScenarios, 'LowerBound', 0, 'Budget', 1);
or equivalently by:
```

```
p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

To confirm that this is a valid portfolio optimization problem, the following method determines whether the set of PortfolioMAD choices is bounded (a necessary condition for portfolio optimization).

```
[lb, ub, isbounded] = estimateBounds(p);
```

Given the problem specified in the PortfolioMAD object `p`, the efficient frontier for this problem can be displayed with:

```
plotFrontier(p);
and efficient portfolios can be obtained with:
pwgt = estimateFrontier(p);
```

Construction

`p = PortfolioMAD` constructs an empty PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis. You can then add elements to the PortfolioMAD object using the supported `add` and `set` methods. For more information, see “Constructing the PortfolioCVaR Object”.

`p = PortfolioMAD(Name, Value)` constructs a PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis with additional options specified by one or more `Name, Value` pair arguments. `Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

`p = PortfolioMAD(p, Name, Value)` constructs a PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis using a previously constructed

PortfolioMAD object `p` with additional options specified by one or more Name,Value pair arguments.

Input Arguments

p

(Optional) Previously constructed MAD portfolio object (`p`).

Property Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, . . . ,NameN,ValueN.

'AInequality'

Linear inequality constraint matrix ([] or [matrix]).

Default: []

'AssetList'

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Default: []

'bInequality'

Linear inequality constraint vector ([] or [vector]).

Default: []

'BuyCost'

Proportional purchase costs ([] or vector).

Default: []

'BuyTurnover'

Turnover constraint on purchases ([] or [scalar]).

Default: []

'GroupA'

Group A weights to be bounded by weights in group B ([] or [matrix]).

Default: []

'GroupB'

Group B weights ([] or [matrix]).

Default: []

'GroupMatrix'

Group membership matrix ([] or [matrix]).

Default: []

'InitPort'

Initial portfolio ([] or vector).

Default: []

'LowerBudget'

Lower-bound budget constraint ([] or [scalar]).

Default: []

'LowerGroup'

Lower-bound group constraint ([] or [vector]).

Default: []

'LowerRatio'

Minimum ratio of allocations between groups A and B ([] or [vector]).

Default: []

'Name'

Name for instance of the PortfolioMAD object ([] or [string]).

Default: []

'NumAssets'

Number of assets in the universe ([] or [integer scalar]).

Note: Scenarios cannot be assigned directly to a PortfolioMAD object. Scenarios must always be set through either the PortfolioMAD constructor, the setScenarios method, or any of the scenario simulation methods. For more information, see “Constructing the PortfolioMAD Object”.

Default: []

'NumScenarios'

Number of scenarios ([] or [integer scalar]).

Default: []

'RiskFreeRate'

Risk-free rate ([] or scalar).

Default: []

'SellCost'

Proportional sales costs ([] or vector).

Default: []

'SellTurnover'

Turnover constraint on sales ([] or [scalar]).

Default: []

'Turnover'

Turnover constraint ([] or [scalar]).

Default: []

'UpperBound'

Upper-bound constraint ([] or [vector]).

Default: []

'UpperBudget'

Upper-bound budget constraint ([] or [scalar]).

Default: []

'UpperGroup'

Upper-bound group constraint ([] or [vector]).

Default: []

'UpperRatio'

Maximum ratio of allocations between groups A and B ([] or [vector]).

Default: []

Properties

The following properties are from the PortfolioCVaR class.

BuyCost

Proportional purchase costs ([] or vector).

Attributes:

SetAccess public

GetAccess public

BuyTurnover

Turnover constraint on purchases ([] or [scalar]).

Attributes:

SetAccess public

GetAccess public

NumScenarios

Number of scenarios ([] or [integer scalar]).

Attributes:

SetAccess private

GetAccess public

RiskFreeRate

Risk-free rate ([] or scalar).

Attributes:

SetAccess public

GetAccess public

SellCost

Proportional sales costs ([] or vector).

Attributes:

SetAccess public

GetAccess public

SellTurnover

Turnover constraint on sales ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

Turnover

Turnover constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

Properties

The following properties are inherited from the `AbstractPortfolio` class.

AEquality

Linear equality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

AInequality

Linear inequality constraint matrix ([] or [matrix]).

Attributes:

SetAccess	public
-----------	--------

GetAccess public

AssetList

Names or symbols of assets in the universe ([] or [vector cell of strings]).

Attributes:

SetAccess public

GetAccess public

bEquality

Linear equality constraint vector ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

bInequality

Linear inequality constraint vector ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

GroupA

Group A weights to be bounded by group B ([] or [matrix]).

Attributes:

SetAccess public

GetAccess public

GroupB

Group B weights ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

GroupMatrix

Group membership matrix ([] or [matrix]).

Attributes:

SetAccess	public
GetAccess	public

InitPort

Initial portfolio ([] or vector).

Attributes:

SetAccess	public
GetAccess	public

LowerBound

Lower-bound constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerBudget

Lower-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess	public
GetAccess	public

LowerGroup

Lower-bound group constraint ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

LowerRatio

Minimum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess	public
GetAccess	public

Name

Name for instance of the PortfolioMAD object ([] or [string]).

Attributes:

SetAccess	public
GetAccess	public

NumAssets

Number of assets in the universe ([] or [integer scalar]).

Attributes:

SetAccess	public
-----------	--------

GetAccess public

UpperBound

Upper-bound constraint ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

UpperBudget

Upper-bound budget constraint ([] or [scalar]).

Attributes:

SetAccess public

GetAccess public

UpperGroup

Upper-bound group constraint ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

UpperRatio

Maximum ratio of allocations between groups A and B ([] or [vector]).

Attributes:

SetAccess public

GetAccess public

Inherited Methods

The following methods are inherited from the `AbstractPortfolio` class.

<code>addEquality</code>	Add equality constraints for portfolio weights to existing constraints.
<code>addGroupRatio</code>	Add group ratio constraints for portfolio weights to existing constraints.
<code>addGroups</code>	Add group constraints for portfolio weights to existing constraints.
<code>addInequality</code>	Add inequality constraints for portfolio weights to existing constraints.
<code>checkFeasibility</code>	Determine if portfolios are members of the set of feasible portfolios.
<code>estimateBounds</code>	Determine if the set of feasible portfolios is nonempty and bounded.
<code>estimateFrontier</code>	Estimate portfolios on the entire efficient frontier.
<code>estimateFrontierByReturn</code>	Estimate portfolios on the efficient frontier with targeted returns or return proxies.
<code>estimateFrontierByRisk</code>	Estimate portfolios on the efficient frontier with targeted risks or risk proxies.
<code>estimateFrontierLimits</code>	Estimate portfolios at the extreme ends of the efficient frontier (minimum risk and maximum return).
<code>estimatePortReturn</code>	Estimate the return or return proxy for specified portfolios.
<code>estimatePortRisk</code>	Estimate the risk or risk proxy for specified portfolios.
<code>getBounds</code>	Get lower and upper bounds from the object.
<code>getBudget</code>	Get lower and upper budget constraints from the object.
<code>getEquality</code>	Get equality constraint matrix and vector from the object.

<code>getGroupRatio</code>	Get base matrix, comparison matrix, and lower and upper bounds for group ratio constraints from the object.
<code>getGroups</code>	Get group matrix and lower and upper bounds for group constraints from the object.
<code>getInequality</code>	Get inequality constraint matrix and vector from the object.
<code>plotFrontier</code>	Plot efficient frontier and optionally obtain risks and returns for portfolios on the efficient frontier.
<code>setAssetList</code>	Set up a list of asset names and symbols to be associated with assets in the universe.
<code>setBounds</code>	Set up lower and upper bounds for portfolio weights.
<code>setBudget</code>	Set up lower and upper budget constraints for portfolio weights.
<code>setDefaultConstraints</code>	Set up default constraints for portfolio weights (nonnegative weights that must sum to 1).
<code>setEquality</code>	Set up equality constraints for portfolio weights.
<code>setGroupRatio</code>	Set up group ratio constraints for portfolio weights.
<code>setGroups</code>	Set up group constraints for portfolio weights.
<code>setInequality</code>	Set up inequality constraints for portfolio weights.
<code>setInitPort</code>	Set up initial portfolio weights.
<code>setOptions</code>	Set up hidden control properties in object (not implemented).
<code>setSolver</code>	Set up solver to estimate efficient portfolios.

Methods

<code>estimatePortStd</code>	Estimate standard deviation of portfolio returns.
<code>estimateScenarioMoments</code>	Estimate mean and covariance of scenarios.

<code>simulateNormalScenariosByData</code>	Simulate multivariate normal asset return scenarios from data.
<code>simulateNormalScenariosByMoments</code>	Simulate multivariate normal asset return scenarios from a mean and covariance of asset returns.
<code>getCosts</code>	Get purchase and sales proportional transaction costs from the object.
<code>getOneWayTurnover</code>	Get one-way portfolio turnover constraints.
<code>getScenarios</code>	Obtain scenarios from <code>PortfolioMAD</code> object.
<code>setCosts</code>	Set up purchase and sale proportional transaction costs for assets in the universe.
<code>setOneWayTurnover</code>	Set up one-way portfolio turnover constraints.
<code>setScenarios</code>	Set asset returns scenarios by direct matrix.
<code>setTurnover</code>	Set up average turnover constraints for portfolio weights.

Definitions

Mean-Absolute Deviation Portfolio Optimization

For more information on the theory and definition of mean-absolute deviation (MAD) optimization supported by portfolio optimization tools in Financial Toolbox, see “Portfolio Optimization Theory” on page 5-2.

Copy Semantics

Value. To learn how value classes affect copy operations, see “Copying Objects” in the MATLAB documentation.

Examples

Construct a `PortfolioMAD` Object and Determine Efficient Portfolios

Create efficient portfolios:


```

load CAPMuniverse

p = PortfolioMAD('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 , 'missingdata',true);
p = setDefaultConstraints(p);

plotFrontier(p);

pwgt = estimateFrontier(p, 5);

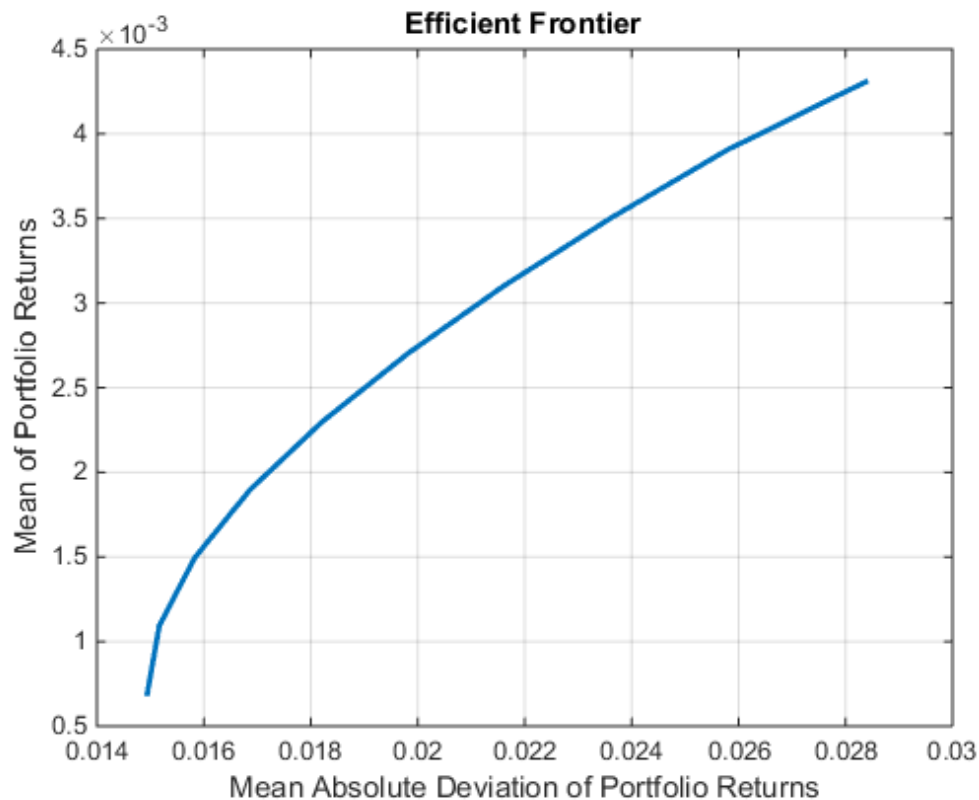
pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([pwgt],pnames,'obsnames',p.AssetList);

disp(Blotter);

```

	Port1	Port2	Port3	Port4	Port5
AAPL	0.030236	0.075387	0.11278	0.13456	1.5092e-14
AMZN	1.5693e-21	0	0	3.8519e-34	2.8997e-14
CSCO	7.7592e-23	0	4.4077e-23	0	4.1869e-14
DELL	0.0089659	0	6.3609e-23	1.4444e-34	3.9048e-14
EBAY	8.2218e-22	6.8538e-23	4.2239e-23	7.4575e-17	1.3394e-15
GOOG	0.16117	0.35201	0.54486	0.74888	1
HPQ	0.056551	0.024037	1.2732e-22	0	3.8894e-14
IBM	0.45905	0.37891	0.29383	0.11656	3.7902e-14
INTC	0	6.399e-22	4.4988e-24	5.3234e-17	3.8264e-14
MSFT	0.28403	0.16966	0.048527	0	4.0873e-14
ORCL	0	0	0	-2.4475e-34	3.7811e-14
YHOO	0	1.0943e-22	0	0	3.535e-14



- “Constructing the PortfolioMAD Object”
- “Class Attributes”
- “Property Attributes”

References

For a complete list of references for the PortfolioMAD object and portfolio optimization tools, see “Portfolio Optimization”.

See Also

`plotFrontier`

More About

- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object” on page 6-17

portopt

Portfolios on constrained efficient frontier

As an alternative to `portopt`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow”.

Syntax

```
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn, ExpCovariance,
NumPorts, PortReturn, ConSet, varargin)
```

Arguments

ExpReturn	1 by number of assets (NASSETS) vector specifying the expected (mean) return of each asset.
ExpCovariance	NASSETS-by-NASSETS matrix specifying the covariance of the asset returns.
NumPorts	(Optional) Number of portfolios generated along the efficient frontier. Returns are equally spaced between the maximum possible return and the minimum risk point. If NumPorts is empty (entered as []), computes 10 equally spaced points.
PortReturn	(Optional) Expected return of each portfolio. A number of portfolios (NPORTS) by 1 vector. If not entered or empty, NumPorts equally spaced returns between the minimum and maximum possible values are used.
ConSet	(Optional) Constraint matrix for a portfolio of asset investments, created using portcons. If not specified, a default is created.
varargin	(Optional) varargin supports the following parameter-value pairs:

- 'algorithm' – Defines which algorithm to use with portopt. Use either a value of 'lcprog' or 'quadprog' to indicate the algorithm to use. The default is 'lcprog'.
- 'maxiter' – Maximum number of iterations before termination of algorithm. The default is 100000.
- 'tiebreak' – Method to break ties for pivot selection. This value pair applies only to 'lcprog' algorithm. The default is 'first'. Options are:
 - 'first' – Selects pivot with lowest index.
 - 'last' – Selects pivot with highest index.
 - 'random' – Selects pivot at random.
- 'tolcon' – Tolerance for constraint violations. This value pair applies only to 'lcprog' algorithm. The default is 1.0e-6.
- 'tolpiv' – Pivot value below which a number is considered to be zero. This value pair applies only to 'lcprog' algorithm. The default is 1.0e-9.

Description

[PortRisk, PortReturn, PortWts] = portopt(ExpReturn, ExpCovariance, NumPorts, PortReturn, ConSet, varargin) returns the mean-variance efficient frontier with user-specified covariance, returns, and asset constraints (ConSet). Given a collection of NASSETS risky assets, computes a portfolio of asset investment weights that minimize the risk for given values of the expected return. The portfolio risk is minimized subject to constraints on the total portfolio value, the individual asset minimum and maximum allocation, the asset group minimum and maximum allocation, or the asset group-to-group comparison.

PortRisk is an NPORTS-by-1 vector of the standard deviation of each portfolio.

PortReturn is an NPORTS-by-1 vector of the expected return of each portfolio.

PortWts is an NPORTS-by-NASSETS matrix of weights allocated to each asset. Each row represents a portfolio. The total of all weights in a portfolio is 1.

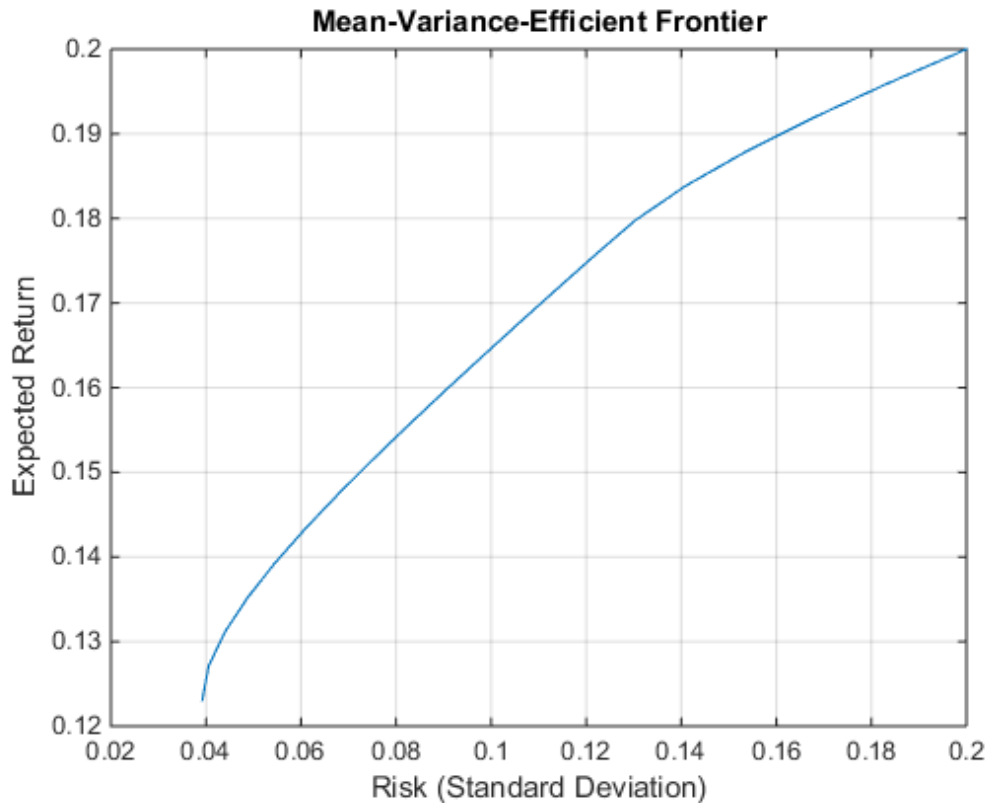
If `portopt` is invoked without output arguments, it returns a plot of the efficient frontier.

Examples

Plot the Risk-Return Efficient Frontier

This example shows how to plot the risk-return efficient frontier of portfolios allocated among three assets. Connect 20 portfolios along the frontier having evenly spaced returns. By default, choose among portfolios without short-selling and scale the value of the portfolio to 1.

```
ExpReturn = [0.1 0.2 0.15];  
  
ExpCovariance = [0.005  -0.010  0.004  
                 -0.010  0.040  -0.002  
                 0.004  -0.002  0.023];  
  
NumPorts = 20;  
portopt(ExpReturn, ExpCovariance, NumPorts)
```



Return the two efficient portfolios that have returns of 16% and 17%. Limit to portfolios that have at least 20% of the allocation in the first asset, and cap the total value in the first and third assets at 50% of the portfolio.

```
ExpReturn = [0.1 0.2 0.15];
```

```
ExpCovariance = [0.005  -0.010  0.004
                 -0.010  0.040  -0.002
                  0.004  -0.002  0.023];
```

```
PortReturn = [0.16
              0.17];
```

```
NumAssets = 3;
```

```
AssetMin = [0.20 NaN NaN];  
Group    = [1    0    1];  
GroupMax = 0.50;  
  
ConSet = portcons('Default', NumAssets, 'AssetLims', AssetMin,...  
NaN, 'GroupLims', Group, NaN, GroupMax);  
  
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn,...  
ExpCovariance, [], PortReturn, ConSet)  
  
PortRisk =  
  
    0.0919  
    0.1138  
  
PortReturn =  
  
    0.1600  
    0.1700  
  
PortWts =  
  
    0.3000    0.5000    0.2000  
    0.2000    0.6000    0.2000
```

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

ewstats | frontcon | frontier | portcons | portstats

portrand

Randomized portfolio risks, returns, and weights

Syntax

```
[PortRisk, PortReturn, PortWts] = portrand(Asset, Return,
Points, Method)
```

```
portrand(Asset, Return, Points, Method)
```

Arguments

Asset	Matrix of time series data. Each row is an observation and each column represents a single security.
Return	(Optional) Row vector where each column represents the rate of return for the corresponding security in Asset . By default, Return is computed by taking the average value of each column of Asset .
Points	(Optional) Scalar that specifies how many random points should be generated. Default = 1000.
Method	<p>(Optional) A string that specifies how to generate random portfolios from the set of portfolios with two possible methods:</p> <ul style="list-style-type: none"> • 'uniform' – Uniformly-distributed portfolio weights (default method). The 'uniform' method generates portfolio weights that are uniformly-distributed on the set of portfolio weights. • 'geometric' – Concentrated portfolio weights around the geometric center of the set of portfolios. The 'geometric' method generates portfolio weights that are concentrated around the geometric center of the set of portfolio weights. <p>Note: The 'uniform' and 'geometric' methods generate weights that are distributed symmetrically around the geometric center of the set of weights.</p>

Description

[PortRisk, PortReturn, PortWts] = portrand(Asset, Return, Points, Method) returns the risks, rates of return, and weights of random portfolio configurations.

PortRisk	Points-by-1 vector of standard deviations.
PortReturn	Points-by-1 vector of expected rates of return.
PortWts	Points by number of securities matrix of asset weights. Each row of PortWts is a different portfolio configuration.

portrand(Asset, Return, Points, Method) plots the points representing each portfolio configuration. It does not return any data to the MATLAB workspace.

Note: Portfolios are selected at random from a set of portfolios such that portfolio weights are nonnegative and sum to 1. The sample mean and covariance of asset returns are used to compute portfolio returns for each random portfolio.

References

Bodie, Kane, and Marcus, *Investments*, Chapter 7.

See Also

frontcon | portror | portvar

portror

Portfolio expected rate of return

Syntax

```
R = portror(Return, Weight)
```

Arguments

Return	1-by-N matrix of rates of return. Each column of Return represents the rate of return for a single security
Weight	M-by-N matrix of weights. Each row of Weight represents a different weighting combination of the assets in the portfolio.

Description

`R = portror(Return, Weight)` returns a 1-by-M vector for the expected rate of return.

Examples

Portfolio Expected Rate of Return

This example shows a portfolio that is made up of two assets ABC and XYZ having expected rates of return of 10% and 14%, respectively. If 40% percent of the portfolio's funds are allocated to asset ABC and the remaining funds are allocated to asset XYZ, the portfolio's expected rate of return is:

```
r = portror([.1 .14],[.4 .6])
```

```
r =
```

```
0.1240
```

References

Bodie, Kane, and Marcus, *Investments*, Chapter 7.

See Also

frontcon | portrand | portvar

portsim

Monte Carlo simulation of correlated asset returns

Syntax

```
RetSeries = portsim(ExpReturn, ExpCovariance, NumObs, RetIntervals,
NumSim, Method)
```

Arguments

ExpReturn	1 by number of assets (NASSETS) vector specifying the expected (mean) return of each asset.
ExpCovariance	NASSETS-by-NASSETS matrix of asset return covariances. ExpCovariance must be symmetric and positive semidefinite (no negative eigenvalues). The standard deviations of the returns are $\text{ExpSigma} = \text{sqrt}(\text{diag}(\text{ExpCovariance}))$.
NumObs	Positive scalar integer indicating the number of consecutive observations in the return time series. If NumObs is entered as the empty matrix [], the length of RetIntervals is used.
RetIntervals	(Optional) Positive scalar or number of observations (NUMOBS)-by-1 vector of interval times between observations. If RetIntervals is not specified, all intervals are assumed to have length 1.
NumSim	(Optional) Positive scalar integer indicating the number of simulated sample paths (realizations) of NUMOBS observations. Default = 1 (single realization of NUMOBS correlated asset returns).
Method	(Optional) String indicating the type of Monte Carlo simulation: 'Exact' (default) generates correlated asset returns in which the sample mean and covariance match the input

	<p>mean (<code>ExpReturn</code>) and covariance (<code>ExpCovariance</code>) specifications.</p> <p>'Expected' generates correlated asset returns in which the sample mean and covariance are statistically equal to the input mean and covariance specifications. (The expected value of the sample mean and covariance are equal to the input mean (<code>ExpReturn</code>) and covariance (<code>ExpCovariance</code>) specifications.)</p> <p>For either method the sample mean and covariance returned are appropriately scaled by <code>RetIntervals</code>.</p>
--	--

Description

`portsim` simulates correlated returns of `NASSETS` assets over `NUMOBS` consecutive observation intervals. Asset returns are simulated as the proportional increments of constant drift, constant volatility stochastic processes, thereby approximating continuous-time geometric Brownian motion.

`RetSeries` is a `NUMOBS`-by-`NASSETS`-by-`NUMSIM` three-dimensional array of correlated, normally distributed, proportional asset returns. Asset returns over an interval of length dt are given by

$$\frac{dS}{S} = \mu dt + \sigma dz = \mu dt + \sigma \varepsilon \sqrt{dt},$$

where S is the asset price, μ is the expected rate of return, σ is the volatility of the asset price, and ε represents a random drawing from a standardized normal distribution.

Notes

- When *Method* is 'Exact', the sample mean and covariance of all realizations (scaled by `RetIntervals`) match the input mean and covariance. When the returns are subsequently converted to asset prices, all terminal prices for a given asset are in close agreement. Although all realizations are drawn independently, they produce similar terminal asset prices. Set *Method* to 'Expected' to avoid this behavior.

- The returns from the portfolios in `PortWts` are given by `PortReturn = PortWts * RetSeries(:, :, 1)'`, where `PortWts` is a matrix in which each row contains the asset allocations of a portfolio. Each row of `PortReturn` corresponds to one of the portfolios identified in `PortWts`, and each column corresponds to one of the observations taken from the first realization (the first plane) in `RetSeries`. See `portopt` and `portstats` for portfolio specification and optimization.
-

Examples

Distinction Between Simulation Methods

This example shows the distinction between the `Exact` and `Expected` methods of simulation.

Consider a portfolio of five assets with the following expected returns, standard deviations, and correlation matrix based on daily asset returns (where `ExpReturn` and `Sigmas` are divided by 100 to convert percentages to returns).

```
ExpReturn      = [0.0246  0.0189  0.0273  0.0141  0.0311]/100;
Sigmas         = [0.9509  1.4259  1.5227  1.1062  1.0877]/100;
Correlations   = [1.0000  0.4403  0.4735  0.4334  0.6855
                  0.4403  1.0000  0.7597  0.7809  0.4343
                  0.4735  0.7597  1.0000  0.6978  0.4926
                  0.4334  0.7809  0.6978  1.0000  0.4289
                  0.6855  0.4343  0.4926  0.4289  1.0000];
```

Convert the correlations and standard deviations to a covariance matrix.

```
ExpCovariance = corr2cov(Sigmas, Correlations)
```

```
ExpCovariance =
```

```
1.0e-03 *
    0.0904    0.0597    0.0686    0.0456    0.0709
    0.0597    0.2033    0.1649    0.1232    0.0674
    0.0686    0.1649    0.2319    0.1175    0.0816
    0.0456    0.1232    0.1175    0.1224    0.0516
```



```
0.0709    0.0674    0.0816    0.0516    0.1183
```

Assume that there are 252 trading days in a calendar year, and simulate two sample paths (realizations) of daily returns over a two-year period. Since `ExpReturn` and `ExpCovariance` are expressed daily, set `RetIntervals = 1`.

```
StartPrice = 100;
NumObs     = 504; % two calendar years of daily returns
NumSim     = 2;
RetIntervals = 1; % one trading day
NumAssets  = 5;
```

To illustrate the distinction between methods, simulate two paths by each method, starting with the same random number state.

```
rng('default');
RetExact = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Exact');
randn('state',0);
RetExpected = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected');
```

Compare the mean and covariance of `RetExact` with the inputs (`ExpReturn` and `ExpCovariance`), you will observe that they are almost identical.

At this point, `RetExact` and `RetExpected` are both 504-by-5-by-2 arrays. Now assume an equally weighted portfolio formed from the five assets and create arrays of portfolio returns in which each column represents the portfolio return of the corresponding sample path of the simulated returns of the five assets. The portfolio arrays `PortRetExact` and `PortRetExpected` are 504-by-2 matrices.

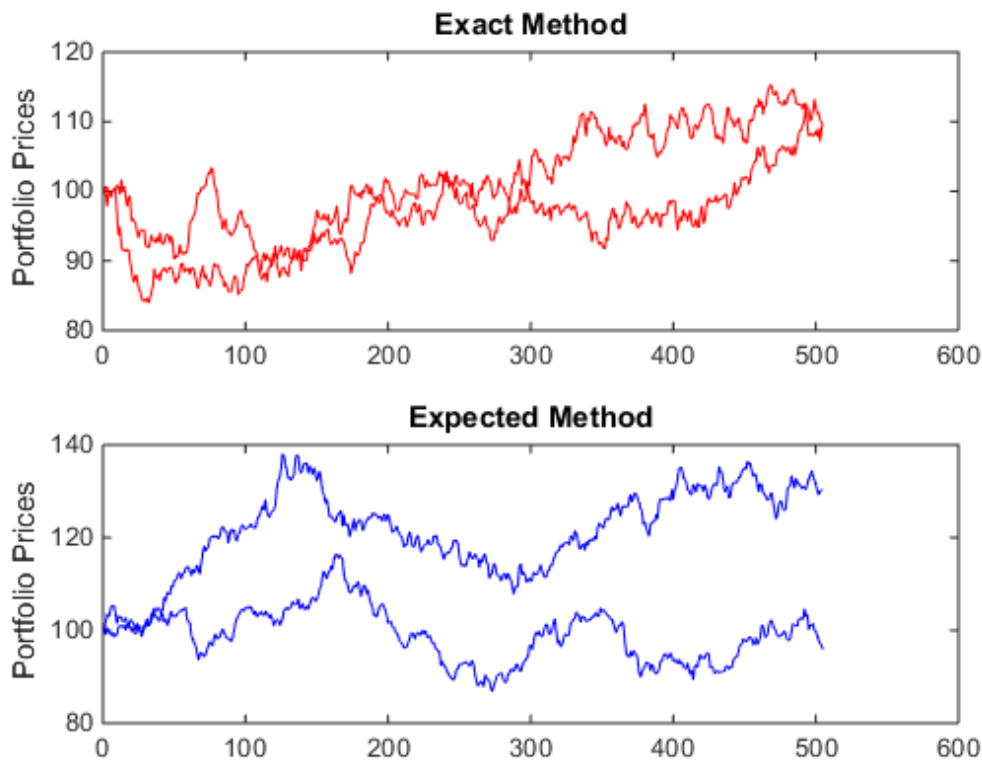
```
Weights = ones(NumAssets, 1)/NumAssets;
PortRetExact = zeros(NumObs, NumSim);
PortRetExpected = zeros(NumObs, NumSim);

for i = 1:NumSim
    PortRetExact(:,i) = RetExact(:, :, i) * Weights;
    PortRetExpected(:,i) = RetExpected(:, :, i) * Weights;
end
```

Finally, convert the simulated portfolio returns to prices and plot the data. In particular, note that since the `Exact` method matches expected return and covariance, the terminal

portfolio prices are virtually identical for each sample path. This is not true for the Expected simulation method. Although this example examines portfolios, the same methods apply to individual assets as well. Thus, Exact simulation is most appropriate when unique paths are required to reach the same terminal prices.

```
PortExact = ret2tick(PortRetExact, ...
    repmat(StartPrice,1,NumSim));
PortExpected = ret2tick(PortRetExpected, ...
    repmat(StartPrice,1,NumSim));
subplot(2,1,1), plot(PortExact, '-r')
ylabel('Portfolio Prices')
title('Exact Method')
subplot(2,1,2), plot(PortExpected, '-b')
ylabel('Portfolio Prices')
title('Expected Method')
```



Interaction Between ExpReturn, ExpCovariance and RetIntervals

This example shows the interplay among ExpReturn, ExpCovariance, and RetIntervals. Recall that portsim simulates correlated asset returns over an interval of length dt , given by the equation

$$\frac{dS}{S} = \mu dt + \sigma dz = \mu dt + \sigma \varepsilon \sqrt{dt},$$

where S is the asset price, μ is the expected rate of return, σ is the volatility of the asset price, and ε represents a random drawing from a standardized normal distribution.

The time increment dt is determined by the optional input `RetIntervals`, either as an explicit input argument or as a unit time increment by default. Regardless, the periodicity of `ExpReturn`, `ExpCovariance` and `RetIntervals` must be consistent. For example, if `ExpReturn` and `ExpCovariance` are annualized, then `RetIntervals` must be in years. This point is often misunderstood.

To illustrate the interplay among `ExpReturn`, `ExpCovariance`, and `RetIntervals`, consider a portfolio of five assets with the following expected returns, standard deviations, and correlation matrix based on daily asset returns.

```
ExpReturn      = [0.0246  0.0189  0.0273  0.0141  0.0311]/100;
Sigmas         = [0.9509  1.4259  1.5227  1.1062  1.0877]/100;
Correlations   = [1.0000  0.4403  0.4735  0.4334  0.6855
                  0.4403  1.0000  0.7597  0.7809  0.4343
                  0.4735  0.7597  1.0000  0.6978  0.4926
                  0.4334  0.7809  0.6978  1.0000  0.4289
                  0.6855  0.4343  0.4926  0.4289  1.0000];
```

Convert the correlations and standard deviations to a covariance matrix of daily returns.

```
ExpCovariance = corr2cov(Sigmas, Correlations);
```

Assume 252 trading days per calendar year, and simulate a single sample path of daily returns over a four-year period. Since the `ExpReturn` and `ExpCovariance` inputs are expressed daily, set `RetIntervals = 1`.

```
StartPrice    = 100;
NumObs        = 1008;    % four calendar years of daily returns
RetIntervals  = 1;      % one trading day
NumAssets     = length(ExpReturn);
randn('state',0);
RetSeries1 = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, 1, 'Expected');
```

Now annualize the daily data, thereby changing the periodicity of the data, by multiplying `ExpReturn` and `ExpCovariance` by 252 and dividing `RetIntervals` by 252 (`RetIntervals = 1/252` of a year). Resetting the random number generator to its initial state, you can reproduce the results.

```
rng('default');
RetSeries2 = portsim(ExpReturn*252, ExpCovariance*252, ...
NumObs, RetIntervals/252, 1, 'Expected');
```

Assume an equally weighted portfolio and compute portfolio returns associated with each simulated return series.

```
Weights = ones(NumAssets, 1)/NumAssets;
```

```
PortRet1 = RetSeries2 * Weights;
```

```
PortRet2 = RetSeries2 * Weights;
```

Comparison of the data reveals that `PortRet1` and `PortRet2` are identical.

Univariate Geometric Brownian Motion

This example shows how to simulate a univariate geometric Brownian motion process. It is based on an example found in Hull, *Options, Futures, and Other Derivatives*, 5th Edition (see example 12.2 on page 236). In addition to verifying Hull's example, it also graphically illustrates the lognormal property of terminal stock prices by a rather large Monte Carlo simulation.

Assume you own a stock with an initial price of \$20, an annualized expected return of 20% and volatility of 40%. Simulate the daily price process for this stock over the course of one full calendar year (252 trading days).

```
StartPrice = 20;
ExpReturn = 0.2;
ExpCovariance = 0.4^2;
NumObs = 252;
NumSim = 10000;
RetIntervals = 1/252;
```

Note that `RetIntervals` is expressed in years, consistent with the fact that `ExpReturn` and `ExpCovariance` are annualized. Also, note that `ExpCovariance` is entered as a variance rather than the more familiar standard deviation (volatility).

Set the random number generator state, and simulate 10,000 trials (realizations) of stock returns over a full calendar year of 252 trading days.

```
rng('default');
RetSeries = squeeze(portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected'));
```

The `squeeze` function reformats the output array of simulated returns from a 252-by-1-by-10000 array to more convenient 252-by-10000 array. (Recall that `portsim` is fundamentally a multivariate simulation engine).

In accordance with Hull's equations 12.4 and 12.5 on page 236

$$E(S_T) = S_0 e^{\mu T}$$

$$var(S_T) = S_0^2 e^{2\mu T} (e^{\sigma^2 T} - 1)$$

convert the simulated return series to a price series and compute the sample mean and the variance of the terminal stock prices.

```
StockPrices = ret2tick(RetSeries, repmat(StartPrice, 1, NumSim));
```

```
SampMean = mean(StockPrices(end,:))
```

```
SampVar = var(StockPrices(end,:))
```

```
SampMean =
```

```
24.4489
```

```
SampVar =
```

```
101.4243
```

Compare these values with the values you obtain by using Hull's equations.

```
ExpValue = StartPrice*exp(ExpReturn)
```

```
ExpVar = ...
```

```
StartPrice*StartPrice*exp(2*ExpReturn)*(exp((ExpCovariance)) - 1)
```

```
ExpValue =
```

```
24.4281
```

```
ExpVar =
```

```
103.5391
```

These results are very close to the results shown in Hull's example 12.2.

Display the sample density function of the terminal stock price after one calendar year. From the sample density function, the lognormal distribution of terminal stock prices is apparent.

```
[count, BinCenter] = hist(StockPrices(end,:), 30);
```

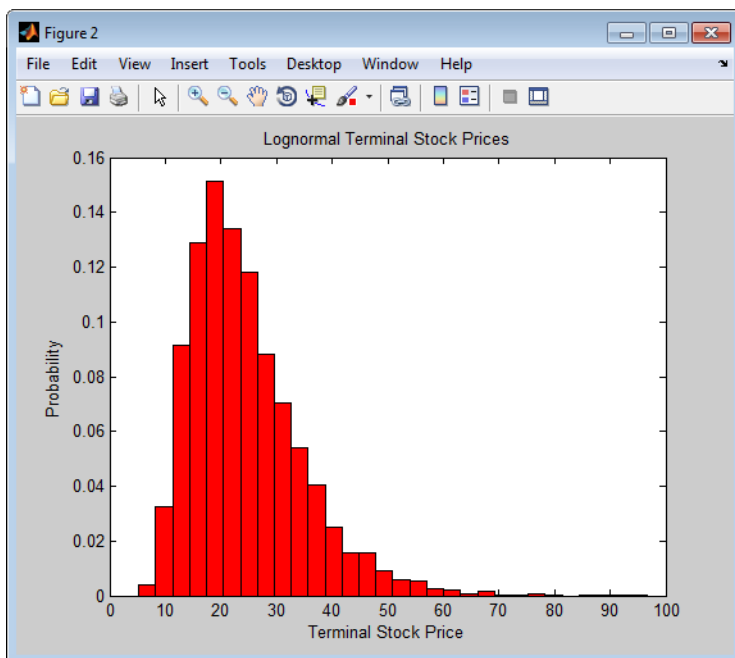
```
figure
```

```
bar(BinCenter, count/sum(count), 1, 'r')
```

```
xlabel('Terminal Stock Price')
```

```
ylabel('Probability')
```

```
title('Lognormal Terminal Stock Prices')
```



References

Hull, John, C., *Options, Futures, and Other Derivatives*, Upper Saddle River, New Jersey: Prentice-Hall. 5th ed., 2003, ISBN 0-13-009056-5.

See Also

ewstats | portopt | portstats | randn | ret2tick | squeeze

portstats

Portfolio expected return and risk

Syntax

```
[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance,  
PortWts)
```

Arguments

ExpReturn	1-by-number of assets (NASSETS) vector specifying the expected (mean) return of each asset.
ExpCovariance	NASSETS-by-NASSETS matrix specifying the covariance of the asset returns.
PortWts	(Optional) Number of portfolios (NPORTS) by NASSETS matrix of weights allocated to each asset. Each row represents a different weighting combination. Default = 1 / NASSETS (equally weighted).

Description

[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance, PortWts) computes the expected rate of return and risk for a portfolio of assets.

PortRisk is an NPORTS-by-1 vector of the standard deviation of each portfolio.

PortReturn is an NPORTS-by-1 vector of the expected return of each portfolio.

Examples

Computes the Expected Rate of Return and Risk for a Portfolio of Assets

This example shows how to calculate the expected rate of return and risk for a portfolio of assets.


```
ExpReturn = [0.1 0.2 0.15];

ExpCovariance = [0.0100    -0.0061    0.0042
                 -0.0061    0.0400   -0.0252
                 0.0042   -0.0252    0.0225 ];

PortWts=[0.4 0.2 0.4; 0.2 0.4 0.2];

[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance,...
PortWts)

PortRisk =

    0.0560
    0.0550

PortReturn =

    0.1400
    0.1300
```

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion”
- “Constraint Specification”
- “Active Returns and Tracking Error Efficient Frontier”

More About

- “Analyzing Portfolios”
- “Portfolio Optimization Functions”

See Also

frontcon

portvar

Variance for portfolio of assets

Syntax

```
V = portvar(Asset, Weight)
```

Arguments

Asset	M-by-N matrix of M asset returns for N securities.
Weight	R-by-N matrix of R portfolio weights for N securities. Each row of Weight constitutes a portfolio of securities in Asset.

Description

`V = portvar(Asset, Weight)` returns the portfolio variance as an R-by-1 vector (assuming `Weight` is a matrix of size R-by-N) with each row representing a variance calculation for each row of `Weight`.

`V = portvar(Asset)` assigns each security an equal weight when calculating the portfolio variance.

References

Bodie, Kane, and Marcus, *Investments*, Chapter 7.

See Also

frontcon | portrand | portror

portvrisk

Portfolio value at risk (VaR)

Syntax

```
ValueAtRisk = portvrisk(PortReturn, PortRisk, RiskThreshold,
PortValue)
```

Arguments

PortReturn	Number of portfolios (NPORTS)-by-1 vector or scalar of the expected return of each portfolio over the period.
PortRisk	NPORTS-by-1 vector or scalar of the standard deviation of each portfolio over the period.
RiskThreshold	(Optional) NPORTS-by-1 vector or scalar specifying the loss probability. Default = 0.05 (5%).
PortValue	(Optional) NPORTS-by-1 vector or scalar specifying the total value of asset portfolio. Default = 1.

Description

`ValueAtRisk = portvrisk(PortReturn, PortRisk, RiskThreshold, PortValue)` returns the maximum potential loss in the value of a portfolio over one period of time, given the loss probability level `RiskThreshold`.

`ValueAtRisk` is an NPORTS-by-1 vector of the estimated maximum loss in the portfolio, predicted with a confidence probability of $1 - \text{RiskThreshold}$. `portvrisk` calculates `ValueAtRisk` using a normal distribution.

If `PortValue` is not given, `ValueAtRisk` is presented on a per-unit basis. A value of 0 indicates no losses.

Examples

Compute the Maximum Potential Loss in the Value of a Portfolio Over One Period of Time

This example shows how to return the maximum potential loss in the value of a portfolio over one period of time, where `ValueAtRisk` is computed on a per-unit basis.

```
PortReturn = 0.29/100;  
PortRisk = 3.08/100;  
RiskThreshold = [0.01;0.05;0.10];  
PortValue = 1;  
ValueAtRisk = portvrisk(PortReturn,PortRisk,...  
RiskThreshold,PortValue)
```

```
ValueAtRisk =  
  
    0.0688  
    0.0478  
    0.0366
```

Compute the Maximum Potential Loss in the Value of a Portfolio Over One Period of Time Using Actual Values

This example shows how to return the maximum potential loss in the value of a portfolio over one period of time, where `ValueAtRisk` is computed with actual values.

```
PortReturn = [0.29/100;0.30/100];  
PortRisk = [3.08/100;3.15/100];  
RiskThreshold = 0.10;  
PortValue = [1000000000;500000000];  
ValueAtRisk = portvrisk(PortReturn,PortRisk,...  
RiskThreshold,PortValue)
```

```
ValueAtRisk =  
  
    1.0e+07 *  
  
    3.6572  
    1.8684
```

See Also

frontcon | portopt

posvalidx

Positive volume index

Syntax

```
pvi = posvalidx(closep, tvolume, initpvi)
```

```
pvi = posvalidx([closep tvolume], initpvi)
```

```
pvits = posvalidx(tsobj)
```

```
pvits = posvalidx(tsobj, initpvi, ParameterName, ParameterValue, ...)
```

Arguments

<code>closep</code>	Closing price (vector).
<code>tvolume</code>	Volume traded (vector).
<code>initpvi</code>	(Optional) Initial value for positive volume index. Default = 100.
<code>tsobj</code>	Financial time series object.

Description

`pvi = posvalidx(closep, tvolume, initpvi)` calculates the positive volume index from a set of stock closing prices (`closep`) and volume traded (`tvolume`) data. `pvi` is a vector representing the positive volume index. If `initpvi` is specified, `posvalidx` uses that value instead of the default (100).

`pvi = posvalidx([closep tvolume], initpvi)` accepts a two-column matrix, the first column representing the closing prices (`closep`) and the second representing the volume traded (`tvolume`). If `initpvi` is specified, `posvalidx` uses that value instead of the default (100).

`pvits = posvalidx(tsobj)` calculates the positive volume index from the financial time series object `tsobj`. The object must contain, at least, the series `Close` and `Volume`.

The `pvits` output is a financial time series object with dates similar to `tsobj` and a data series named PVI. The initial value for the positive volume index is arbitrarily set to 100.

`pvits = posvalidx(tsobj, initpvi, ParameterName, ParameterValue,...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

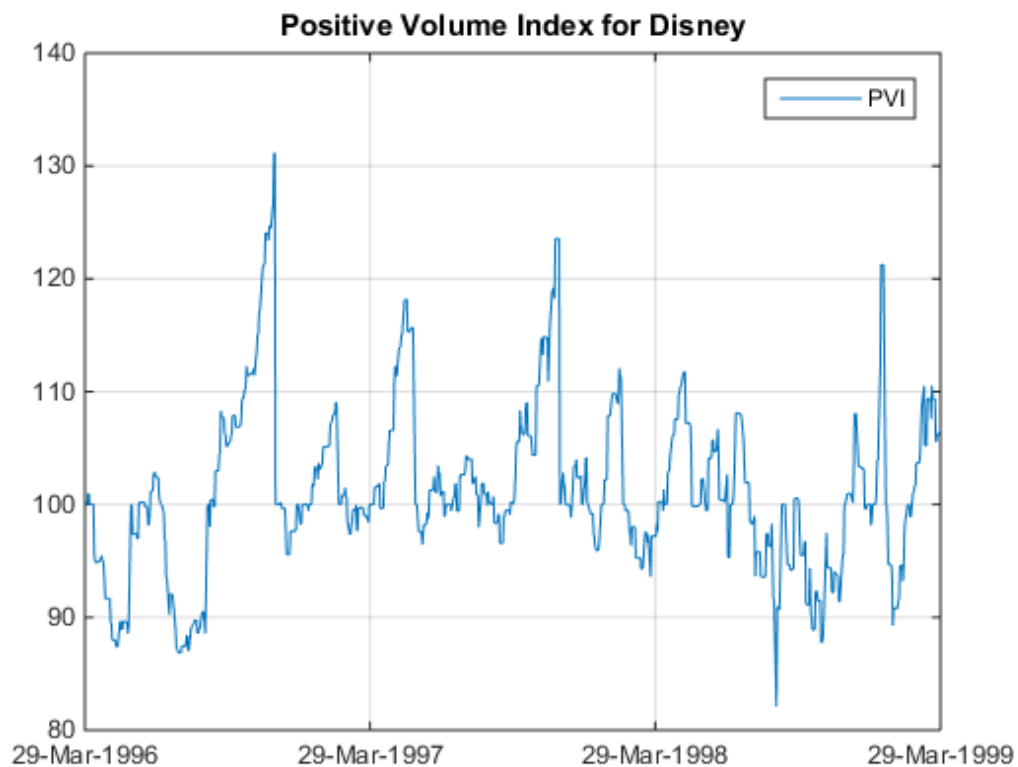
Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Positive Volume Index

This example shows how to compute the positive volume index for Disney stock and plot the results.

```
load disney.mat
dis_PosVol = posvalidx(dis);
plot(dis_PosVol)
title('Positive Volume Index for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 236 - 238.

See Also

onbalvol | negvolidx

power

Financial time series power

Syntax

```
newfts = tsobj .^ array
```

```
newfts = array .^tsobj
```

```
newfts = tsobj_1 .^ tsobj_2
```

Arguments

<code>tsobj</code>	Financial time series object.
<code>array</code>	Scalar value or array with the number of rows equal to the number of dates in <code>tsobj</code> and the number of columns equal to the number of data series in <code>tsobj</code> .
<code>tsobj_1, tsobj_2</code>	Pair of financial time series objects.

Description

`newfts = tsobj .^ array` raises all values in the data series of the financial time series object `tsobj` element by element to the power indicated by the array value. The results are stored in another financial time series object `newfts`. The `newfts` object contains the same data series names as `tsobj`.

`newfts = array .^ tsobj` raises the array values element by element to the values contained in the data series of the financial time series object `tsobj`. The results are stored in another financial time series object `newfts`. The `newfts` object contains the same data series names as `tsobj`.

`newfts = tsobj_1 .^ tsobj_2` raises the values in the object `tsobj_1` element by element to the values in the object `tsobj_2`. The data series names, the dates, and the number of data points in both series must be identical. `newfts` contains the same data series names as the original time series objects.

See Also

minus | plus | rdivide | times

prbyzero

Price bonds in portfolio by set of zero curves

Syntax

`BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates, Compounding)`

Arguments

Bonds	Coupon bond information used to compute prices. A number of bonds (NUMBONDS)-by-6 matrix where each row describes a bond. The first two columns are required; the rest are optional but must be added in order. All rows in Bonds must have the same number of columns. Columns are [Maturity CouponRate Face Period Basis EndMonthRule] where:	
	Maturity	Maturity date as a serial date number or date string.
	CouponRate	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
	Face	(Optional) Face or par value of the bond. Default = 100.
	Period	(Optional) Coupons per year of the bond. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
	Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365

		<ul style="list-style-type: none"> • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
	EndMonthRule	(Optional) End-of-month rule. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
Settle	Serial date number of the settlement date.	
ZeroRates	NUMDATES-by-NUMCURVES matrix of observed zero rates, as decimal fractions. Each column represents a rate curve. Each row represents an observation date.	
ZeroDates	NUMDATES-by-1 column of dates for observed zeros	
Compounding	Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors. Compounding values are: 1, 2, 3, 4, 6, 12.	

Description

`BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates, Compounding)` computes the bond prices in a portfolio using a set of zero curves.

`BondPrices` is a NUMBONDS-by-NUMCURVES matrix of clean bond prices. Each column is derived from the corresponding zero curve in `ZeroRates`.

In addition, you can use the Financial Instruments Toolbox™ method `getZeroRates` for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `prbyzero`. For more information, see “Converting an `IRDataCurve` or `IRFunctionCurve` Object”.

Examples

Compute the Bond Prices in a Portfolio Using a Set of Zero Curves

This example uses the function `zbtprice` to compute a zero curve given a portfolio of coupon bonds and their prices. It then reverses the process, using the zero curve as input to the function `prbyzero` to compute the prices.

```
Bonds = [datenum('6/1/1998') 0.0475 100 2 0 0;
          datenum('7/1/2000') 0.06 100 2 0 0;
          datenum('7/1/2000') 0.09375 100 6 1 0;
          datenum('6/30/2001') 0.05125 100 1 3 1;
          datenum('4/15/2002') 0.07125 100 4 1 0;
          datenum('1/15/2000') 0.065 100 2 0 0;
          datenum('9/1/1999') 0.08 100 3 3 0;
          datenum('4/30/2001') 0.05875 100 2 0 0;
          datenum('11/15/1999') 0.07125 100 2 0 0;
          datenum('6/30/2000') 0.07 100 2 3 1;
          datenum('7/1/2001') 0.0525 100 2 3 0;
          datenum('4/30/2002') 0.07 100 2 0 0];
```

```
Prices = [ 99.375;
           99.875;
           105.75 ;
           96.875;
           103.625;
           101.125;
           103.125;
```

```
    99.375;  
    101.0  ;  
    101.25 ;  
    96.375;  
    102.75 ];
```

```
Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve, on an actual/365 basis.

```
OutputCompounding = 2;
```

Execute the function `zbtprice` which returns the zero curve at the maturity dates.

```
[ZeroRates, ZeroDates] = zbtprice(Bonds, Prices, Settle,...  
OutputCompounding)
```

```
ZeroRates =
```

```
    0.0616  
    0.0609  
    0.0658  
    0.0590  
    0.0647  
    0.0655  
    0.0606  
    0.0601  
    0.0642  
    0.0621  
    0.0627
```

```
ZeroDates =
```

```
    729907  
    730364  
    730439  
    730500  
    730667  
    730668  
    730971  
    731032  
    731033  
    731321
```

731336

Execute the function `prbyzero`.

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

```
BondPrices =
```

```
99.3750  
98.7980  
106.8270  
96.8750  
103.6249  
101.1250  
103.1250  
99.3637  
101.0000  
101.2500  
96.3750  
102.7384
```

In this example `zbtprice` and `prbyzero` do not exactly reverse each other. Many of the bonds have the end-of-month rule off (`EndMonthRule = 0`). The rule subtly affects the time factor computation. If you set the rule on (`EndMonthRule = 1`) everywhere in the `Bonds` matrix, then `prbyzero` returns the original prices, except when the two incompatible prices fall on the same maturity date.

See Also

`tr2bonds` | `zbtprice`

prcroc

Price rate of change

Syntax

```
proc = prcroc(closep, nTimes)
```

```
procts = prcroc(tsobj, nTimes)
```

```
procts = prcroc(tsobj, nTimes, ParameterName, ParameterValue)
```

Arguments

closep	Closing price
nTimes	(Optional) Time difference. Default = 12.
tsobj	Financial time series object

Description

`proc = prcroc(closep, nTimes)` calculates the price rate of change `proc` from the closing price `closep`. If `nTimes` time is specified, the price rate of change is calculated between the current closing price and the closing price `nTimes` ago.

`procts = prcroc(tsobj, nTimes)` calculates the price rate of change `procts` from the financial time series object `tsobj`. `tsobj` must contain a data series named `Close`. The output `procts` is a financial time series object with similar dates as `tsobj` and a data series named `PriceROC`. If `nTimes` is specified, the price rate of change is calculated between the current closing price and the closing price `nTimes` ago.

`procts = prcroc(tsobj, nTimes, ParameterName, ParameterValue)` specifies the name for the required data series when it is different from the default name. The valid parameter name is

- `CloseName`: closing price series name

The parameter value is a string that represents the valid parameter name.

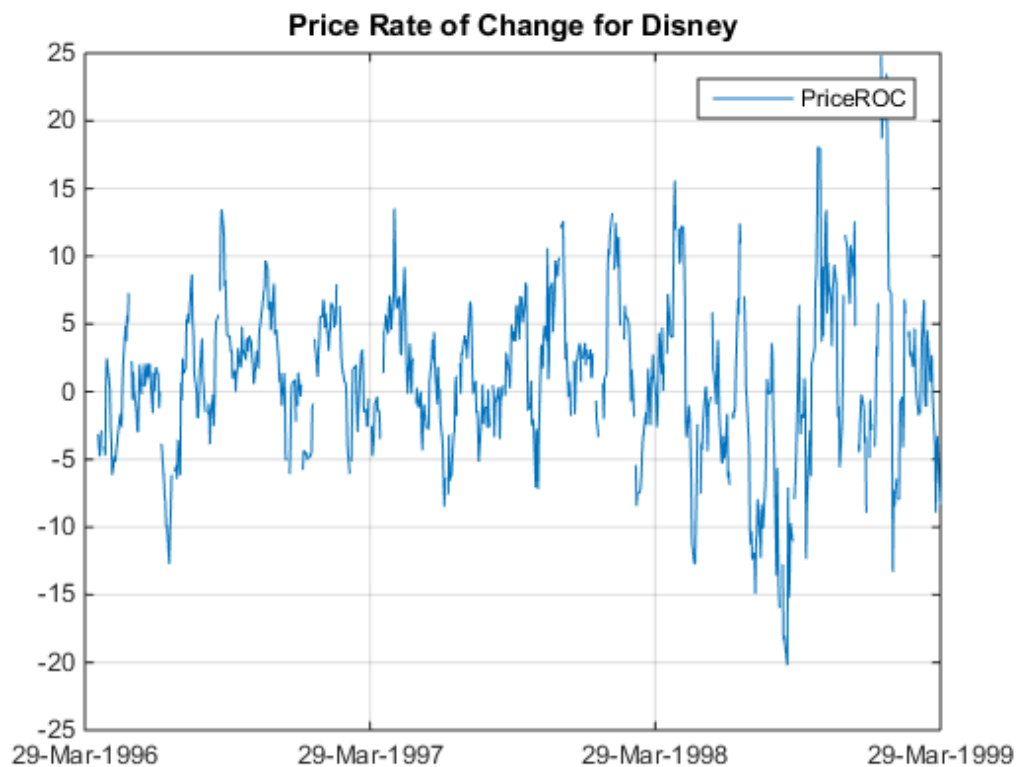
Note, to compute a quantity over n periods, you must specify $n+1$ for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Compute the Price Rate of Change

This example shows how to compute the price rate of change for Disney stock and plot the results.

```
load disney.mat
dis_PriceRoc = prcroc(dis);
plot(dis_PriceRoc)
title('Price Rate of Change for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 243 - 245.

See Also

volroc

prdisc

Price of discounted security

Syntax

Price = prdisc(Settle, Maturity, Face, Discount, Basis)

Arguments

Settle	Enter as serial date number or date string. Settle must be earlier than Maturity.
Maturity	Enter as serial date number or date string.
Face	Redemption (par, face) value.
Discount	Bank discount rate of the security. Enter as decimal fraction.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252

For more information, see basis .
--

Description

Price = `prdisc(Settle, Maturity, Face, Discount, Basis)` returns the price of a security whose yield is quoted as a bank discount rate (for example, U. S. Treasury bills).

Examples

Calculate the Price of a Security Whose Yield is Quoted as a Bank Discount Rate

This example shows how to return the price of a security whose yield is quoted as a bank discount rate (for example, U. S. Treasury bills).

```
Settle = '10/14/2000';  
Maturity = '03/17/2001';  
Face = 100;  
Discount = 0.087;  
Basis = 2;
```

```
Price = prdisc(Settle, Maturity, Face, Discount, Basis)
```

```
Price =  
  
    96.2783
```

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 2.

See Also

`acrudisc` | `bndprice` | `discrate` | `prmat` | `ylddisc`

priceandvol

Price and volume chart

Syntax

priceandvol(X)

Arguments

X	M-by-6 matrix where the columns are date, open, high, low, close, and volume.
---	---

Description

priceandvol(X) plots the asset data displaying the open, high, low, and closing prices on one axis and the volume on a second axis.

Examples

Create a Price Volume Chart

This example shows how to create a price volume chart, given asset X as an M-by-6 matrix for date, open, high, low, close, and volume.

X = [...

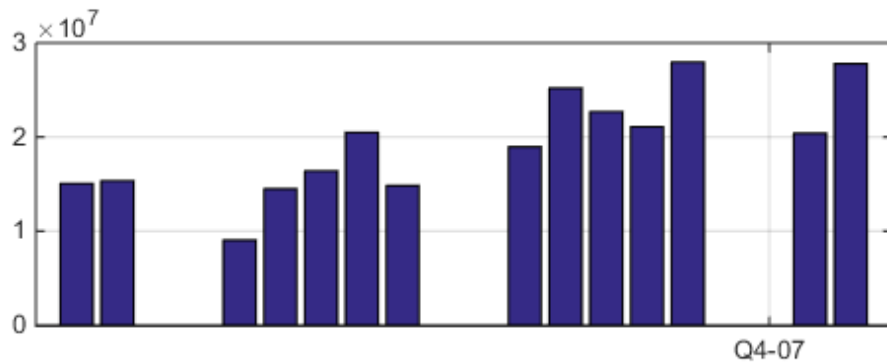
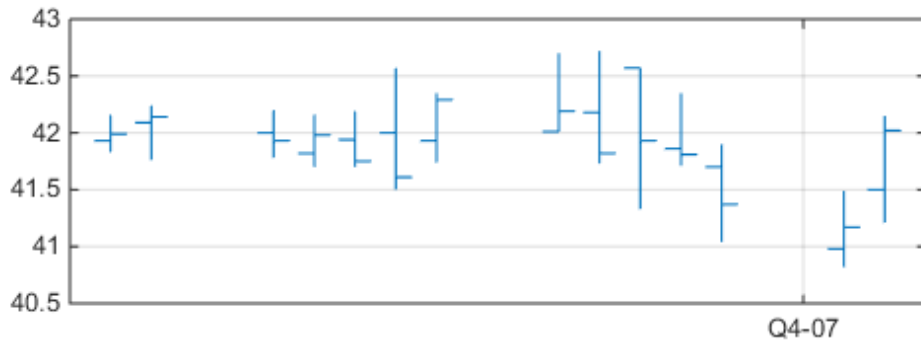
```

733299.00      41.93      42.15      41.83      41.99      15045445.00;...
733300.00      42.09      42.24      41.76      42.14      15346658.00;...
733303.00      42.00      42.20      41.78      41.93      9034397.00;...
733304.00      41.82      42.16      41.70      41.98      14486275.00;...
733305.00      41.94      42.19      41.70      41.75      16389872.00;...
733306.00      42.00      42.57      41.50      41.61      20475208.00;...
733307.00      41.93      42.35      41.74      42.29      14833200.00;...
733310.00      42.01      42.70      42.01      42.19      18945176.00;...
733311.00      42.18      42.72      41.73      41.82      25188101.00;...

```

733312.00	42.57	42.57	41.33	41.93	22689878.00;...
733313.00	41.86	42.35	41.71	41.81	21084723.00;...
733314.00	41.70	41.90	41.04	41.37	27963619.00;...
733317.00	40.98	41.49	40.82	41.17	20385033.00;...
733318.00	41.50	42.15	41.21	42.02	27783775.00];

priceandvol(X)



See Also

[bolling](#) | [kagi](#) | [candle](#) | [highlow](#) | [linebreak](#) | [movavg](#) | [pointfig](#) | [renko](#) | [volarea](#)

prmat

Price with interest at maturity

Syntax

[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face, CouponRate, Yield, Basis)

Arguments

Settle	Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Enter as serial date number or date string.
Issue	Enter as serial date number or date string.
Face	Redemption (par, face) value.
CouponRate	Enter as decimal fraction.
Yield	Annual yield. Enter as decimal fraction.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Description

`[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face, CouponRate, Yield, Basis)` returns the price and accrued interest of a security that pays interest at maturity. This function also applies to zero-coupon bonds or pure discount securities by setting `CouponRate = 0`.

Examples

Compute the Price and Accrued Interest of a Security That Pays Interest at Maturity

This example shows how to compute the price and accrued interest of a security that pays interest at maturity.

```
Settle = '02/07/2002';  
Maturity = '04/13/2002';  
Issue = '10/11/2001';  
Face = 100;  
CouponRate = 0.0608;  
Yield = 0.0608;  
Basis = 1;
```

```
[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face, ...  
CouponRate, Yield, Basis)
```

```
Price =
```

```
99.9784
```

```
AccruInterest =
```


1.9591

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 4.

See Also

acrubond | acrudisc | bndprice | prdisc | yldmat

prtbill

Price of Treasury bill

Syntax

Price = prtbill(Settle, Maturity, Face, Discount)

Arguments

Settle	Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Enter as serial date number or date string.
Face	Redemption (par, face) value.
Discount	Discount rate of the Treasury bill. Enter as decimal fraction.

Description

Price = prtbill(Settle, Maturity, Face, Discount) returns the price for a Treasury bill.

Examples

Calculate the Price for a Treasury Bill

This example shows how to return the price for a Treasury bill, where the settlement date of a Treasury bill is February 10, 2002, the maturity date is August 6, 2002, the discount rate is 3.77%, and the par value is \$1000.

```
Price = prtbill('2/10/2002', '8/6/2002', 1000, 0.0377)
```

Price =

981.4642

References

Bodie, Kane, and Marcus, *Investments*, pages 41-43.

See Also

beytbill | yldtbill

pvfix

Present value with fixed periodic payments

Syntax

```
PresentVal = pvfix(Rate, NumPeriods, Payment, ExtraPayment, Due)
```

Arguments

rate	Periodic interest rate, as a decimal fraction.
NumPeriods	Number of periods.
Payment	Periodic payment.
ExtraPayment	(Optional) Payment received other than Payment in the last period. Default = 0.
Due	(Optional) When payments are due or made: 0 = end of period (default), or 1 = beginning of period.

Description

```
PresentVal = pvfix(Rate, NumPeriods, Payment, ExtraPayment, Due)
```

returns the present value of a series of equal payments.

Examples

Calculate the Present Value of a Series of Equal Payments

This example shows how to return the present value of a series of equal payments, where \$200 is paid monthly into a savings account earning 6%. The payments are made at the end of the month for five years.

```
PresentVal = pvfix(0.06/12, 5*12, 200, 0, 0)
```

PresentVal =

1.0345e+04

See Also

fvfix | fvvar | payper | pvvar

pvtrend

Price and Volume Trend (PVT)

Syntax

```
pvt = pvtrend(closep, tvolume)
```

```
pvt = pvtrend([closep tvolume])
```

```
pvtts = pvtrend(tsobj)
```

```
pvtts = pvtrend(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

closep	Closing price.
tvolume	Volume traded.
tsobj	Financial time series object.
ParameterName	Valid parameter names are: <ul style="list-style-type: none">• <code>CloseName</code>: closing prices series name• <code>VolumeName</code>: volume traded series name
ParameterValue	Parameter values are the strings that represent the valid parameter names.

Description

`pvt = pvtrend(closep, tvolume)` calculates the Price and Volume Trend (PVT) from the stock closing price (`closep`) data and the volume traded (`tvolume`) data.

`pvt = pvtrend([closep tvolume])` accepts a two-column matrix in which the first column contains the closing prices (`closep`) and the second contains the volume traded (`tvolume`).

`pvtts = pvtrend(tsobj)` calculates the PVT from the stock data contained in the financial time series object `tsobj`. The object `tsobj` must contain the closing price series `Close` and the volume traded series `Volume`. The output `pvtts` is a financial time series object with dates similar to `tsobj` and a data series named `PVT`.

`pvtts = pvtrend(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name/ parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the strings that represent the valid parameter names.

Examples

Calculate the Price and Volume Trend (PVT)

This example shows how to calculate the PVT for Disney stock and plot the results.

```
load disney.mat
dis_PVTrend = pvtrend(dis);
plot(dis_PVTrend)
title('Price and Volume Trend for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 239 - 240.

pvvar

Present value of varying cash flow

Syntax

PresentVal = pvvar(CashFlow, Rate, CFDates)

Arguments

CashFlow	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number). If CashFlow is a matrix, each column is treated as a separate cash-flow stream.
Rate	Periodic interest rate. Enter as a decimal fraction. If CashFlow is a matrix, a scalar Rate is allowed when the same rate applies to all cash-flow streams in CashFlow . When multiple cash-flow streams require different discount rates, Rate must be a vector whose length equals the number of columns in CashFlow .
CFDates	(Optional) A vector of serial date numbers or date strings on which the cash flows occur. Specify CFDates when there are irregular (nonperiodic) cash flows. The default assumes that CashFlow contains regular (periodic) cash flows. If CashFlow is a matrix, and all cash-flow streams share the same dates, CFDates can be a vector whose length matches the number of rows in CashFlow . When different cash-flow streams have different payment dates, specify CFDates as a matrix the same size as CashFlow .

Description

PresentVal = pvvar(CashFlow, Rate, CFDates) returns the net present value of a varying cash flow. Present value is calculated at the time the first cash flow occurs.

Examples

This cash flow represents the yearly income from an initial investment of \$10,000. The annual interest rate is 8%.

Year 1	\$2000
Year 2	\$1500
Year 3	\$3000
Year 4	\$3800
Year 5	\$5000

To calculate the net present value of this regular cash flow

```
PresentVal = pvvar([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
PresentVal =
```

```
1715.39
```

An investment of \$10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

Cash Flow	Dates
(\$10000)	January 12, 1987
\$2500	February 14, 1988
\$2000	March 3, 1988
\$3000	June 14, 1988
\$4000	December 1, 1988

To calculate the net present value of this irregular cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
```

```
CFDates = [ '01/12/1987'  
            '02/14/1988'  
            '03/03/1988'  
            '06/14/1988'
```

```
        '12/01/1988'];  
PresentVal = pvvar(CashFlow, 0.09, CFDates)  
  
returns  
PresentVal =  
  
        142.16
```

The net present value of the same investment under different discount rates of 7%, 9%, and 11% is obtained in a single call:

```
PresentVal = pvvar(repmat(CashFlow,3,1)', [.07 .09 .11], CFDates)  
  
pv =  
  
    419.0136   142.1648  -122.1275
```

See Also

fvfix | fvvar | irr | payuni | pvfix

pyld2zero

Zero curve given par yield curve

Syntax

```
[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates, Settle,
Compounding, Basis, InputCompounding)
```

Arguments

ParRates	Column vector of annualized implied par yield rates, as decimal fractions. (Par yields = coupon rates.) In aggregate, the yield rates in ParRates constitute an implied par yield curve for the investment horizon represented by CurveDates .
CurveDates	Column vector of maturity dates (as serial date numbers) that correspond to the par rates.
Settle	Serial date number that is the common settlement date for the par rates.
Compounding	(Optional) Scalar value representing the periodicity in which the output zero rates are compounded when annualized. Allowed values are:
	1 Annual compounding
	2 Semiannual compounding (default)
	3 Compounding three times per year
	4 Quarterly compounding
	6 Bimonthly compounding
	12 Monthly compounding
	365 Daily compounding
Basis	(Optional) Day-count basis used to annualize the zero rates. <ul style="list-style-type: none"> • 0 = actual/actual (default)

	<ul style="list-style-type: none"> • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
InputCompounding	(Optional) Scalar value representing the periodicity in which the input par rates were compounded when annualized. The default is the value for Compounding.

Description

[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates, Settle, Compounding, Basis, InputCompounding) returns a zero curve given a par yield curve and its maturity dates.

Note: pyld2zero uses zbtprice for computations.

ZeroRates	Column vector of decimal fractions. In aggregate, the rates in ZeroRates constitute a zero curve for the investment horizon represented by CurveDates.
-----------	--

CurveDates	Column vector of maturity dates (as serial date numbers) corresponding to the zero rates. This vector is the same as the input vector CurveDates.
------------	---

Examples

Compute Zero Curve Given Par Yield Curve

Define the settlement date, maturity, and zero rates.

```
Settle = datenum('01-Feb-2013');  
CurveDates = datemnth(Settle,12*[1 2 3 5 7 10 20 30]');  
ZeroRates = [.11 0.30 0.64 1.44 2.07 2.61 3.29 3.55]'/100;  
Compounding = 2;  
Basis = 1;
```

Compute par yield curve from zero rates.

```
ParRates = zero2pyld(ZeroRates, CurveDates, Settle, Compounding, Basis)
```

```
ParRates =
```

```
    0.0011  
    0.0030  
    0.0064  
    0.0142  
    0.0201  
    0.0251  
    0.0309  
    0.0330
```

Compute zero curve from the par yield curve.

```
ZeroRates = pyld2zero(ParRates, CurveDates, Settle, Compounding, Basis)
```

```
ZeroRates =
```

```
    0.0011  
    0.0030  
    0.0064
```

0.0144
0.0207
0.0261
0.0329
0.0355

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

zero2pyld

rdivide

Financial time series division

Syntax

```
newfts = tsobj_1 ./ tsobj_2
```

```
newfts = tsobj ./ array
```

```
newfts = array ./ tsobj
```

Arguments

<code>tsobj_1, tsobj_2</code>	Pair of financial time series objects.
<code>array</code>	Scalar value or array with the number of rows equal to the number of dates in <code>tsobj</code> and the number of columns equal to the number of data series in <code>tsobj</code> .

Description

The `rdivide` method divides, element by element, the components of one financial time series object by the components of the other. You can also divide the whole object by an array or divide a financial time series object into an array.

If an object is to be divided by another object, both objects must have the same dates and data series names, although the order need not be the same. The order of the data series, when an object is divided by another object, follows the order of the first object.

`newfts = tsobj_1 ./ tsobj_2` divides financial time series objects element by element.

`newfts = tsobj ./ array` divides a financial time series object element by element by an array.

`newfts = array ./ tsobj` divides an array element by element by a financial time series object.

For financial time series objects, the `rdivide` operation is identical to the `mrdivide` operation.

See Also

`minus` | `mrdivide` | `plus` | `times`

renko

Renko chart

Syntax

`renko(X)`

`renko(X, threshold)`

Arguments

<code>X</code>	M-by-2 matrix where the first column contains date numbers and the second column is the asset price.
<code>threshold</code>	(Optional) Specifies a threshold value for asset price. By default, <code>threshold</code> is set to 1.

Description

`renko(X)` plots asset price with respect to dates.

`renko(X, threshold)` plots the asset data, `X`, adding a new box only when the price has changed but at least the value specified by `threshold`.

Examples

Plot Asset Price With Respect to Dates

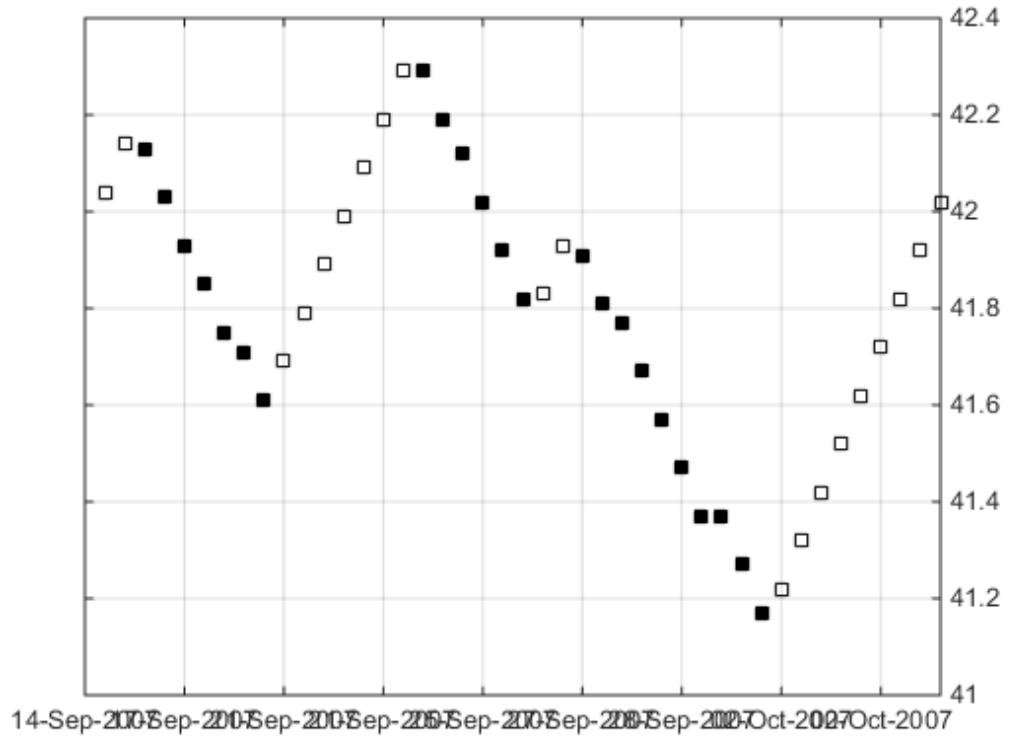
This example shows how to plot asset price with respect to dates, given asset `X` as an M-by-2 matrix of date numbers and asset prices, generate a Renko chart.

```
X = [...
```

```
733299.00      41.99; ...
733300.00      42.14; ...
733303.00      41.93; ...
733304.00      41.98; ...
```

733305.00	41.75;...
733306.00	41.61;...
733307.00	42.29;...
733310.00	42.19;...
733311.00	41.82;...
733312.00	41.93;...
733313.00	41.81;...
733314.00	41.37;...
733317.00	41.17;...
733318.00	42.02];

renko(X, .1)



See Also

`bolling` | `kagi` | `candle` | `highlow` | `linebreak` | `movavg` | `pointfig` | `priceandvol` | `volarea`

resamplets

Downsample data

Syntax

```
newfts = resamplets(oldfts, samplestep)
```

Description

`newfts = resamplets(oldfts, samplestep)` downsamples the data contained in the financial time series object `oldfts` every `samplestep` periods. For example, to have the new financial time series object contain every other data element from `oldfts`, set `samplestep` to 2.

`newfts` is a financial time series object containing the same data series (names) as the input `oldfts`.

See Also

`filter`

ret2tick

Convert return series to price series

Syntax

```
[TickSeries, TickTimes] = ret2tick(RetSeries, StartPrice,
RetIntervals, StartTime, Method)
```

Arguments

RetSeries	Number of observations (NUMOBS) by number of assets (NASSETS) time series array of asset returns associated with the prices in TickSeries. The <i>i</i> th return is quoted for the period TickTimes(<i>i</i>) to TickTimes(<i>i</i> +1) and is not normalized by the time increment between successive price observations.
StartPrice	(Optional) 1-by-NASSETS vector of initial asset prices or a single scalar initial price applied to all assets. Prices start at 1 if StartPrice is not specified.
RetIntervals	(Optional) Scalar or NUMOBS-by-1 vector of interval times between observations. If this argument is not specified, all intervals are assumed to have length 1.
StartTime	(Optional) Starting time for first observation, applied to the price series of all assets. The default is zero.
Method	(Optional) Character string indicating the method to convert asset returns to prices. Must be 'Simple' (default) or 'Continuous'. If Method is 'Simple', ret2tick uses simple periodic returns. If Method is 'Continuous', the function uses continuously compounded returns. Case is ignored for Method.

Description

`[TickSeries, TickTimes] = ret2tick(RetSeries, StartPrice, RetIntervals, StartTime, Method)` generates price values from the starting prices of NASSETS investments and NUMOBS incremental return observations.

`TickSeries` is a `NUMOBS+1`-by-`NASSETS` times series array of equity prices. The first row contains the oldest observations and the last row the most recent. Observations across a given row occur at the same time for all columns. Each column is a price series of an individual asset. If `Method` is unspecified or `'Simple'`, the prices are

$$\text{TickSeries}(i+1) = \text{TickSeries}(i) * [1 + \text{RetSeries}(i)]$$

If `Method` is `'Continuous'`, the prices are

$$\text{TickSeries}(i+1) = \text{TickSeries}(i) * \exp[\text{RetSeries}(i)]$$

`TickTimes` is a `NUMOBS+1` column vector of monotonically increasing observation times associated with the prices in `TickSeries`. The initial time is zero unless specified in `StartTime`, and sequential observation times occur at unit increments unless specified in `RetIntervals`.

Examples

Convert a Return Series to a Price Series

This example shows how to compute the price increase of two stocks over a year's time based on three incremental return observations.

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05];
```

```
RetIntervals = [182
                91
                92];
```

```
StartTime = datenum('18-Dec-2000');
```

```
[TickSeries, TickTimes] = ret2tick(RetSeries, [], RetIntervals, ...
    StartTime)
```

```
datestr(TickTimes)
```

```
TickSeries =
```

```
    1.0000    1.0000  
    1.1000    1.1200  
    1.1550    1.1648  
    1.0973    1.2230
```

```
TickTimes =
```

```
    730838  
    731020  
    731111  
    731203
```

```
ans =
```

```
18-Dec-2000  
18-Jun-2001  
17-Sep-2001  
18-Dec-2001
```

See Also

portsim | tick2ret

ret2tick (fts)

Convert return series to price series for time series object

Syntax

```
priceFts = ret2tick(returnFts)
```

```
priceFts = ret2tick(returnFts, 'PARAM1', VALUE1,
'PARAM2', VALUE2', ...)
```

Arguments

returnFts	Financial time series object of returns.
'PARAM1 '	(Optional) StartPrice is a Numeric value and is a scalar or 1-by-N vector of initial prices for each asset. If StartPrice is unspecified or empty, the initial price of all assets is 1.
'PARAM2 '	(Optional) StartTime is Date value for a scalar date number or a single date string specifying the starting time for the first observation. This date is applied to the price series of all assets. Note: The first period price value of the resulting price series will not be reported if StartTime is not specified. The resulting price series will be scaled based on the StartPrice , even if StartTime is not supplied.
'PARAM3 '	(Optional) Method is a character string indicating the method to convert asset returns to prices. The value must be defined as 'Simple' (default) or 'Continuous'. If Method is 'Simple', ret2tick uses simple periodic returns. If Method is 'Continuous', the function uses continuously compounded returns. Case is ignored for Method .

Description

`priceFts = ret2tick(returnFts, 'PARAM1', VALUE1, 'PARAM2', VALUE2', ...)` generates a financial time series object of prices.

If *Method* is unspecified or 'Simple', the prices are

```
PriceSeries(i+1) = PriceSeries(i)*[1 + ReturnSeries(i)]
```

If *Method* is 'Continuous', the prices are

```
PriceSeries(i+1) = PriceSeries(i)*exp[ReturnSeries(i)]
```

Examples

Compute the price series from the following return series:

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05]
```

Use the following dates:

```
Dates = {'18-Jun-2001'; '17-Sep-2001'; '18-Dec-2001'}
```

where

```
ret = fints(Dates, RetSeries)
```

```
ret =
desc: (none)
freq: Unknown (0)
```

```
'dates: (3)'      'series1: (3)'      'series2: (3)'
'18-Jun-2001'    [      0.1000]    [      0.1200]
'17-Sep-2001'    [      0.0500]    [      0.0400]
'18-Dec-2001'    [     -0.0500]    [      0.0500]
```

PriceFts is computed as:

```
PriceFts = ret2tick(ret, 'StartPrice', 100, 'StartTime', '18-Dec-2000')
```

```
PriceFts =
```

```
desc: (none)
freq: Unknown (0)
```

```
'dates: (4)'      'series1: (4)'    'series2: (4)'  
'18-Dec-2000'    [          100]    [          100]  
'18-Jun-2001'    [    110.0000]    [    112.0000]  
'17-Sep-2001'    [    115.5000]    [    116.4800]  
'18-Dec-2001'    [    109.7250]    [    122.3040]
```

See Also

portsim | tick2ret

rmfield

Remove data series

Syntax

```
fts = rmfield(tsoj, fieldname)
```

Arguments

<code>tsoj</code>	Financial time series object.
<code>fieldname</code>	String array containing the data series name to remove a single series from the object. Cell array of data series names to remove multiple data series from the object at the same time.

Description

`fts = rmfield(tsoj, fieldname)` removes the data series `fieldname` and its contents from the financial time series object `tsoj`.

See Also

`chfield` | `extfield` | `fieldnames` | `getfield` | `isfield`

rsindex

Relative Strength Index (RSI)

Syntax

```
rsi = rsindex(closep, nperiods)
```

```
rsits = rsindex(tsobj, nperiods)
```

```
rsits = rsindex(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

<code>closep</code>	Vector of closing prices.
<code>nperiods</code>	(Optional) Number of periods. Default = 14.
<code>tsobj</code>	Financial time series object.

Description

`rsi = rsindex(closep, nperiods)` calculates the Relative Strength Index (RSI) from the closing price vector `closep`.

`rsits = rsindex(tsobj, nperiods)` calculates the RSI from the closing price series in the financial time series object `tsobj`. The object `tsobj` must contain at least the series `Close`, representing the closing prices. The output `rsits` is a financial time series object whose dates are the same as `tsobj` and whose data series name is `RSI`.

`rsits = rsindex(tsobj, nperiods, ParameterName, ParameterValue)` accepts a parameter name/parameter value pair as input. This pair specifies the name for the required data series if it is different from the expected default name. The valid parameter name is

`CloseName`: closing prices series name

The parameter value is the string that represents the valid parameter name.

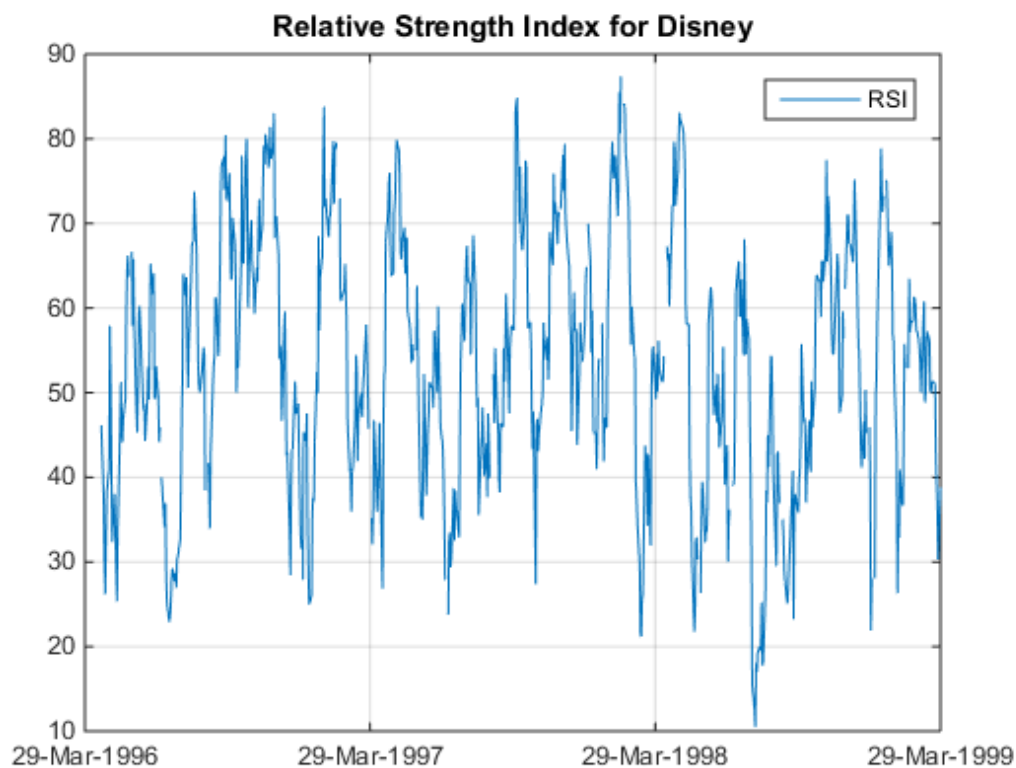
- 1 The relative strength factor is calculated by dividing the average of the gains by the average of the losses within a specified time period:
$$RS = (\text{average gains}) / (\text{average losses}).$$
- 2 The first value of RSI, `RSI(1)`, is set as `NaN` to preserve the dimensions of `CLOSEP`.

Examples

Calculate the Relative Strength Index (RSI)

This example shows how to calculate the RSI for Disney stock and plot the results.

```
load disney.mat
dis_RSI = rsindex(dis);
plot(dis_RSI)
title('Relative Strength Index for Disney')
```



References

Murphy, John J., *Technical Analysis of the Futures Market*, New York Institute of Finance, 1986, pp. 295-302.

See Also

negvalidx | posvalidx

sde

Create SDE model from user-specified functions

Syntax

```
SDE = sde(DriftRate, DiffusionRate)
```

```
SDE = sde(DriftRate, DiffusionRate, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Class

Stochastic Differential Equations

Description

This function creates and displays general stochastic differential equation (SDE) models from user-defined drift and diffusion rate functions. Use SDE models 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 function enables you to simulate any 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.
- 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	<p>User-defined drift-rate function, denoted by F. DriftRate is a function that returns an NVARS-by-1 drift-rate vector when called with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t. <p>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.</p>
DiffusionRate	<p>User-defined diffusion-rate function, denoted by G. DiffusionRate is a function that returns an NVARS-by-NBROWNS diffusion-rate matrix when called with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t. <p>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.</p>

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

StartTime	<p>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.</p>
-----------	---

StartState	<p>Scalar, <code>NVARS</code>-by-1 column vector, or <code>NVARS</code>-by-<code>NTRIALS</code> matrix of initial values of the state variables.</p> <p>If <code>StartState</code> is a scalar, <code>sde</code> applies the same initial value to all state variables on all trials.</p> <p>If <code>StartState</code> is a column vector, <code>sde</code> applies a unique initial value to each state variable on all trials.</p> <p>If <code>StartState</code> is a matrix, <code>sde</code> applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for <code>StartState</code>, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify <code>Correlation</code> as an <code>NBROWNS</code>-by-<code>NBROWNS</code> positive semidefinite matrix, or as a deterministic function $C(t)$ that accepts the current time t and returns an <code>NBROWNS</code>-by-<code>NBROWNS</code> positive semidefinite correlation matrix.</p> <p>A <code>Correlation</code> matrix represents a static condition.</p> <p>As a deterministic function of time, <code>Correlation</code> allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for <code>Correlation</code>, the default is an <code>NBROWNS</code>-by-<code>NBROWNS</code> identity matrix representing independent Gaussian processes.</p>
Simulation	<p>A user-defined simulation function or SDE simulation method. If you do not specify a value for <code>Simulation</code>, the default method is simulation by Euler approximation (<code>simByEuler</code>).</p>

Output Arguments

SDE	<p>Stochastic differential equation model (SDE) with the following parameters:</p> <ul style="list-style-type: none"> • <code>StartTime</code>: 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

Examples

- “Base SDE Models” on page 17-15
- Implementing Multidimensional Equity Market Models, Implementation 1: Using SDE Objects

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | diffusion

sdeddo

Create `sdeddo` model from Drift and Diffusion objects

Syntax

```
SDE = sdeddo(DriftRate, DiffusionRate)
```

```
SDE = sdeddo(DriftRate, DiffusionRate, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Class

SDEDDO

Description

This constructor creates and displays `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:

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

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 <code>drift</code> that encapsulates a user-defined drift-rate specification, represented as F .
DiffusionRate	Object of class <code>diffusion</code> that encapsulates a user-defined diffusion-rate specification, represented as G .

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

StartTime	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for <code>StartTime</code> , the default is 0.
StartState	Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables. If <code>StartState</code> is a scalar, <code>sdeddo</code> applies the same initial value to all state variables on all trials.

	<p>If StartState is a column vector, sdeddo applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, sdeddo applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>
Simulation	<p>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).</p>

Output Arguments

SDE	<p>Object of class sdeddo with the following parameters:</p> <ul style="list-style-type: none"> • 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

Examples

- “Drift and Diffusion Models” on page 17-18
- Implementing Multidimensional Equity Market Models, Implementation 2: Using SDEDDO Objects

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`sde` | `drift` | `diffusion`

sdeld

Construct stochastic differential equation from linear drift-rate models

Syntax

```
SDE = sdeld(A, B, Alpha, Sigma)
```

```
SDE = sdeld(A, B, Alpha, Sigma, 'Name1', Value1, 'Name2',  
Value2, ...)
```

Class

SDELD

Description

This constructor creates and displays 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).

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:

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

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 a .
- V is an NVARs-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time \mathbf{t} as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

A	<p>A represents the parameter A. If you specify A as an array, it must be an NVARs-by-1 column vector of intercepts. As a deterministic function of time, when A is called with a real-valued scalar time \mathbf{t} as its only input, A must produce an NVARs-by-1 column vector. If you specify A as a function of time and state, it must generate an NVARs-by-1 column vector of intercepts when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
----------	---

B	<p>B represents the parameter B. If you specify B as an array, it must be an NVARS-by-NVARS matrix of state vector coefficients. As a deterministic function of time, when B is called with a real-valued scalar time t as its only input, B must produce an NVARS-by-NVARS matrix. If you specify B as a function of time and state, it must generate an NVARS-by-NVARS matrix of state vector coefficients when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t.
Alpha	<p>Alpha determines the format of the parameter D. If you specify Alpha as an array, it represents an NVARS-by-1 column vector of exponents. As a deterministic function of time, when Alpha is called with a real-valued scalar time t as its only input, Alpha must produce an NVARS-by-1 column vector. If you specify it as a function of time and state, it must return an NVARS-by-1 column vector of exponents when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V. 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. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARS-by-NBROWNS matrix. If you specify it as a function of time and state, it must generate an NVARS-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARS-by-1 state vector X_t.

Note: Although the constructor does not enforce restrictions on the signs of **Alpha** or **Sigma**, each parameter is usually specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, sdeld applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, sdeld applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, sdeld applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	<p>Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.</p> <p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p>

	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	<p>Object of class sdelD with the following parameters:</p> <ul style="list-style-type: none"> • 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
------------	---

Examples

- “Linear Drift Models” on page 17-22
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | diffusion | sdeddo

sdemrd

Construct stochastic differential equation from mean-reverting drift-rate models

Syntax

```
SDE = sdemrd(Speed, Level, Alpha, Sigma)
```

```
SDE = sdemrd(Speed, Level, Alpha, Sigma, 'Name1', Value1, 'Name2', Value2, ...)
```

Class

SDEMIRD

Description

This constructor creates and displays 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). Use **SDEMIRD** 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 **sde1d**). 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:

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

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 a .
- V is an NVARs-by-NBROWNS instantaneous volatility rate matrix.
- dW_t is an NBROWNS-by-1 Brownian motion vector.

Input Arguments

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.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time t as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

The required input parameters are:

Speed	<p>Speed represents the parameter S. 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). As a deterministic function of time, when Speed is called with a real-valued scalar time t as its only input, Speed must produce an NVARs-by-NVARs matrix. If you specify Speed as a function time and state, Speed calculates the speed of mean reversion. This function must generate an NVARs-by-NVARs matrix of reversion rates when called with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t.
--------------	---

	<ul style="list-style-type: none"> • An NVARs-by-1 state vector X_t.
Level	<p>Level represents the parameter L. If you specify Level as an array, it must be an NVARs-by-1 column vector of reversion levels. As a deterministic function of time, when Level is called with a real-valued scalar time t as its only input, Level must produce an NVARs-by-1 column vector. If you specify Level as a function of time and state, must generate an NVARs-by-1 column vector of reversion levels when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Alpha	<p>Alpha determines the format of the parameter D. If you specify Alpha as an array, it must be an NVARs-by-1 column vector of exponents. As a deterministic function of time, when Alpha is called with a real-valued scalar time t as its only input, Alpha must produce an NVARs-by-1 column vector. If you specify it as a function of time and state, it must return an NVARs-by-1 column vector of exponents when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.
Sigma	<p>Sigma represents the parameter V. 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. As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARs-by-NBROWNS matrix. If you specify it as a function of time and state, it must generate an NVARs-by-NBROWNS matrix of volatility rates when invoked with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An NVARs-by-1 state vector X_t.

Note: Although the constructor does not enforce restrictions on the signs of these input arguments, each argument is usually specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

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	<p>Scalar, NVARs-by-1 column vector, or NVARs-by-NTRIALS matrix of initial values of the state variables.</p> <p>If StartState is a scalar, sdemrd applies the same initial value to all state variables on all trials.</p> <p>If StartState is a column vector, sdemrd applies a unique initial value to each state variable on all trials.</p> <p>If StartState is a matrix, sdemrd applies a unique initial value to each state variable on each trial.</p> <p>If you do not specify a value for StartState, all variables start at 1.</p>
Correlation	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). 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.

	<p>A Correlation matrix represents a static condition.</p> <p>As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.</p> <p>If you do not specify a value for Correlation, the default is an NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes.</p>
Simulation	<p>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).</p>

Output Arguments

SDE	<p>Object of class SDEMRD, with the following parameters:</p> <ul style="list-style-type: none"> • 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
------------	---

Examples

See “Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEM RD) Models” on page 17-27.

More About

Algorithms

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.

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

drift | diffusion | sdeddo

simByEuler

Euler simulation of stochastic differential equations (SDEs)

Syntax

```
[Paths, Times, Z] = simByEuler(MDL, NPERIODS)
```

```
[Paths, Times, Z] = simByEuler(MDL, NPERIODS, 'Name1', Value1,  
'Name2', Value2, ...)
```

Classes

All classes in the “SDE Class Hierarchy” on page 17-5.

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

MDL	Stochastic differential equation object created with the <code>sdeddo</code> constructor.
NPERIODS	Positive scalar integer number of simulation periods. The value of <code>NPERIODS</code> determines the number of rows of the simulated output series.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- 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:

NTRIALS	Positive scalar integer number of simulated trials (sample paths) of <code>NPERIODS</code> 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 <code>NPERIODS</code> -by-1 column vector of positive time increments between observations. <code>DeltaTime</code> 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 <code>DeltaTime</code>). The <code>simByEuler</code> method partitions each time increment dt into <code>NSTEPS</code> subintervals of length $dt/NSTEPS$, and refines the simulation by evaluating the simulated state vector at <code>NSTEPS - 1</code> intermediate points. Although <code>simByEuler</code> 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

	<p>continuous-time process. If you do not specify a value for <code>NSTEPS</code>, the default is 1, indicating no intermediate evaluation.</p>
Antithetic	<p>Scalar logical flag that indicates whether <code>simByEuler</code> uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes).</p> <p>When <code>Antithetic</code> is <code>TRUE</code> (logical 1), <code>simByEuler</code> performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs:</p> <ul style="list-style-type: none"> • 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. <p>If you specify <code>Antithetic</code> to be any value other than <code>TRUE</code>, <code>simByEuler</code> assumes that it is <code>FALSE</code> (logical 0) by default, and does not perform antithetic sampling. When you specify an input noise process (see <code>Z</code>), <code>simByEuler</code> ignores the value of <code>Antithetic</code>.</p>
Z	<p>Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. Specify this argument as a function, or as an (<code>NPERIODS * NSTEPS</code>)-by-<code>NBROWNS</code>-by-<code>NTRIALS</code> three-dimensional array of dependent random variates. If you specify <code>Z</code> as a function, it must return an <code>NBROWNS</code>-by-1 column vector, and you must call it with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An <code>NVARS</code>-by-1 state vector X_t. <p>If you do not specify a value for <code>Z</code>, <code>simByEuler</code> generates correlated Gaussian variates based on the <code>Correlation</code> member of the <code>SDE</code> object.</p>
StorePaths	<p>Scalar logical flag that indicates how the output array <code>Paths</code> is stored and returned to the caller. If <code>StorePaths</code> is <code>TRUE</code> (the default value) or is unspecified, <code>simByEuler</code> returns <code>Paths</code> as a three-dimensional time series array. If <code>StorePaths</code> is <code>FALSE</code> (logical 0), <code>simByEuler</code> returns the <code>Paths</code> output array as an empty matrix.</p>

Processes	<p>Function or cell array of functions that indicates a sequence of end-of-period processes or state vector adjustments of the form</p> $X_t = P(t, X_t)$ <p><code>simByEuler</code> 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.</p> <p>If you specify more than one processing function, <code>simByEuler</code> 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.</p> <p>If you do not specify a processing function, <code>simByEuler</code> makes no adjustments and performs no processing.</p>
-----------	---

Output Arguments

Paths	(NPERIODS + 1)-by-NVARS-by-NTRIALS three-dimensional time series array, consisting of simulated paths of correlated state variables. For a given trial, each row of <code>Paths</code> is the transpose of the state vector X_t at time t . When the input flag <code>StorePaths = FALSE</code> , <code>simByEuler</code> returns <code>Paths</code> as an empty matrix.
Times	(NPERIODS + 1)-by-1 column vector of observation times associated with the simulated paths. Each element of <code>Times</code> is associated with the corresponding row of <code>Paths</code> .
Z	(NPERIODS * NSTEPS)-by-NBROWNS-by-NTRIALS three-dimensional time series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation.

Examples

Implementing Multidimensional Equity Market Models, Implementation 5: Using the `simByEuler` Method

More About

Algorithms

- 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).

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

simBySolution | simulate

simBySolution

Simulate approximate solution of diagonal-drift HWV and GBM processes

Syntax

```
[Paths, Times, Z] = simBySolution(MDL, NPERIODS)
```

```
[Paths, Times, Z] = simBySolution(MDL, 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:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$$

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:

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$$

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, 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

MDL	Hull-White/Vasicek (HW) 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

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:

- 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:

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: <ul style="list-style-type: none"> • 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.

	<p>If you specify <code>Antithetic</code> to be any value other than <code>TRUE</code>, <code>simBySolution</code> assumes that it is <code>FALSE</code> (logical 0) by default, and does not perform antithetic sampling. When you specify an input noise process (see <code>Z</code>), <code>simBySolution</code> ignores the value of <code>Antithetic</code>.</p>
<code>Z</code>	<p>Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. Specify this argument as a function, or as an <code>(NPERIODS * NSTEPS)-by-NBROWNS-by-NTRIALS</code> array of dependent random variates. If you specify <code>Z</code> as a function, it must return an <code>NBROWNS-by-1</code> column vector, and you must call it with two inputs:</p> <ul style="list-style-type: none"> • A real-valued scalar observation time t. • An <code>NVARS-by-1</code> state vector X_t. <p>If you do not specify a value for <code>Z</code>, <code>simBySolution</code> generates correlated Gaussian variates based on the <code>Correlation</code> member of the SDE object.</p>
<code>StorePaths</code>	<p>Scalar logical flag that indicates how <code>simBySolution</code> stores the output array <code>Paths</code> and returns it to the caller. If <code>StorePaths</code> is <code>TRUE</code> (the default value) or is unspecified, <code>simBySolution</code> returns <code>Paths</code> as a three-dimensional time series array. If <code>StorePaths</code> is <code>FALSE</code> (logical 0), <code>simBySolution</code> returns the <code>Paths</code> output array as an empty matrix.</p>

Processes	<p>Function or cell array of functions that indicates a sequence of end-of-period processes or state vector adjustments of the form</p> $X_t = P(t, X_t)$ <p><code>simBySolution</code> 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.</p> <p>If you specify more than one processing function, <code>simBySolution</code> 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.</p> <p>If you do not specify a processing function, <code>simBySolution</code> makes no adjustments and performs no processing.</p>
-----------	--

Output Arguments

Paths	<p>(<code>NPERIODS + 1</code>)-by-<code>NVARS</code>-by-<code>NTRIALS</code> three-dimensional time series array, consisting of simulated paths of correlated state variables. For a given trial, each row of <code>Paths</code> is the transpose of the state vector X_t at time t. When the input flag <code>StorePaths = FALSE</code>, <code>simBySolution</code> returns <code>Paths</code> as an empty matrix.</p>
Times	<p>(<code>NPERIODS + 1</code>)-by-1 column vector of observation times associated with the simulated paths. Each element of <code>Times</code> is associated with the corresponding row of <code>Paths</code>.</p>
Z	<p>(<code>NPERIODS * NSTEPS</code>)-by-<code>NBROWNS</code>-by-<code>NTRIALS</code> three-dimensional time series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation.</p>

Examples

Implementing Multidimensional Equity Market Models, Implementation 6: Using GBM Simulation Methods

More About

Algorithms

- 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).

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `simulate`

simulate

Simulate multivariate stochastic differential equations (SDEs)

Syntax

```
[Paths, Times, Z] = simulate(MDL, ...)
```

Classes

All classes in the “SDE Class Hierarchy” on page 17-5.

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.

`[Paths, Times, Z] = simulate(MDL, ...)` 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

MDL	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 three-dimensional time series array, consisting of simulated paths of correlated state variables. For a given trial, each row of <code>Paths</code> 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 <code>Times</code> is associated with a corresponding row of <code>Paths</code> .
Z	NTIMES-by-NBROWNS-by-NTRIALS three-dimensional time series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drove the simulated results found in <code>Paths</code> . NTIMES is the number of time steps at which <code>simulate</code> samples the state vector. NTIMES includes intermediate times designed to improve accuracy, which <code>simulate</code> does not necessarily report in the <code>Paths</code> 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);
```

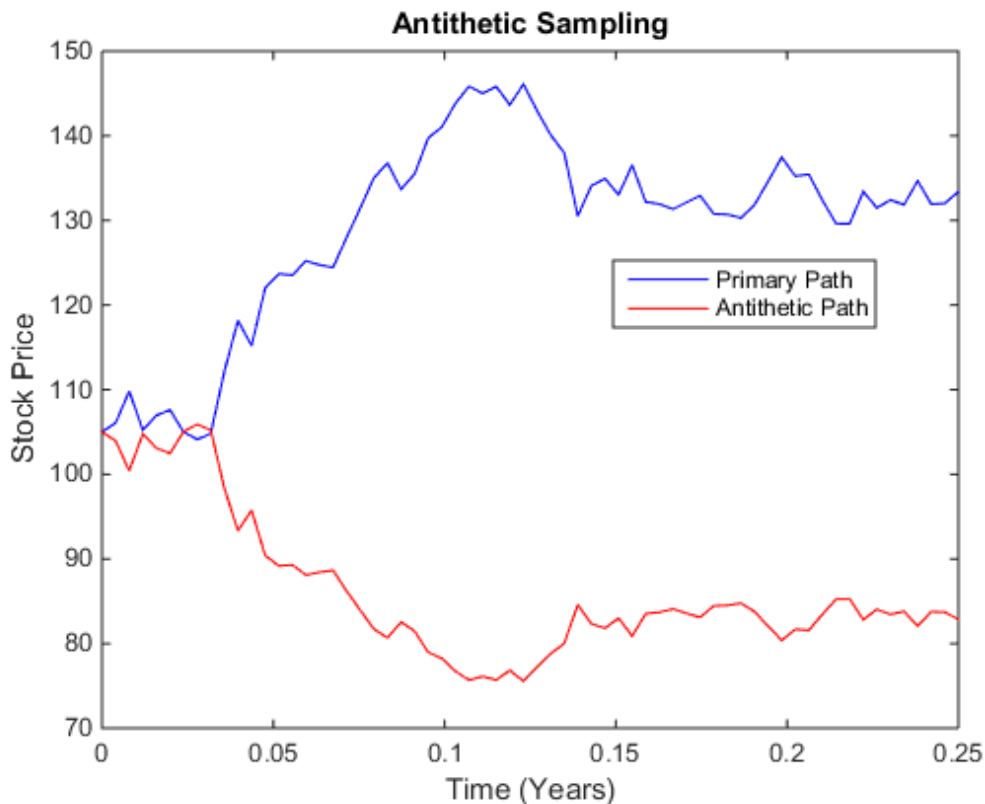
- 2 Perform a very small-scale simulation that explicitly returns two simulated paths:

```
rng('default')           % make output reproducible
[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. For more information on using an end-of-period processing function, see “Pricing Equity Options”.

- 1 Simulate 200 paths using the processing function method:

```
rng('default')           % make output reproducible
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);
nPaths   = 200;               % # of paths = 100 sets of pairs
f        = Example_BarrierOption(nPeriods, nPaths);
simulate(obj, 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:    6.6572
Standard Error of Price:           0.7292
Confidence Interval Lower Bound:    5.2280
Confidence Interval Upper Bound:    8.0864

```

References

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.

Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

simByEuler | simBySolution

second

Seconds of date or time

Syntax

```
Seconds = second(Date)
```

Description

`Seconds = second(Date)` returns the seconds given a serial date number or a date string.

Examples

```
Seconds = second(738647.558427893)
```

or

```
Seconds = second('06-May-2022, 13:24:08.17')
```

returns

```
Seconds =
```

```
8.1700
```

See Also

`datevec` | `hour` | `minute`

selectreturn

Portfolio configurations from 3-D efficient frontier

Syntax

```
PortConfigs = selectreturn(AllMean, All Covariance, Target)
```

Arguments

AllMean	Number of curves (NCURVES) by 1 cell array where each element is a 1-by-NASSETS (number of assets) vector of the expected asset returns used to generate each curve on the surface.
AllCovariance	NCURVES-by-1 cell array where each element is an NASSETS-by-NASSETS vector of the covariance matrix used to generate each curve on the surface.
Target	Target return value for each curve in the frontier.

Description

`PortConfigs = selectreturn(AllMean, All Covariance, Target)` returns the portfolio configurations for a target return given the average return and covariance for a rolling efficient frontier.

`PortConfigs` is a NASSETS-by-NCURVES matrix of asset allocation weights needed to obtain the target rate of return.

See Also

frontier

setAssetList

Class: Portfolio

Set up list of identifiers for assets

Syntax

```
obj = setAssetList(obj, varargin)
```

Description

`obj = setAssetList(obj, varargin)` sets up a list of identifiers for the assets.

Tips

- You can also use dot notation to set up list of identifiers for assets.

```
obj = obj.setAssetList(varargin);
```

- To clear an `AssetList`, call this method with `[]` or `{[]}`.

Input Arguments

obj

Portfolio object [Portfolio].

varargin

Asset identifiers. Either a comma-separated list of strings or a vector cell array of strings. Each string is an asset identifier.

If an asset list is entered as an input, this method overwrites an existing asset list in the object if one exists.

If no asset list is entered as an input, three actions can occur:

- If `NumAssets` is nonempty and `AssetList` is empty, `AssetList` becomes a numbered list of assets with default names according to the hidden property in `defaultforAssetList ('Asset')`.
- If `NumAssets` is nonempty and `AssetList` is nonempty, nothing happens.
- If `NumAssets` is empty and `AssetList` is empty, the default `NumAssets =1` is set and a default asset list is created (`'Asset1'`).

For more information on setting up a list of asset identifiers, see “Common Operations on the Portfolio Object” on page 4-34.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Note: The underlying object has a number of public hidden properties to format the asset list:

- `defaultforAssetList` — Default name for assets (`'Asset'`). Change this name to create default asset names such as `'ETF'`, `'Bond'`, and so on.
 - `sortAssetList` — Reserved for future implementation.
 - `uppercaseAssetList` — If `true`, make all asset identifiers uppercase strings. Otherwise does nothing. Default is `false`.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Default List of Asset Names with Three Assets

Create a default list of asset names with three assets.

```
p = Portfolio('NumAssets',3);  
p = setAssetList(p);  
disp(p.AssetList);  
  
    'Asset1'    'Asset2'    'Asset3'
```

Create an Explicitly Named List of Asset Names with Three Assets

Create a list of asset names for three equities AGG, EEM, and VEU.

```
p = Portfolio;  
p = setAssetList(p, 'AGG', 'EEM', 'VEU');  
disp(p.AssetList);  
  
    'AGG'    'EEM'    'VEU'
```

- “Setting Up a List of Asset Identifiers” on page 4-35
- “Common Operations on the Portfolio Object” on page 4-34

See Also

Portfolio

setAssetList

Class: PortfolioCVaR

Set up list of identifiers for assets for PortfolioCVaR object

Syntax

```
obj = setAssetList(obj,varargin)
```

Description

`obj = setAssetList(obj,varargin)` sets up a list of identifiers for the assets.

Tips

- You can also dot notation to set up list of identifiers for assets.

```
obj = obj.setAssetList(varargin);
```

- To clear an `AssetList`, call this method with `[]` or `{[]}`.

Input Arguments

obj

CVaR portfolio object [`PortfolioCVaR`].

varargin

Asset identifiers. Either a comma-separated list of strings or a vector cell array of strings. Each string is an asset identifier.

If an asset list is entered as an input, this method overwrites an existing asset list in the object if one exists.

If no asset list is entered as an input, three actions can occur:

- If `NumAssets` is nonempty and `AssetList` is empty, `AssetList` becomes a numbered list of assets with default names according to the hidden property in `defaultforAssetList('Asset')`.
- If `NumAssets` is nonempty and `AssetList` is nonempty, nothing happens.
- If `NumAssets` is empty and `AssetList` is empty, the default `NumAssets = 1` is set and a default asset list is created (`'Asset1'`).

For more information on setting up a list of asset identifiers, see “Common Operations on the PortfolioCVaR Object” on page 5-34.

Output Arguments

obj

Updated CVaR portfolio object [`PortfolioCVaR`].

Note: The underlying object has many public hidden properties to format the asset list:

- `defaultforAssetList` — Default name for assets (`'Asset'`). Change this name to create default asset names such as `'ETF'`, `'Bond'`, and so on.
 - `sortAssetList` — Reserved for future implementation.
 - `uppercaseAssetList` — If `true`, make all asset identifiers uppercase strings. Otherwise, it does nothing. Default is `false`.
-

Attributes

<code>Access</code>	<code>public</code>
<code>Static</code>	<code>false</code>
<code>Hidden</code>	<code>false</code>

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Default List of Asset Names with Three Assets

Create a default list of asset names with three assets.

```
p = PortfolioCVaR('NumAssets',3);  
p = setAssetList(p);  
disp(p.AssetList);  
  
    'Asset1'    'Asset2'    'Asset3'
```

Create an Explicitly Named List of Asset Names with Three Assets

Create a list of asset names for three equities AGG, EEM, and VEU.

```
p = PortfolioCVaR;  
p = setAssetList(p, 'AGG', 'EEM', 'VEU');  
disp(p.AssetList);  
  
    'AGG'    'EEM'    'VEU'
```

- “Setting Up a List of Asset Identifiers” on page 5-35
- “Common Operations on the PortfolioCVaR Object” on page 5-34

See Also

PortfolioCVaR

setAssetList

Class: PortfolioMAD

Set up list of identifiers for assets for PortfolioMAD object

Syntax

```
obj = setAssetList(obj, varargin)
```

Description

`obj = setAssetList(obj, varargin)` sets up a list of identifiers for the assets.

Tips

- You can also use dot notation to set up list of identifiers for assets.

```
obj = obj.setAssetList(varargin);
```

- To clear an `AssetList`, call this method with `[]` or `{[]}`.

Input Arguments

obj

MAD portfolio object [`PortfolioMAD`].

varargin

Asset identifiers specified as either a comma-separated list of strings or a vector cell array of strings. Each string is an asset identifier.

If an asset list is entered as an input, this method overwrites an existing asset list in the object if one exists.

If no asset list is entered as an input, three actions can occur:

- If `NumAssets` is nonempty and `AssetList` is empty, `AssetList` becomes a numbered list of assets with default names according to the hidden property in `defaultforAssetList ('Asset')`.
- If `NumAssets` is nonempty and `AssetList` is nonempty, nothing happens.
- If `NumAssets` is empty and `AssetList` is empty, the default `NumAssets = 1` is set and a default asset list is created (`'Asset1'`).

This method accepts empty strings as identifiers and, depending upon the existing value for `NumAssets`, either pads or truncates the list of assets with default names according to the property `defaultforAssetList` (currently `'Asset'`).

For more information on setting up a list of asset identifiers, see “Common Operations on the PortfolioCVaR Object” on page 5-34.

Output Arguments

obj

Updated MAD portfolio object [`PortfolioMAD`].

Note: The underlying object has many public hidden properties to format the asset list:

- `defaultforAssetList` — Default name for assets (`'Asset'`). Change this name to create default asset names such as `'ETF'`, `'Bond'`, and so on.
 - `sortAssetList` — Reserved for future implementation.
 - `uppercaseAssetList` — If `true`, make all asset identifiers uppercase strings. Otherwise, it does nothing. Default is `false`.
-

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Default List of Asset Names with Three Assets

Create a default list of asset names with three assets.

```
p = PortfolioMAD('NumAssets',3);  
p = setAssetList(p);  
disp(p.AssetList);  
  
    'Asset1'    'Asset2'    'Asset3'
```

Create an Explicitly Named List of Asset Names with Three Assets

Create a list of asset names for three equities AGG, EEM, and VEU.

```
p = PortfolioMAD;  
p = setAssetList(p, 'AGG', 'EEM', 'VEU');  
disp(p.AssetList);  
  
    'AGG'    'EEM'    'VEU'
```

- “Setting Up a List of Asset Identifiers” on page 6-35
- “Common Operations on the PortfolioMAD Object” on page 6-34

See Also

PortfolioMAD

setAssetMoments

Class: Portfolio

Set moments (mean and covariance) of asset returns

Syntax

```
obj = setAssetMoments(obj,AssetMean)
obj = setAssetMoments(obj,AssetMean,AssetCovar,NumAssets)
```

Description

`obj = setAssetMoments(obj,AssetMean)` sets the mean of asset returns.

`obj = setAssetMoments(obj,AssetMean,AssetCovar,NumAssets)` sets moments (mean and covariance) of the asset returns with additional options for `AssetCovar` and `NumAssets`.

Tips

- You can also use dot notation to set moments (mean and covariance) of the asset returns.

```
obj = obj.setAssetMoments(obj, AssetMean, AssetCovar, NumAssets);
```

- To clear `AssetMean` and `AssetCovar`, use this method to set these respective inputs to `[]`.

Input Arguments

obj

Portfolio object [Portfolio].

AssetMean

Mean of asset returns [vector].

Note: If `AssetMean` is a scalar and the number of assets is known, scalar expansion occurs. If the number of assets cannot be determined, this method assumes that `NumAssets = 1`.

AssetCovar

(Optional) Covariance of asset returns [matrix].

Note: `AssetCovar` must be a symmetric positive-semidefinite matrix.

If `AssetCovar` is a scalar and the number of assets is known, a diagonal matrix is formed with the scalar value along the diagonals. If it is not possible to determine the number of assets, this method assumes that `NumAssets = 1`.

If `AssetCovar` is a vector, a diagonal matrix is formed with the vector along the diagonal.

NumAssets

(Optional) Number of assets [integer].

Note: If `NumAssets` is not already set in the object, `NumAssets` can be entered to resolve array expansions with `AssetMean` or `AssetCovar`.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false

Hidden false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Asset Moments

Set the asset moment properties, given the mean and covariance of asset returns in the variables `m` and `C`.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = setAssetMoments(p, m, C);
[assetmean, assetcovar] = getAssetMoments(p)
```

assetmean =

```
0.0042
0.0083
0.0100
0.0150
```

assetcovar =

```
0.0005    0.0003    0.0002         0
0.0003    0.0024    0.0017    0.0010
0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

- “Asset Returns and Moments of Asset Returns” on page 4-41

See Also

`estimateAssetMoments` | `estimateFrontierByRisk` | `Portfolio`

More About

- “Portfolio Optimization Theory” on page 4-2

setBounds

Class: Portfolio

Set up bounds for portfolio weights

Syntax

```
obj = setBounds(obj, LowerBound)
obj = setBounds(obj, LowerBound, UpperBound, NumAssets)
```

Description

`obj = setBounds(obj, LowerBound)` sets up the lower bound for portfolio weights.

`obj = setBounds(obj, LowerBound, UpperBound, NumAssets)` sets up bounds for portfolio weights with additional options for `UpperBound`, and `NumAssets`.

Given bound constraints `LowerBound` and `UpperBound`, every weight in a portfolio `Port` must satisfy the following:

$$\text{LowerBound} \leq \text{Port} \leq \text{UpperBound}$$

Tips

You can also use dot notation to set up the bounds for portfolio weights.

```
obj = obj.setBounds(LowerBound, UpperBound, NumAssets);
```

Input Arguments

obj

Portfolio object [Portfolio].

LowerBound

Lower-bound weight for each asset [vector].

UpperBound

(Optional) Upper-bound weight each asset [vector].

NumAssets

(Optional) Number of assets in portfolio [scalar]. `NumAssets` cannot be used to change the dimension of a `Portfolio` object.

Note: If either `LowerBound` or `UpperBound` are input as empties with `[]`, the corresponding attributes in the `Portfolio` object are cleared and set to `[]`.

If `LowerBound` or `UpperBound` are specified as scalars and `NumAssets` exists or can be imputed, then they undergo scalar expansion. The default value for `NumAssets` is 1.

If both `LowerBound` and `UpperBound` exist and they are not ordered correctly, this method switches bounds if necessary.

Output Arguments

obj

Updated `Portfolio` object [`Portfolio`].

Attributes

<code>Access</code>	<code>public</code>
<code>Static</code>	<code>false</code>
<code>Hidden</code>	<code>false</code>

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Bound Constraints

Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. To set the bound constraints for a balanced fund.

```
lb = [ 0.5; 0.25 ];  
ub = [ 0.75; 0.5 ];  
  
p = Portfolio;  
p = setBounds(p, lb, ub);  
disp(p.NumAssets);  
disp(p.LowerBound);  
disp(p.UpperBound);
```

```
2
```

```
0.5000
```

```
0.2500
```

```
0.7500
```

```
0.5000
```

- “Working with Bound Constraints” on page 4-60

See Also

getBounds | Portfolio

More About

- “Working with Portfolio Constraints” on page 4-57

setBounds

Class: PortfolioCVaR

Set up bounds for portfolio weights in PortfolioCVaR object

Syntax

```
obj = setBounds(obj, LowerBound)
obj = setBounds(obj, LowerBound, UpperBound, NumAssets)
```

Description

`obj = setBounds(obj, LowerBound)` sets up the lower bound for portfolio weights.

`obj = setBounds(obj, LowerBound, UpperBound, NumAssets)` sets up bounds for portfolio weights with additional options for `UpperBound`, and `NumAssets`.

Given bound constraints `LowerBound` and `UpperBound`, every weight in a portfolio `Port` must satisfy the following:

$$\text{LowerBound} \leq \text{Port} \leq \text{UpperBound}$$

Tips

You can also use dot notation to set up the bounds for portfolio weights.

```
obj = obj.setBounds(LowerBound, UpperBound, NumAssets);
```

Input Arguments

obj

CVaR portfolio object [`PortfolioCVaR`].

LowerBound

Lower-bound weight for each asset [vector].

UpperBound

(Optional) Upper-bound weight for each asset [vector].

NumAssets

(Optional) Number of assets in CVaR portfolio object [scalar]. **NumAssets** cannot be used to change the dimension of a CVaR portfolio object.

Note: If **LowerBound** or **UpperBound** is input as empty with [], the corresponding attributes in the CVaR portfolio object are cleared and set to [].

If **LowerBound** or **UpperBound** is specified as scalars and **NumAssets** exists or can be imputed, then they undergo scalar expansion. The default value for **NumAssets** is 1.

If both **LowerBound** and **UpperBound** exist and they are not ordered correctly, this method switches bounds if necessary.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Bound Constraints

Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. To set the bound constraints for a balanced fund.

```
lb = [ 0.5; 0.25 ];  
ub = [ 0.75; 0.5 ];
```

```
p = PortfolioCvAR;  
p = setBounds(p, lb, ub);  
disp(p.NumAssets);  
disp(p.LowerBound);  
disp(p.UpperBound);
```

```
2
```

```
0.5000
```

```
0.2500
```

```
0.7500
```

```
0.5000
```

- “Working with Bound Constraints” on page 5-59

See Also

[getBounds](#) | [PortfolioCvAR](#)

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setBounds

Class: PortfolioMAD

Set up bounds for portfolio weights in PortfolioMAD object

Syntax

```
obj = setBounds(obj, LowerBound)
obj = setBounds(obj, LowerBound, UpperBound, NumAssets)
```

Description

`obj = setBounds(obj, LowerBound)` sets up the lower bound for portfolio weights.

`obj = setBounds(obj, LowerBound, UpperBound, NumAssets)` sets up bounds for portfolio weights with additional options for `UpperBound`, and `NumAssets`.

Given bound constraints `LowerBound` and `UpperBound`, every weight in a portfolio `Port` must satisfy the following:

$$\text{LowerBound} \leq \text{Port} \leq \text{UpperBound}$$

Tips

You can also use dot notation to set up the bounds for portfolio weights.

```
obj = obj.setBounds(LowerBound, UpperBound, NumAssets);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

LowerBound

Lower-bound weight for each asset [vector].

UpperBound

(Optional) Upper-bound weight for each asset [vector].

NumAssets

(Optional) Number of assets in a PortfolioMAD object [scalar]. `NumAssets` cannot be used to change the dimension of a PortfolioMAD object.

Note: If `LowerBound` or `UpperBound` is input as empty with `[]`, the corresponding attributes in the MAD portfolio object are cleared and set to `[]`.

If `LowerBound` or `UpperBound` is specified as scalars and `NumAssets` exists or can be imputed, then they undergo scalar expansion. The default value for `NumAssets` is 1.

If both `LowerBound` and `UpperBound` exist and they are not ordered correctly, this method switches bounds if necessary.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Bound Constraints

Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. To set the bound constraints for a balanced fund.

```
lb = [ 0.5; 0.25 ];  
ub = [ 0.75; 0.5 ];  
  
p = PortfolioMAD;  
p = setBounds(p, lb, ub);  
disp(p.NumAssets);  
disp(p.LowerBound);  
disp(p.UpperBound);
```

```
2
```

```
0.5000
```

```
0.2500
```

```
0.7500
```

```
0.5000
```

- “Working with Bound Constraints” on page 6-59

See Also

getBounds | PortfolioMAD

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setBudget

Class: Portfolio

Set up budget constraints

Syntax

```
obj = setBudget(obj, LowerBudget)
obj = setBudget(obj, LowerBudget, UpperBudget)
```

Description

`obj = setBudget(obj, LowerBudget)` sets up the lower budget constraint.

`obj = setBudget(obj, LowerBudget, UpperBudget)` sets up budget constraints with an additional option for `UpperBudget`.

Tips

You can also use dot notation to set up the budget constraints.

```
obj = obj.setBudget(LowerBudget, UpperBudget);
```

Input Arguments

obj

Portfolio object [Portfolio].

LowerBudget

Lower-bound for budget constraint [scalar].

UpperBudget

(Optional) Upper-bound for budget constraint [scalar].

Note: Given bounds for a budget constraints in either `LowerBudget` or `UpperBudget`, budget constraints require any portfolio in `Port` to satisfy:

```
LowerBudget <= sum(Port) <= UpperBudget
```

One or both constraints may be specified. The usual budget constraint for a fully-invested portfolio is to have `LowerBudget = UpperBudget = 1`. However, if the portfolio has allocations in cash, the budget constraints can be used to specify the cash constraints. For example, if the portfolio can hold between 0% and 10% in cash, the budget constraint would be set up with

```
obj = setBudget(obj, 0.9, 1)
```

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Budget Constraint

Assume you have a fund that permits up to 10% leverage, which means that your portfolio can be from 100% to 110% invested in risky assets. Given a Portfolio object `p`, set the budget constraint.

```
p = Portfolio;
```

```
p = setBudget(p, 1, 1.1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
1
```

```
1.1000
```

- “Working with Budget Constraints” on page 4-62

See Also

getBudget | Portfolio

More About

- “Working with Portfolio Constraints” on page 4-57

setBudget

Class: PortfolioCVaR

Set up budget constraints for PortfolioCVaR object

Syntax

```
obj = setBudget(obj, LowerBudget)
obj = setBudget(obj, LowerBudget, UpperBudget)
```

Description

`obj = setBudget(obj, LowerBudget)` sets up the lower budget constraint.

`obj = setBudget(obj, LowerBudget, UpperBudget)` sets up budget constraints with an additional option for `UpperBudget`.

Tips

You can also use dot notation to set up the budget constraints.

```
obj = obj.setBudget(LowerBudget, UpperBudget);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

LowerBudget

Lower bound for budget constraint [scalar].

UpperBudget

(Optional) Upper bound for budget constraint [scalar].

Note: Given bounds for budget constraints in either `LowerBudget` or `UpperBudget`, budget constraints require any portfolio in `Port` to satisfy:

```
LowerBudget <= sum(Port) <= UpperBudget
```

One or both constraints can be specified. The usual budget constraint for a fully invested portfolio is to have `LowerBudget = UpperBudget = 1`. However, if the portfolio has allocations in cash, the budget constraints can be used to specify the cash constraints. For example, if the portfolio can hold from 0% to 10% in cash, the budget constraint would be set up with

```
obj = setBudget(obj, 0.9, 1)
```

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Budget Constraint

Assume you have a fund that permits up to 10% leverage, which means that your portfolio can be from 100% to 110% invested in risky assets. Given a CVaR portfolio object `p`, set the budget constraint.

```
p = PortfolioCVaR;
```



```
p = setBudget(p, 1, 1.1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
1
```

```
1.1000
```

- “Working with Budget Constraints” on page 5-61

See Also

getBudget | PortfolioCVar

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setBudget

Class: PortfolioMAD

Set up budget constraints for PortfolioMAD object

Syntax

```
obj = setBudget(obj, LowerBudget)
obj = setBudget(obj, LowerBudget, UpperBudget)
```

Description

`obj = setBudget(obj, LowerBudget)` sets up the lower budget constraint.

`obj = setBudget(obj, LowerBudget, UpperBudget)` sets up budget constraints with an additional option for `UpperBudget`.

Tips

You can also use dot notation to set up the budget constraints.

```
obj = obj.setBudget(LowerBudget, UpperBudget);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

LowerBudget

Lower bound for budget constraint [scalar].

UpperBudget

(Optional) Upper bound for budget constraint [scalar].

Note: Given bounds for budget constraints in either `LowerBudget` or `UpperBudget`, budget constraints require any portfolio in `Port` to satisfy:

```
LowerBudget <= sum(Port) <= UpperBudget
```

One or both constraints can be specified. The usual budget constraint for a fully invested portfolio is to have `LowerBudget = UpperBudget = 1`. However, if the portfolio has allocations in cash, the budget constraints can be used to specify the cash constraints. For example, if the portfolio can hold from 0% to 10% in cash, the budget constraint would be set up with

```
obj = setBudget(obj, 0.9, 1)
```

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Budget Constraint

Assume you have a fund that permits up to 10% leverage, which means that your portfolio can be from 100% to 110% invested in risky assets. Given PortfolioMAD object `p`, set the budget constraint.

```
p = PortfolioMAD;
```

```
p = setBudget(p, 1, 1.1);  
disp(p.LowerBudget);  
disp(p.UpperBudget);
```

```
1
```

```
1.1000
```

- “Working with Budget Constraints” on page 6-61

See Also

getBudget | PortfolioMAD

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setCosts

Class: Portfolio

Set up proportional transaction costs

Syntax

```
obj = setCosts(obj, BuyCost)
obj = setCosts(obj, BuyCost, SellCost, InitPort, NumAssets)
```

Description

`obj = setCosts(obj, BuyCost)` sets up proportional transaction costs to purchase assets.

`obj = setCosts(obj, BuyCost, SellCost, InitPort, NumAssets)` sets up proportional transaction costs with additional options specified for `SellCost`, `InitPort`, and `NumAssets`.

Given proportional transaction costs and an initial portfolio in the variables `BuyCost`, `SellCost`, and `InitPort`, the transaction costs for any portfolio `Port` reduce expected portfolio return by:

$$\text{BuyCost}' * \max\{0, \text{Port} - \text{InitPort}\} + \text{SellCost}' * \max\{0, \text{InitPort} - \text{Port}\}$$

Tips

- You can also use dot notation to set up proportional transaction costs.

```
obj = obj.setCosts(BuyCost, SellCost, InitPort, NumAssets);
```

- If `BuyCost` or `SellCost` are input as empties with `[]`, the corresponding attributes in the Portfolio object are cleared and set to `[]`. If `InitPort` is set to empty with `[]`, it will only be cleared and set to `[]` if `BuyCost`, `SellCost`, and `Turnover` are also empty. Otherwise, it is an error.

Input Arguments

obj

Portfolio object [Portfolio].

BuyCost

Proportional transaction cost to purchase each asset [vector].

SellCost

Proportional transaction cost to sell each asset [vector].

InitPort

Initial or current portfolio weights [vector].

Note: If no `InitPort` is specified, that value is assumed to be 0.

NumAssets

Number of assets in portfolio [scalar]. `NumAssets` cannot be used to change the dimension of a `Portfolio` object.

Note: If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is 1.

Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Up Transaction Costs

Assume you have the same costs and initial portfolio as in the previous example. Given a Portfolio object `p` with an initial portfolio already set, use the `setCosts` method to set up transaction costs.

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = Portfolio('InitPort', x0);
p = setCosts(p, bc, sc);
```

```
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5
```

```
0.0013
0.0013
0.0013
0.0013
0.0013
```

```
0.0013
0.0070
0.0013
```

0.0013
0.0024

0.4000
0.2000
0.2000
0.1000
0.1000

- “Working with Transaction Costs” on page 4-52

See Also

Portfolio | setInitPort

setCosts

Class: PortfolioCVaR

Set up proportional transaction costs for PortfolioCVaR object

Syntax

```
obj = setCosts(obj, BuyCost)
obj = setCosts(obj, BuyCost, SellCost, InitPort, NumAssets)
```

Description

`obj = setCosts(obj, BuyCost)` sets up proportional transaction costs to purchase assets.

`obj = setCosts(obj, BuyCost, SellCost, InitPort, NumAssets)` sets up proportional transaction costs with additional options specified for `SellCost`, `InitPort`, and `NumAssets`.

Given proportional transaction costs and an initial portfolio in the variables `BuyCost`, `SellCost`, and `InitPort`, the transaction costs for any portfolio `Port` reduce expected portfolio return by:

$$\text{BuyCost}' * \max\{0, \text{Port} - \text{InitPort}\} + \text{SellCost}' * \max\{0, \text{InitPort} - \text{Port}\}$$

Tips

- You can also use dot notation to set up proportional transaction costs.

```
obj = obj.setCosts(BuyCost, SellCost, InitPort, NumAssets);
```

- If `BuyCost` or `SellCost` is input as empty with `[]`, the corresponding attributes in the CVaR portfolio object are cleared and set to `[]`. If `InitPort` is set to empty with `[]`, it is only cleared and set to `[]` if `BuyCost`, `SellCost`, and `Turnover` are also empty. Otherwise, it is an error.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

BuyCost

Proportional transaction cost to purchase each asset [vector].

SellCost

Proportional transaction cost to sell each asset [vector].

InitPort

Initial or current portfolio weights [vector].

Note: If no `InitPort` is specified, that value is assumed to be 0.

NumAssets

Number of assets in a portfolio [scalar]. `NumAssets` cannot be used to change the dimension of a `PortfolioCVaR` object.

Note: If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is 1.

Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Up Transaction Costs

Given a CVaR portfolio object `p` with an initial portfolio already set, use the `setCosts` method to set up transaction costs.

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = PortfolioCVaR('InitPort', x0);
p = setCosts(p, bc, sc);
```

```
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5
```

```
0.0013
0.0013
0.0013
0.0013
0.0013
```

```
0.0013
0.0070
0.0013
0.0013
0.0024
```

0.4000
0.2000
0.2000
0.1000
0.1000

- “Working with Transaction Costs” on page 5-51

See Also

`getCosts` | `PortfolioCVaR` | `setInitPort`

setCosts

Class: PortfolioMAD

Set up proportional transaction costs for PortfolioMAD object

Syntax

```
obj = setCosts(obj,BuyCost)
obj = setCosts(obj,BuyCost,SellCost,InitPort,NumAssets)
```

Description

`obj = setCosts(obj,BuyCost)` sets up proportional transaction costs to purchase assets.

`obj = setCosts(obj,BuyCost,SellCost,InitPort,NumAssets)` sets up proportional transaction costs with additional options specified for `SellCost`, `InitPort`, and `NumAssets`.

Given proportional transaction costs and an initial portfolio in the variables `BuyCost`, `SellCost`, and `InitPort`, the transaction costs for any portfolio `Port` reduce expected portfolio return by:

$$\text{BuyCost}' * \max\{0, \text{Port} - \text{InitPort}\} + \text{SellCost}' * \max\{0, \text{InitPort} - \text{Port}\}$$

Tips

- You can also use dot notation to set up proportional transaction costs:

```
obj = obj.setCosts(BuyCost, SellCost, InitPort, NumAssets);
```

- If `BuyCost` or `SellCost` is input as empty with `[]`, the corresponding attributes in the PortfolioMAD object are cleared and set to `[]`. If `InitPort` is set to empty with `[]`, it is only cleared and set to `[]` if `BuyCost`, `SellCost`, and `Turnover` are also empty. Otherwise, it is an error.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

BuyCost

Proportional transaction cost to purchase each asset [vector].

SellCost

Proportional transaction cost to sell each asset [vector].

InitPort

Initial or current portfolio weights [vector].

Note: If no `InitPort` is specified, that value is assumed to be 0.

NumAssets

Number of assets in a portfolio [scalar]. `NumAssets` cannot be used to change the dimension of a `PortfolioMAD` object.

Note: If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is 1.

Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Up Transaction Costs

Given PortfolioMAD object `p` with an initial portfolio already set, use the `setCosts` method to set up transaction costs.

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
```

```
p = PortfolioMAD('InitPort', x0);
p = setCosts(p, bc, sc);
```

```
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
5
```

```
0.0013
0.0013
0.0013
0.0013
0.0013
```

```
0.0013
0.0070
0.0013
0.0013
0.0024
```

0.4000
0.2000
0.2000
0.1000
0.1000

- “Working with Transaction Costs” on page 6-51

See Also

`getCosts` | `PortfolioMAD` | `setInitPort`

setDefaultConstraints

Class: Portfolio

Set up portfolio constraints with nonnegative weights that sum to 1

Syntax

```
obj = setDefaultConstraints(obj)
obj = setDefaultConstraints(obj, NumAssets)
```

Description

`obj = setDefaultConstraints(obj)` sets up the portfolio constraints with nonnegative weights that must sum to 1.

`obj = setDefaultConstraints(obj, NumAssets)` sets up the portfolio constraints with nonnegative weights that must sum to 1 with an additional option for `NumAssets`.

A "default" portfolio set has `LowerBound = 0` and `LowerBudget = UpperBudget = 1` such that a portfolio `Port` must satisfy $\text{sum}(\text{Port}) = 1$ with $\text{Port} \geq 0$.

Tips

- You can also use dot notation to set up the default portfolio set.

```
obj = obj.setDefaultConstraints(NumAssets);
```

- This method does not modify any existing constraints in a Portfolio object other than the bound and budget constraints. If an `UpperBound` constraint exists, it is cleared and set to `[]`.

Input Arguments

obj

Portfolio object [Portfolio].

NumAssets

(Optional) Number of assets in portfolio [scalar]. `NumAssets` cannot be used to change the dimension of a `Portfolio` object.

Default: 1

Output Arguments

obj

Updated `Portfolio` object [`Portfolio`].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Define Default Constraints for the Portfolio Object

Assuming you have 20 assets, you can define the "default" portfolio set.

```
p = Portfolio('NumAssets', 20);  
p = setDefaultConstraints(p);  
disp(p);
```

```
Portfolio with properties:
```

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []
```

```
AssetMean: []
AssetCovar: []
Turnover: []
BuyTurnover: []
SellTurnover: []
Name: []
NumAssets: 20
AssetList: []
InitPort: []
AInequality: []
bInequality: []
AEquality: []
bEquality: []
LowerBound: [20x1 double]
UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
LowerGroup: []
UpperGroup: []
GroupA: []
GroupB: []
LowerRatio: []
UpperRatio: []
```

- “Setting Default Constraints for Portfolio Weights” on page 4-57

See Also

[getBounds](#) | [Portfolio](#) | [setBounds](#) | [setBudget](#)

More About

- “Default Portfolio Problem” on page 4-14

setDefaultConstraints

Class: PortfolioCVaR

Set up portfolio constraints with nonnegative weights that sum to 1 for PortfolioCVaR object

Syntax

```
obj = setDefaultConstraints(obj)
obj = setDefaultConstraints(obj, NumAssets)
```

Description

`obj = setDefaultConstraints(obj)` sets up portfolio constraints with nonnegative weights that must sum to 1.

`obj = setDefaultConstraints(obj, NumAssets)` sets up portfolio constraints with nonnegative weights that must sum to 1 with an additional option for NumAssets.

A "default" portfolio set has `LowerBound = 0` and `LowerBudget = UpperBudget = 1` such that a portfolio `Port` must satisfy `sum(Port) = 1` with `Port >= 0`.

Tips

- You can also use dot notation to set up the default portfolio set.

```
obj = obj.setDefaultConstraints(NumAssets);
```
- This method does not modify any existing constraints in a PortfolioCVaR object other than the bound and budget constraints. If an `UpperBound` constraint exists, it is cleared and set to `[]`.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

NumAssets

(Optional) Number of assets in a portfolio [scalar]. `NumAssets` cannot be used to change the dimension of a `PortfolioCVaR` object.

Default: 1

Output Arguments

obj

Updated CVaR portfolio object [`PortfolioCVaR`].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Define Default Constraints for the PortfolioCVaR Object

Assuming you have 20 assets, you can define the "default" portfolio set.

```
p = PortfolioCVaR('NumAssets', 20);  
p = setDefaultConstraints(p);  
disp(p);
```

```
PortfolioCVaR with properties:
```

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []
```

```
ProbabilityLevel: []
  Turnover: []
  BuyTurnover: []
  SellTurnover: []
  NumScenarios: []
    Name: []
    NumAssets: 20
    AssetList: []
    InitPort: []
  AInequality: []
  bInequality: []
    AEquality: []
    bEquality: []
  LowerBound: [20x1 double]
  UpperBound: []
  LowerBudget: 1
  UpperBudget: 1
  GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
      GroupA: []
      GroupB: []
  LowerRatio: []
  UpperRatio: []
```

- “Setting Default Constraints for Portfolio Weights” on page 5-56

See Also

`getBounds` | `PortfolioCVaR` | `setBounds` | `setBudget`

More About

- “Default Portfolio Problem” on page 5-14

setDefaultConstraints

Class: PortfolioMAD

Set up portfolio constraints with nonnegative weights that sum to 1 for PortfolioMAD object

Syntax

```
obj = setDefaultConstraints(obj)
obj = setDefaultConstraints(obj, NumAssets)
```

Description

`obj = setDefaultConstraints(obj)` sets up portfolio constraints with nonnegative weights that must sum to 1.

`obj = setDefaultConstraints(obj, NumAssets)` sets up portfolio constraints with nonnegative weights that must sum to 1 with an additional option for NumAssets.

A "default" portfolio set has `LowerBound = 0` and `LowerBudget = UpperBudget = 1` such that a portfolio `Port` must satisfy $\text{sum}(\text{Port}) = 1$ with $\text{Port} \geq 0$.

Tips

- You can also use dot notation to set up the default portfolio set.

```
obj = obj.setDefaultConstraints(NumAssets);
```
- This method does not modify any existing constraints in a PortfolioMAD object other than the bound and budget constraints. If an `UpperBound` constraint exists, it is cleared and set to `[]`.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

NumAssets

(Optional) Number of assets in a PortfolioMAD object [scalar]. `NumAssets` cannot be used to change the dimension of a PortfolioMAD object.

Default: 1

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Define Default Constraints for the PortfolioMAD Object

Assuming you have 20 assets, you can define the "default" portfolio set.

```
p = PortfolioMAD('NumAssets', 20);  
p = setDefaultConstraints(p);  
disp(p);
```

```
PortfolioMAD with properties:
```

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []
```



```
Turnover: []
BuyTurnover: []
SellTurnover: []
NumScenarios: []
  Name: []
  NumAssets: 20
  AssetList: []
  InitPort: []
AInequality: []
bInequality: []
  AEquality: []
  bEquality: []
LowerBound: [20x1 double]
UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
  LowerGroup: []
  UpperGroup: []
    GroupA: []
    GroupB: []
  LowerRatio: []
  UpperRatio: []
```

- “Setting Default Constraints for Portfolio Weights” on page 6-56

See Also

[getBounds](#) | [PortfolioMAD](#) | [setBounds](#) | [setBudget](#)

More About

- “Default Portfolio Problem” on page 6-14

setEquality

Class: Portfolio

Set up linear equality constraints for portfolio weights

Syntax

```
obj = setEquality(obj,AEquality,bEquality)
```

Description

`obj = setEquality(obj,AEquality,bEquality)` sets up linear equality constraints for portfolio weights.

Given linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

$$AEquality * Port = bEquality$$

Tips

- You can also use dot notation to set up linear equality constraints for portfolio weights.

```
obj = obj.setEquality(AEquality, bEquality);
```
- Linear equality constraints can be removed from a Portfolio object by entering [] for each property you want to remove.

Input Arguments

obj

Portfolio object [Portfolio].

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Note: An error results if **AEquality** is empty and **bEquality** is nonempty or if **AEquality** is nonempty and **bEquality** is empty.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Linear Equality Constraints

Suppose you have a portfolio of five assets, and you want to ensure that the first three assets are 50% of your portfolio. Given a Portfolio object **p**, set the linear equality constraints with the following.

```
A = [ 1 1 1 0 0 ];  
b = 0.5;
```

```
p = Portfolio;  
p = setEquality(p, A, b);  
  
disp(p.NumAssets);  
disp(p.AEquality);  
disp(p.bEquality);  
  
5  
  
1    1    1    0    0  
  
0.5000
```

- “Working with Linear Equality Constraints” on page 4-69

See Also

`addEquality` | `getEquality` | `Portfolio`

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setEquality

Class: PortfolioCVaR

Set up linear equality constraints for portfolio weights in PortfolioCVaR object

Syntax

```
obj = setEquality(obj,AEquality,bEquality)
```

Description

`obj = setEquality(obj,AEquality,bEquality)` sets up linear equality constraints for portfolio weights.

Given linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

$$AEquality * Port = bEquality$$

Tips

- You can also use dot notation to set up linear equality constraints for portfolio weights.

```
obj = obj.setEquality(AEquality, bEquality);
```

- Linear equality constraints can be removed from a PortfolioCVaR object by entering `[]` for each property you want to remove.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Note: An error results if **AEquality** is empty and **bEquality** is nonempty or if **AEquality** is nonempty and **bEquality** is empty.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Linear Equality Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioCVaR object **p**, set the linear equality constraints and obtain the values for **AEquality** and **bEquality**:

```
A = [ 1 1 1 0 0 ];  
b = 0.5;
```

```
p = PortfolioCVaR;  
p = setEquality(p, A, b);  
disp(p.NumAssets);  
disp(p.AEquality);  
disp(p.bEquality);  
  
5  
  
1    1    1    0    0  
  
0.5000
```

- “Working with Linear Equality Constraints” on page 5-68

See Also

addEquality | getEquality | PortfolioCVaR

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setEquality

Class: PortfolioMAD

Set up linear equality constraints for portfolio weights in PortfolioMAD object

Syntax

```
obj = setEquality(obj,AEquality,bEquality)
```

Description

`obj = setEquality(obj,AEquality,bEquality)` sets up linear equality constraints for portfolio weights.

Given linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

$$\text{AEquality} * \text{Port} = \text{bEquality}$$

Tips

- You can also use dot notation to set up linear equality constraints for portfolio weights:

```
obj = obj.setEquality(AEquality, bEquality);
```
- Linear equality constraints can be removed from a PortfolioMAD object by entering `[]` for each property you want to remove.

Input Arguments

obj

MAD portfolio object [`PortfolioMAD`].

AEquality

Matrix to form linear equality constraints [matrix].

bEquality

Vector to form linear equality constraints [vector].

Note: An error results if **AEquality** is empty and **bEquality** is nonempty or if **AEquality** is nonempty and **bEquality** is empty.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Linear Equality Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioMAD object `p`, set the linear equality constraints and obtain the values for **AEquality** and **bEquality**:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
```

```
p = PortfolioMAD;  
p = setEquality(p, A, b);  
[AEquality, bEquality] = getEquality(p)
```

```
AEquality =
```

```
    1    1    1    0    0
```

```
bEquality =
```

```
    0.5000
```

- “Working with Linear Equality Constraints” on page 6-68

See Also

[addEquality](#) | [getEquality](#) | [PortfolioMAD](#)

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setGroupRatio

Class: Portfolio

Set up group ratio constraints for portfolio weights

Syntax

```
obj = setGroupRatio(obj,GroupA)
obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,
UpperRatio)
```

Description

`obj = setGroupRatio(obj,GroupA)` sets up the group ratio constraints for portfolio weights with lower bound on the ratio between groups.

`obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` to set up group ratio constraints for portfolio weights with an additional option specified for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and `LowerRatio` or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Tips

- You can also use dot notation to set up group ratio constraints for portfolio weight.

```
obj = obj.setGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio);
```

- To remove group ratio constraints, enter empty arrays for the corresponding arrays. To add to existing group ratio constraints, use `addGroupRatio`.

Input Arguments

obj

Portfolio object [Portfolio].

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

Note: The group matrices `GroupA` and `GroupB` are usually indicators of membership in groups, which means that their elements are usually either 0 or 1. Because of this interpretation, `GroupA` and `GroupB` matrices can be either logical or numerical arrays.

LowerRatio

Lower bound for ratio of `GroupB` groups to `GroupA` groups [vector].

Note: If input is scalar, `LowerRatio` undergoes scalar expansion to be conformable with the group matrices.

UpperRatio

(Optional) Upper bound for ratio of `GroupB` groups to `GroupA` groups [vector].

Note: If input is scalar, `UpperRatio` undergoes scalar expansion to be conformable with the group matrices.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Group Ratio Constraints

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolio never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6
```

```
1 1 1 0 0 0
```

```
0 0 0 1 1 1
0.5000
```

- “Working with Group Ratio Constraints” on page 4-66

See Also

`addGroupRatio` | `getGroupRatio` | `Portfolio`

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setGroupRatio

Class: PortfolioCVaR

Set up group ratio constraints for portfolio weights in PortfolioCVaR object

Syntax

```
obj = setGroupRatio(obj,GroupA)
obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,
UpperRatio)
```

Description

`obj = setGroupRatio(obj,GroupA)` sets up the group ratio constraints for CVaR portfolio weights with lower bound on the ratio between groups.

`obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` to set up group ratio constraints for portfolio weights with an additional option specified for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and `LowerRatio` or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Tips

- You can also use dot notation to set up group ratio constraints for portfolio weight.

```
obj = obj.setGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio);
```

- To remove group ratio constraints, enter empty arrays for the corresponding arrays. To add to existing group ratio constraints, use `addGroupRatio`.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

Note: The group matrices `GroupA` and `GroupB` often indicate membership in groups, which means that their elements are usually either 0 or 1. Because of this interpretation, `GroupA` and `GroupB` matrices can be either logical or numerical arrays.

LowerRatio

Lower bound for ratio of `GroupB` groups to `GroupA` groups [vector].

Note: If input is scalar, `LowerRatio` undergoes scalar expansion to conform with the group matrices.

UpperRatio

(Optional) Upper bound for ratio of `GroupB` groups to `GroupA` groups [vector].

Note: If input is scalar, `UpperRatio` undergoes scalar expansion to conform with the group matrices.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Group Ratio Constraints

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolio never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6
```

```
1 1 1 0 0 0
```

```
0 0 0 1 1 1
0.5000
```

- “Working with Group Ratio Constraints”

See Also

`addGroupRatio` | `getGroupRatio` | `PortfolioCVaR`

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setGroupRatio

Class: PortfolioMAD

Set up group ratio constraints for portfolio weights in PortfolioMAD object

Syntax

```
obj = setGroupRatio(obj,GroupA)
obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,
UpperRatio)
```

Description

`obj = setGroupRatio(obj,GroupA)` sets up the group ratio constraints for MAD portfolio weights with lower bound on the ratio between groups.

`obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` to set up group ratio constraints for portfolio weights with an additional option specified for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and `LowerRatio` or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Tips

- You can also use dot notation to set up group ratio constraints for portfolio weight.

```
obj = obj.setGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio);
```

- To remove group ratio constraints, enter empty arrays for the corresponding arrays. To add to existing group ratio constraints, use `addGroupRatio`.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

GroupA

Matrix that forms base groups for comparison [matrix].

GroupB

Matrix that forms comparison groups [matrix].

Note: The group matrices `GroupA` and `GroupB` often indicate membership in groups, which means that their elements are usually either 0 or 1. Because of this interpretation, `GroupA` and `GroupB` matrices can be either logical or numerical arrays.

LowerRatio

Lower bound for ratio of `GroupB` groups to `GroupA` groups [vector].

Note: If input is scalar, `LowerRatio` undergoes scalar expansion to conform with the group matrices.

UpperRatio

(Optional) Upper bound for ratio of `GroupB` groups to `GroupA` groups [vector].

Note: If input is scalar, `UpperRatio` undergoes scalar expansion to conform with the group matrices.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Group Ratio Constraints

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolio never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ]; % financial companies
GB = [ false false false true true true ]; % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6
```

```
1 1 1 0 0 0
```

```
0 0 0 1 1 1
0.5000
```

- “Working with Group Ratio Constraints” on page 6-65

See Also

`addGroupRatio` | `getGroupRatio` | `PortfolioMAD`

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setGroups

Class: Portfolio

Set up group constraints for portfolio weights

Syntax

```
obj = setGroups(obj, GroupMatrix, LowerGroup)
obj = setGroups(obj, GroupMatrix, LowerGroup, UpperGroup)
```

Description

`obj = setGroups(obj, GroupMatrix, LowerGroup)` sets up group constraints for portfolio weights subject to a lower bound on groups.

`obj = setGroups(obj, GroupMatrix, LowerGroup, UpperGroup)` to set up group constraints for portfolio weights with an additional options specified for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

$$\text{LowerGroup} \leq \text{GroupMatrix} * \text{Port} \leq \text{UpperGroup}$$

Tips

- You can also use dot notation to set up group constraints for portfolio weights.
`obj = obj.setGroups(GroupMatrix, LowerGroup, UpperGroup);`
- To remove group constraints, enter empty arrays for the corresponding arrays. To add to existing group constraints, use `addGroups`.

Input Arguments

obj

Portfolio object [Portfolio].

GroupMatrix

Group constraint matrix [matrix].

Note: The group matrix `GroupMatrix` is usually an indicator of membership in groups, which means that its elements are usually either 0 or 1. Because of this interpretation, `GroupMatrix` can be either a logical or numerical matrix.

LowerGroup

Lower bound for group constraints [vector].

Note: If input is scalar, `LowerGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

UpperGroup

Upper bound for group constraints [vector].

Note: If input is scalar, `UpperGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Group Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a Portfolio object `p`, set the group constraints with the following.

```
G = [ true true true false false ];  
p = Portfolio;  
p = setGroups(p, G, [], 0.3);
```

```
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.UpperGroup);
```

```
5  
1    1    1    0    0  
0.3000
```

- “Working with Group Constraints” on page 4-63

See Also

[addGroups](#) | [getGroups](#) | [Portfolio](#)

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setGroups

Class: PortfolioCVaR

Set up group constraints for portfolio weights in PortfolioCVaR object

Syntax

```
obj = setGroups(obj, GroupMatrix, LowerGroup)
obj = setGroups(obj, GroupMatrix, LowerGroup, UpperGroup)
```

Description

`obj = setGroups(obj, GroupMatrix, LowerGroup)` sets up group constraints for portfolio weights subject to a lower bound on groups.

`obj = setGroups(obj, GroupMatrix, LowerGroup, UpperGroup)` to set up group constraints for portfolio weights with an additional options specified for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a PortfolioCVaR object `Port` must satisfy the following:

```
LowerGroup <= GroupMatrix * Port <= UpperGroup
```

Tips

- You can also use dot notation to set up group constraints for portfolio weights.

```
obj = obj.setGroups(GroupMatrix, LowerGroup, UpperGroup);
```
- To remove group constraints, enter empty arrays for the corresponding arrays. To add to existing group constraints, use `addGroups`.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

GroupMatrix

Group constraint matrix [matrix].

Note: The group matrix `GroupMatrix` is usually an indicator of membership in groups, which means that its elements are usually either 0 or 1. Because of this interpretation, `GroupMatrix` can be either a logical or numerical matrix.

LowerGroup

Lower bound for group constraints [vector].

Note: If input is scalar, `LowerGroup` undergoes scalar expansion to conform with `GroupMatrix`.

UpperGroup

Upper bound for group constraints [vector].

Note: If input is scalar, `UpperGroup` undergoes scalar expansion to conform with `GroupMatrix`.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Group Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a CVaR portfolio object `p`, set the group constraints with the following.

```
G = [ true true true false false ];  
p = PortfolioCVaR;  
p = setGroups(p, G, [], 0.3);
```

```
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.UpperGroup);
```

```
5  
1    1    1    0    0  
0.3000
```

- “Working with Group Constraints” on page 5-63

See Also

`addGroups` | `getGroups` | `PortfolioCVaR`

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setGroups

Class: PortfolioMAD

Set up group constraints for portfolio weights in PortfolioMAD object

Syntax

```
obj = setGroups(obj,GroupMatrix,LowerGroup)
obj = setGroups(obj,GroupMatrix,LowerGroup,UpperGroup)
```

Description

`obj = setGroups(obj,GroupMatrix,LowerGroup)` sets up group constraints for portfolio weights subject to a lower bound on groups.

`obj = setGroups(obj,GroupMatrix,LowerGroup,UpperGroup)` to set up group constraints for portfolio weights with an additional options specified for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a `PortfolioMAD` object `Port` must satisfy the following:

$$\text{LowerGroup} \leq \text{GroupMatrix} * \text{Port} \leq \text{UpperGroup}$$

Tips

- You can also use dot notation to set up group constraints for portfolio weights.
`obj = obj.setGroups(GroupMatrix, LowerGroup, UpperGroup);`
- To remove group constraints, enter empty arrays for the corresponding arrays. To add to existing group constraints, use `addGroups`.

Input Arguments

obj

MAD portfolio object [`PortfolioMAD`].

GroupMatrix

Group constraint matrix [matrix].

Note: The group matrix `GroupMatrix` is usually an indicator of membership in groups, which means that its elements are usually either 0 or 1. Because of this interpretation, `GroupMatrix` can be either a logical or numerical matrix.

LowerGroup

Lower bound for group constraints [vector].

Note: If input is scalar, `LowerGroup` undergoes scalar expansion to conform with `GroupMatrix`.

UpperGroup

Upper bound for group constraints [vector].

Note: If input is scalar, `UpperGroup` undergoes scalar expansion to conform with `GroupMatrix`.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Group Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given PortfolioMAD object `p`, set the group constraints with the following.

```
G = [ true true true false false ];  
p = PortfolioMAD;  
p = setGroups(p, G, [], 0.3);
```

```
disp(p.NumAssets);  
disp(p.GroupMatrix);  
disp(p.UpperGroup);
```

```
5  
1    1    1    0    0  
0.3000
```

- “Working with Group Constraints” on page 6-62

See Also

`addGroups` | `getGroups` | `PortfolioMAD`

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setInequality

Class: Portfolio

Set up linear inequality constraints for portfolio weights

Syntax

```
obj = setInequality(obj,AInequality,bInequality)
```

Description

`obj = setInequality(obj,AInequality,bInequality)` sets up linear inequality constraints for portfolio weights.

Given a linear inequality constraint matrix **AInequality** and vector **bInequality**, every weight in a portfolio **Port** must satisfy the following:

$$\text{AInequality} * \text{Port} \leq \text{bInequality}$$

Tips

- You can also use dot notation to set up linear inequality constraints for portfolio weights.

```
obj = obj.setInequality(AInequality, bInequality);
```

- To remove inequality constraints enter empty arguments. To add to existing inequality constraints, use `addInequality`.

Input Arguments

obj

Portfolio object [Portfolio].

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Note: An error results if **AInequality** is empty and **bInequality** is nonempty or if **AInequality** is nonempty and **bInequality** is empty.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Linear Inequality Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given a Portfolio object **p**, set the linear inequality constraints with the following.

```
A = [ 1 1 1 0 0 ];  
b = 0.5;
```

```
p = Portfolio;  
p = setInequality(p, A, b);
```

```
disp(p.NumAssets);  
disp(p.AInequality);  
disp(p.bInequality);
```

```
5
```

```
1     1     1     0     0
```

```
0.5000
```

- “Working with Linear Inequality Constraints” on page 4-71

See Also

`addInequality` | `getInequality` | `Portfolio`

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setInequality

Class: PortfolioCVaR

Set up linear inequality constraints for portfolio weights in PortfolioCVaR object

Syntax

```
obj = setInequality(obj,AInequality,bInequality)
```

Description

`obj = setInequality(obj,AInequality,bInequality)` sets up linear inequality constraints for portfolio weights.

Given a linear inequality constraint matrix **AInequality** and vector **bInequality**, every weight in portfolio **Port** must satisfy the following:

$$\text{AInequality} * \text{Port} \leq \text{bInequality}$$

Tips

- You can also use dot notation to set up linear inequality constraints for portfolio weights.

```
obj = obj.setInequality(AInequality, bInequality);
```

- To remove inequality constraints, enter empty arguments. To add to existing inequality constraints, use `addInequality`.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Note: An error results if **AInequality** is empty and **bInequality** is nonempty or if **AInequality** is nonempty and **bInequality** is empty.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Linear Inequality Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given a CVaR portfolio object **p**, set the linear inequality constraints with the following.

```
A = [ 1 1 1 0 0 ];  
b = 0.5;
```

```
p = PortfolioCVaR;  
p = setInequality(p, A, b);
```

```
disp(p.NumAssets);  
disp(p.AInequality);  
disp(p.bInequality);
```

```
5  
1 1 1 0 0  
0.5000
```

- “Working with Linear Inequality Constraints” on page 5-70

See Also

[addInequality](#) | [getInequality](#) | [PortfolioCVaR](#)

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setInequality

Class: PortfolioMAD

Set up linear inequality constraints for portfolio weights in PortfolioMAD object

Syntax

```
obj = setInequality(obj,AInequality,bInequality)
```

Description

`obj = setInequality(obj,AInequality,bInequality)` sets up linear inequality constraints for portfolio weights.

Given a linear inequality constraint matrix `AInequality` and vector `bInequality`, every weight in portfolio `Port` must satisfy the following:

$$\text{AInequality} * \text{Port} \leq \text{bInequality}$$

Tips

- You can also use dot notation to set up linear inequality constraints for portfolio weights.

```
obj = obj.setInequality(AInequality, bInequality);
```

- To remove inequality constraints, enter empty arguments. To add to existing inequality constraints, use `addInequality`.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

AInequality

Matrix to form linear inequality constraints [matrix].

bInequality

Vector to form linear inequality constraints [vector].

Note: An error results if **AInequality** is empty and **bInequality** is nonempty or if **AInequality** is nonempty and **bInequality** is empty.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Linear Inequality Constraints

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given PortfolioMAD object **p**, set the linear inequality constraints with the following.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
```

```
p = PortfolioMAD;  
p = setInequality(p, A, b);
```

```
disp(p.NumAssets);  
disp(p.AInequality);  
disp(p.bInequality);
```

```
5
```

```
1     1     1     0     0
```

```
0.5000
```

- “Working with Linear Inequality Constraints” on page 6-70

See Also

`addInequality` | `getInequality` | `PortfolioMAD`

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setInitPort

Class: Portfolio

Set up initial or current portfolio

Syntax

```
obj = setInitPort(obj, InitPort)
obj = setInitPort(obj, InitPort, NumAssets)
```

Description

`obj = setInitPort(obj, InitPort)` sets up the initial or current portfolio.

`obj = setInitPort(obj, InitPort, NumAssets)` sets up the initial or current portfolio with an additional options specified for `NumAssets`.

Tips

- You can also use dot notation to set up an initial or current portfolio.
`obj = obj.setInitPort(InitPort, NumAssets);`
- To remove an initial portfolio, call this method with an empty argument [] for `InitPort`.

Input Arguments

obj

Portfolio object [Portfolio].

InitPort

Initial or current portfolio weights [vector].

Note: If `InitPort` is specified as a scalar and `NumAssets` exists, then `InitPort` undergoes scalar expansion.

NumAssets

(Optional) Number of assets in portfolio [scalar].

Note: If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is 1.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set the InitPort Property

Given an initial portfolio in `x0`, use the `setInitPort` method to set the `InitPort` property.

```
p = Portfolio('NumAssets', 4);  
x0 = [ 0.3; 0.2; 0.2; 0.0 ];  
p = setInitPort(p, x0);
```

```
disp(p.InitPort);  
  
    0.3000  
    0.2000  
    0.2000  
         0
```

Set InitPort to Create an Equally-Weighted Portfolio of Four Assets

Create an equally weighted portfolio of four assets using the `setInitPort` method.

```
p = Portfolio('NumAssets', 4);  
p = setInitPort(p, 1/4, 4);  
disp(p.InitPort);
```

```
    0.2500  
    0.2500  
    0.2500  
    0.2500
```

- “Setting Up an Initial or Current Portfolio” on page 4-37

See Also

`Portfolio` | `setCosts` | `setTurnover`

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setInitPort

Class: PortfolioCVaR

Set up initial or current portfolio for PortfolioCVaR object

Syntax

```
obj = setInitPort(obj, InitPort)
obj = setInitPort(obj, InitPort, NumAssets)
```

Description

`obj = setInitPort(obj, InitPort)` sets up the initial or current portfolio.

`obj = setInitPort(obj, InitPort, NumAssets)` sets up the initial or current portfolio with an additional options specified for `NumAssets`.

Tips

- You can also use dot notation to set up an initial or current portfolio.

```
obj = obj.setInitPort(InitPort, NumAssets);
```
- To remove an initial portfolio, call this method with an empty argument [] for `InitPort`.

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

InitPort

Initial or current portfolio weights [vector].

Note: If `InitPort` is specified as a scalar and `NumAssets` exists, then `InitPort` undergoes scalar expansion.

NumAssets

(Optional) Number of assets in CVaR portfolio [scalar].

Note: If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is 1.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set the InitPort Property

Given an initial portfolio in `x0`, use the `setInitPort` method to set the `InitPort` property.

```
p = PortfolioCVaR('NumAssets', 4);  
x0 = [ 0.3; 0.2; 0.2; 0.0 ];  
p = setInitPort(p, x0);
```

```
disp(p.InitPort);  
  
    0.3000  
    0.2000  
    0.2000  
         0
```

Set InitPort to Create an Equally-Weighted Portfolio of Four Assets

Create an equally weighted portfolio of four assets using the `setInitPort` method.

```
p = PortfolioCvAR('NumAssets', 4);  
p = setInitPort(p, 1/4, 4);  
disp(p.InitPort);
```

```
    0.2500  
    0.2500  
    0.2500  
    0.2500
```

- “Setting Up an Initial or Current Portfolio” on page 5-37

See Also

`PortfolioCvAR` | `setCosts` | `setTurnover`

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setInitPort

Class: PortfolioMAD

Set up initial or current portfolio for PortfolioMAD object

Syntax

```
obj = setInitPort(obj, InitPort)
obj = setInitPort(obj, InitPort, NumAssets)
```

Description

`obj = setInitPort(obj, InitPort)` sets up the initial or current portfolio.

`obj = setInitPort(obj, InitPort, NumAssets)` sets up the initial or current portfolio with an additional options specified for `NumAssets`.

Tips

- You can also use dot notation to set up an initial or current portfolio.
`obj = obj.setInitPort(InitPort, NumAssets);`
- To remove an initial portfolio, call this method with an empty argument [] for `InitPort`.

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

InitPort

Initial or current portfolio weights [vector].

Note: If `InitPort` is specified as a scalar and `NumAssets` exists, then `InitPort` undergoes scalar expansion.

NumAssets

(Optional) Number of assets in MAD portfolio [scalar].

Note: If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is 1.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set the InitPort Property

Given an initial portfolio in `x0`, use the `setInitPort` method to set the `InitPort` property.

```
p = PortfolioMAD('NumAssets', 4);
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = setInitPort(p, x0);
```



```
disp(p.InitPort);  
  
    0.3000  
    0.2000  
    0.2000  
         0
```

Set InitPort to Create an Equally-Weighted Portfolio of Four Assets

Create an equally weighted portfolio of four assets using the `setInitPort` method.

```
p = PortfolioMAD('NumAssets', 4);  
p = setInitPort(p, 1/4, 4);  
disp(p.InitPort);
```

```
    0.2500  
    0.2500  
    0.2500  
    0.2500
```

- “Setting Up an Initial or Current Portfolio” on page 6-37

See Also

`PortfolioMAD` | `setCosts` | `setTurnover`

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setOneWayTurnover

Class: Portfolio

Set up one-way portfolio turnover constraints

Syntax

```
obj = setOneWayTurnover(obj, BuyTurnover)
obj = setOneWayTurnover(obj, BuyTurnover, SellTurnover,
    InitPort, NumAssets)
```

Description

`obj = setOneWayTurnover(obj, BuyTurnover)` sets up one-way portfolio turnover constraints.

`obj = setOneWayTurnover(obj, BuyTurnover, SellTurnover, InitPort, NumAssets)` to set up one-way portfolio turnover constraints with additional options specified by `SellTurnover`, `InitPort`, and `NumAssets`.

Given an initial portfolio in `InitPort` and an upper bound for portfolio turnover on purchases in `BuyTurnover` or sales in `SellTurnover`, the one-way turnover constraints require any portfolio `Port` to satisfy the following:

```
1' * max{0, Port - InitPort} <= BuyTurnover
1' * max{0, InitPort - Port} <= SellTurnover
```

Note: If `Turnover = BuyTurnover = SellTurnover`, the constraint is not equivalent to:

```
1' * | Port - InitPort | <= Turnover
```

To set this constraint, use `setTurnover`.

Tips

You can also use dot notation to set up one-way portfolio turnover constraints.

```
obj = obj.setOneWayTurnover(BuyTurnover, SellTurnover, InitPort, NumAssets)
```

Input Arguments

obj

Portfolio object [Portfolio].

BuyTurnover

Turnover constraint on purchases [scalar].

Note: BuyTurnover must be nonnegative and finite.

SellTurnover

(Optional) Turnover constraint on sales [scalar].

Note: SellTurnover must be nonnegative and finite.

InitPort

(Optional) Initial or current portfolio weights [vector].

Note: InitPort must be a finite vector with NumAssets > 0 elements.

If no InitPort is specified, that value is assumed to be 0.

If InitPort is specified as a scalar and NumAssets exists, then InitPort undergoes scalar expansion.

NumAssets

(Optional) Number of assets in the portfolio [scalar].

Note: If it is impossible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is 1.

Output Arguments

obj

Updated Portfolio object [`Portfolio`].

Definitions

One-Way Turnover Constraint

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the form

$$1^T \max\{0, x - x_0\} \leq \tau_B$$

$$1^T \max\{0, x_0 - x\} \leq \tau_S$$

with

- x — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper-bound for turnover constraint on purchases (scalar)
- τ_S — Upper-bound for turnover constraint on sales (scalar)

To specify one-way turnover constraints, use the following properties in the `Portfolio` object: `BuyTurnover` for τ_B , `SellTurnover` for τ_S , and `InitPort` for x_0 .

Note: The average turnover constraint (see `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Attributes

Access	public
Static	false
Hidden	false

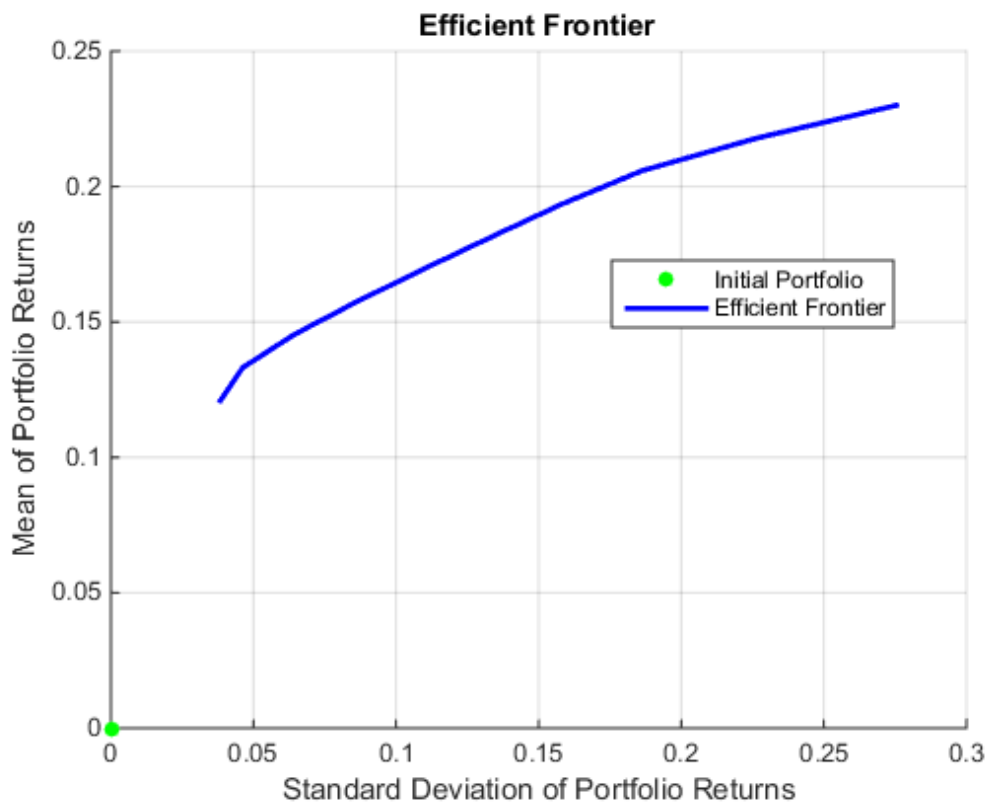
To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set One-Way Turnover Constraints

Set one-way turnover constraints.

```
p = Portfolio('AssetMean',[0.1, 0.2, 0.15], 'AssetCovar',...  
[ 0.005, -0.010, 0.004; -0.010, 0.040, -0.002; 0.004, -0.002, 0.023]);  
p = setBudget(p, 1, 1);  
p = setOneWayTurnover(p, 1.3, 0.3, 0); %130-30 portfolio  
plotFrontier(p);
```



- “Working with One-Way Turnover Constraints” on page 4-75

See Also

`getOneWayTurnover` | `Portfolio` | `setCosts` | `setInitPort` | `setTurnover`

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setOneWayTurnover

Class: PortfolioCVaR

Set up one-way portfolio turnover constraints for PortfolioCVaR object

Syntax

```
obj = setOneWayTurnover(obj, BuyTurnover)
obj = setOneWayTurnover(obj, BuyTurnover, SellTurnover,
  InitPort, NumAssets)
```

Description

`obj = setOneWayTurnover(obj, BuyTurnover)` sets up one-way CVaR portfolio object turnover constraints.

`obj = setOneWayTurnover(obj, BuyTurnover, SellTurnover, InitPort, NumAssets)` sets up one-way portfolio turnover constraints with additional options specified by `SellTurnover`, `InitPort`, and `NumAssets`.

Given an initial portfolio in `InitPort` and an upper bound for portfolio turnover on purchases in `BuyTurnover` or sales in `SellTurnover`, the one-way turnover constraints require any portfolio in `Port` to satisfy the following:

$$1' * \max\{0, \text{Port} - \text{InitPort}\} \leq \text{BuyTurnover}$$

$$1' * \max\{0, \text{InitPort} - \text{Port}\} \leq \text{SellTurnover}$$

Note: If `Turnover = BuyTurnover = SellTurnover`, the constraint is not equivalent to:

$$1' * |\text{Port} - \text{InitPort}| \leq \text{Turnover}$$

To set this constraint, use `setTurnover`.

Tips

You can also use dot notation to set up one-way portfolio turnover constraints.

```
obj = obj.setOneWayTurnover(BuyTurnover, SellTurnover, InitPort, NumAssets)
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

BuyTurnover

Turnover constraint on purchases [scalar].

Note: BuyTurnover must be nonnegative and finite.

SellTurnover

(Optional) Turnover constraint on sales [scalar].

Note: SellTurnover must be nonnegative and finite.

InitPort

(Optional) Initial or current portfolio weights [vector].

Note: InitPort must be a finite vector with NumAssets > 0 elements.

If no InitPort is specified, that value is assumed to be 0.

If InitPort is specified as a scalar and NumAssets exists, then InitPort undergoes scalar expansion.

NumAssets

(Optional) Number of assets in the portfolio [scalar].

Note: If it is impossible to obtain a value for NumAssets, it is assumed that NumAssets is 1.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Definitions

One-Way Turnover Constraint

One-way turnover constraints ensure that estimated optimal CVaR portfolios differ from an initial CVaR portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the form

$$1^T \times \max\{0, x - x_0\} \leq \tau_B$$

$$1^T \times \max\{0, x_0 - x\} \leq \tau_S$$

with

- x — CVaR portfolio (*NumAssets* vector)
- x_0 — Initial CVaR portfolio (*NumAssets* vector)
- τ_B — Upper bound for turnover constraint on purchases (scalar)
- τ_S — Upper bound for turnover constraint on sales (scalar)

To specify one-way turnover constraints, use these properties in the CVaR portfolio object: `BuyTurnover` for τ_B , `SellTurnover` for τ_S , and `InitPort` for x_0 .

Note: The average turnover constraint (see `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Attributes

Access	public
Static	false

Hidden false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set One-Way Turnover Constraints

Set one-way turnover constraints.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];  
p = PortfolioCVaR('InitPort', x0);  
p = setOneWayTurnover(p, 0.3, 0.2);  
disp(p.NumAssets);  
disp(p.BuyTurnover)  
disp(p.SellTurnover)  
disp(p.InitPort);
```

10

0.3000

0.2000

0.1200

0.0900

0.0800

0.0700

0.1000

0.1000

0.1500

0.1100

0.0800

0.1000

- “Working with One-Way Turnover Constraints” on page 5-74

See Also

[getOneWayTurnover](#) | [PortfolioCVaR](#) | [setInitPort](#) | [setTurnover](#)

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setOneWayTurnover

Class: PortfolioMAD

Set up one-way portfolio turnover constraints for PortfolioMAD object

Syntax

```
obj = setOneWayTurnover(obj, BuyTurnover)
obj = setOneWayTurnover(obj, BuyTurnover, SellTurnover,
    InitPort, NumAssets)
```

Description

`obj = setOneWayTurnover(obj, BuyTurnover)` sets up one-way PortfolioMAD object turnover constraints.

`obj = setOneWayTurnover(obj, BuyTurnover, SellTurnover, InitPort, NumAssets)` sets up one-way portfolio turnover constraints with additional options specified by `SellTurnover`, `InitPort`, and `NumAssets`.

Given an initial portfolio in `InitPort` and an upper bound for portfolio turnover on purchases in `BuyTurnover` or sales in `SellTurnover`, the one-way turnover constraints require any portfolio in `Port` to satisfy the following:

```
1' * max{0, Port - InitPort} <= BuyTurnover
1' * max{0, InitPort - Port} <= SellTurnover
```

Note: If `Turnover = BuyTurnover = SellTurnover`, the constraint is not equivalent to:

```
1' * | Port - InitPort | <= Turnover
```

To set this constraint, use `setTurnover`.

Tips

You can also use dot notation to set up one-way portfolio turnover constraints.

```
obj = obj.setOneWayTurnover(BuyTurnover, SellTurnover, InitPort, NumAssets)
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

BuyTurnover

Turnover constraint on purchases [scalar].

Note: BuyTurnover must be nonnegative and finite.

SellTurnover

(Optional) Turnover constraint on sales [scalar].

Note: SellTurnover must be nonnegative and finite.

InitPort

(Optional) Initial or current portfolio weights [vector].

Note: InitPort must be a finite vector with NumAssets > 0 elements.

If no InitPort is specified, that value is assumed to be 0.

If InitPort is specified as a scalar and NumAssets exists, then InitPort undergoes scalar expansion.

NumAssets

(Optional) Number of assets in the portfolio [scalar].

Note: If it is impossible to obtain a value for NumAssets, it is assumed that NumAssets is 1.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Definitions

One-Way Turnover Constraint

One-way turnover constraints ensure that estimated optimal MAD portfolios differ from an initial MAD portfolio by no more than specified amounts according to whether the differences are purchases or sales. The constraints take the form

$$1^T \times \max\{0, x - x_0\} \leq \tau_B$$

$$1^T \times \max\{0, x_0 - x\} \leq \tau_S$$

with

- x — MAD portfolio (*NumAssets* vector)
- x_0 — Initial MAD portfolio (*NumAssets* vector)
- τ_B — Upper bound for turnover constraint on purchases (scalar)
- τ_S — Upper bound for turnover constraint on sales (scalar)

To specify one-way turnover constraints, use these properties in the PortfolioMAD object: `BuyTurnover` for τ_B , `SellTurnover` for τ_S , and `InitPort` for x_0 .

Note: The average turnover constraint (see `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Attributes

Access	public
Static	false

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

setOptions

Class: Portfolio

Set hidden properties in Portfolio object

Syntax

```
obj = setOptions(obj)
obj = setOptions(obj,varargin)
```

Description

`obj = setOptions(obj)` to set the hidden properties in a Portfolio object.

`obj = setOptions(obj,varargin)` to set the hidden properties in a Portfolio object with additional options specified by one or more Name,Value pair arguments.

Warning This method is currently nonfunctional and issues the following warning message:

Warning: The method setOptions, which will enable modification of hidden properties, is not supported yet

The only way to modify hidden properties in a Portfolio object is by direct assignment. An exception is the method `setSolver`, which permits modification of hidden properties associated with the solvers used by the Portfolio object. In addition, the only way to get or display hidden properties is by direct access.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

See Also

`setSolver`

setOptions

Class: PortfolioCVaR

Set hidden properties in PortfolioCVaR object

Syntax

```
obj = setOptions(obj)
obj = setOptions(obj,varargin)
```

Description

`obj = setOptions(obj)` to set the hidden properties in a PortfolioCVaR object.

`obj = setOptions(obj,varargin)` to set the hidden properties in a PortfolioCVaR object with additional options specified by one or more Name,Value pair arguments.

Warning This method is currently nonfunctional and issues the following warning message:

```
Warning: The method setOptions, which will enable modification of hidden properties,
is not supported yet
```

The only way to modify hidden properties in a PortfolioCVaR object is by direct assignment. An exception is the method `setSolver`, which permits modification of hidden properties associated with the solvers used by the PortfolioCVaR object. In addition, the only way to get or display hidden properties is by direct access.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

See Also

`setSolver`

setOptions

Class: PortfolioMAD

Set hidden properties in PortfolioMAD object

Syntax

```
obj = setOptions(obj)
obj = setOptions(obj,varargin)
```

Description

`obj = setOptions(obj)` to set the hidden properties in a PortfolioMAD object.

`obj = setOptions(obj,varargin)` to set the hidden properties in a PortfolioMAD object with additional options specified by one or more Name,Value pair arguments.

Warning This method is currently nonfunctional and issues the following warning message:

```
Warning: The method setOptions, which will enable modification of hidden properties,
is not supported yet
```

The only way to modify hidden properties in a PortfolioMAD object is by direct assignment. An exception is the method `setSolver`, which permits modification of hidden properties associated with the solvers used by the PortfolioMAD object. In addition, the only way to get or display hidden properties is by direct access.

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

See Also

`setSolver`

setProbabilityLevel

Class: PortfolioCVaR

Set probability level for VaR and CVaR calculations

Syntax

```
obj = setProbabilityLevel(obj,ProbabilityLevel)
```

Description

`obj = setProbabilityLevel(obj,ProbabilityLevel)` to set the probability level for VaR and CVaR calculations.

Tips

You can also use dot notation to set the probability level for VaR and CVaR calculations:

```
obj = obj.setProbabilityLevel(ProbabilityLevel)
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

ProbabilityLevel

Probability level which is 1 minus the probability of losses greater than the value-at-risk. [scalar].

Note: `ProbabilityLevel` must be a value from 0 to 1 and, in most cases, should be a value from 0.9 to 0.99.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Probability Level

Set the `ProbabilityLevel` for a CVaR portfolio object.

```
p = PortfolioCVaR;  
p = setProbabilityLevel(p, 0.95);  
disp(p.ProbabilityLevel)
```

```
0.9500
```

- “What Are Scenarios?” on page 5-42

See Also

PortfolioCVaR

More About

- “Conditional Value-at-Risk” on page 5-5
- “Default Portfolio Problem” on page 5-14

setSolver

Class: Portfolio

Choose main solver and specify associated solver options for portfolio optimization

Syntax

```
obj = setSolver(obj,solverType)
obj = setSolver(obj,solverType,varargin)
```

Description

`obj = setSolver(obj,solverType)` chooses the solver for portfolio optimization.

`obj = setSolver(obj,solverType,varargin)` chooses the solver and specifies associated solver options with additional options specified by one or more Name,Value pair arguments or an `optimoptions` object.

After you specify a solver, the `varargin` argument accepts either name-value pair arguments to set options or, for the case of solvers from Optimization Toolbox software, a structure created by `optimoptions`.

Tips

You can also use dot notation to choose the solver and specify associated solver options.

```
obj = obj.setSolver(solverType, varargin);
```

Input Arguments

obj

Portfolio object [Portfolio].

solverType

Solver to use for portfolio optimization [string]. The default solver for the Portfolio object is 'lcprog' with the control variables 'maxiter', 'tiebreak', 'tolpiv'.

Default: 'lcprog'

Name-Value Pair Arguments or optimoptions Object

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'varargin'

Options to control the solver specified in solverType as name-value pair arguments or an optimoptions object. Note, optimoptions is the default and recommended method to set solver options, however optimset is also supported.

Note: The default solver for the Portfolio object is 'lcprog'. The Portfolio object can also use 'quadprog', which has several different options that can be set with optimoptions. Unlike Optimization Toolbox software which uses the trust-region-reflective algorithm as the default algorithm for quadprog, the portfolio optimization tools use the interior-point-convex algorithm. For more information about quadprog and quadratic programming algorithms and options, see “Quadratic Programming Algorithms”.

Output Arguments**obj**

Updated Portfolio object [Portfolio].

Attributes

Access

public

Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Solver Type

If you use the `quadprog` function as the `solverType`, the default is the interior-point-convex version of `quadprog`.

```
load CAPMuniverse
p = Portfolio('AssetList',Assets(1:12));
p = setDefaultConstraints(p);
p = setSolver(p, 'quadprog');
display(p.solverType);
```

```
quadprog
```

You can switch back to `lcprog` with:

```
p = setSolver(p, 'lcprog');
display(p.solverType);
```

```
lcprog
```

- “Choosing and Controlling the Solver” on page 4-94

See Also

`optimoptions` | `quadprog` | `setOptions`

setSolver

Class: PortfolioCVaR

Choose main solver and solver options for CVaR portfolio optimization

Syntax

```
obj = setSolver(obj,solverType)
obj = setSolver(obj,solverType,varargin)
```

Description

`obj = setSolver(obj,solverType)` chooses the solver for CVaR portfolio optimization.

`obj = setSolver(obj,solverType,varargin)` chooses the solver and specifies associated solver options with additional options specified by one or more Name,Value pair arguments or an `optimoptions` object.

After you specify a solver, the `varargin` argument accepts either name-value pair arguments to set options or, for the case of solvers from Optimization Toolbox software, an object created by `optimoptions`.

Tips

You can also use dot notation to choose the solver and specify associated solver options:

```
obj = obj.setSolver(solverType, varargin);
```

Input Arguments

obj

CVaR portfolio object [`PortfolioCVaR`].

solverType

Solver to use for CVaR portfolio optimization [string]. The supported `solverType` are:

- 'fmincon'
- 'cuttingplane'

Use `optimoptions` to specify further options for CVaR portfolio optimization.

Default: 'fmincon' using the 'sqp' algorithm

Name-Value Pair Arguments or optimoptions Object

Specify optional comma-separated pairs of Name,Value arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'varargin'

Options to control the solver specified in `solverType` as name-value pair arguments or an `optimoptions` object. `optimoptions` is the default and recommended method to set solver options, however `optimset` is also supported.

Note:

- For the default solver, 'fmincon', `PortfolioCVaR` by default sets the algorithm to 'sqp', uses objective function gradients, and turns the display off. All `fmincon` options are supported, either as name-value pair arguments or using an `optimoptions` object. For more information about `fmincon` and quadratic programming algorithms and options, see “Quadratic Programming Algorithms”.
 - For the `cuttingplane` solver, the following solver options are available as name-value pair arguments:
 - 'MaxIter'
 - 'AbsTol'
 - 'RelTol'
 - 'MasterSolverOptions'
-

'MaxIter '

Positive integer number for the maximum number of iterations.

Default: 1000

'AbsTol '

Positive scalar value for the absolute stopping tolerance.

Default: 1e-6

'RelTol '

Positive scalar for the relative stopping tolerance.

Default: 1e-5

'MasterSolverOptions'

optimoptions object for the options for the master solver `linprog`. For more information about `linprog` and linear programming algorithms and options, see “Linear Programming Algorithms”.

Default: `optimoptions('linprog','Algorithm','Dual-Simplex','Display','off')`

Output Arguments

obj

Updated CVaR portfolio object [`PortfolioCVaR`].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set the Solver Type as 'fmincon'

Use 'fmincon' as the solverType.

```
p = PortfolioCvAR;
p = setSolver(p, 'fmincon');
display(p.solverType);
```

```
fmincon
```

Set the Solver Type as 'fmincon' and Use Name-Value Pair Arguments to Set the Algorithm

Use 'fmincon' as the solverType and use name-value pair arguments to set the algorithm to 'trust-region-reflective' and to turn off the display.

```
p = PortfolioCvAR;
p = setSolver(p, 'fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

Set the Solver Type as 'fmincon' and Use an optimoptions Object to Set the Algorithm

Use 'fmincon' as the solverType and use an optimoptions object to set the algorithm to 'trust-region-reflective' and to turn off the display.

```
p = PortfolioCvAR;
options = optimoptions('fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

Set 'cuttingplane' as the Solver Type with Default Options

Use 'cuttingplane' as the solverType with default options.

```
p = PortfolioCVaR;  
p = setSolver(p, 'cuttingplane');  
display(p.solverType);
```

```
cuttingplane
```

Set 'cuttingplane' as the Solver Type with Maximum Iterations

Use the Name-Value pair 'MaxIter' to set the maximum number of iterations to 1500.

```
p = PortfolioCVaR;  
p = setSolver(p, 'cuttingplane', 'MaxIter', 1500);  
display(p.solverType);  
display(p.solverOptions);
```

```
cuttingplane
```

```
      MaxIter: 1500  
      AbsTol: 1.0000e-06  
      RelTol: 1.0000e-05  
MasterSolverOptions: [1x1 optim.options.Linprog]
```

Set 'cuttingplane' as the Solver Type and Change the Master Solver Option

For the master solver, continue using the dual-simplex algorithm with no display, but tighten its termination tolerance to $1e-8$.

```
p = PortfolioCVaR;  
options = optimoptions('linprog', 'Algorithm', 'Dual-Simplex', 'Display', 'off', 'TolFun',  
1e-8);  
p = setSolver(p, 'cuttingplane', 'MasterSolverOptions', options);  
display(p.solverType)  
display(p.solverOptions)  
display(p.solverOptions.MasterSolverOptions.Algorithm)  
display(p.solverOptions.MasterSolverOptions.Display)  
display(p.solverOptions.MasterSolverOptions.TolFun)
```

```
cuttingplane
```

```
      MaxIter: 1000  
      AbsTol: 1.0000e-06  
      RelTol: 1.0000e-05  
MasterSolverOptions: [1x1 optim.options.Linprog]
```

```
dual-simplex  
off
```



```
ans =
    1.0000e-08
```

For the master solver, use the interior-point algorithm with no display, and with a termination tolerance of $1e-7$.

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','interior-point','Display','off','TolFun',
p = setSolver(p,'cuttingplane','MasterSolverOptions',options);
display(p.solverType)
display(p.solverOptions)
display(p.solverOptions.MasterSolverOptions.Algorithm)
display(p.solverOptions.MasterSolverOptions.Display)
display(p.solverOptions.MasterSolverOptions.TolFun)

cuttingplane
           MaxIter: 1000
           AbsTol: 1.0000e-06
           RelTol: 1.0000e-05
MasterSolverOptions: [1x1 optim.options.Linprog]

interior-point
off

ans =
    1.0000e-07
```

- “Choosing and Controlling the Solver” on page 5-92
- “Asset Returns and Scenarios” on page 5-41

Algorithms

One version of the CVaR portfolio optimization problem minimizes risk, measured as the CVaR of the portfolio, subject to a target return and other linear constraints on the portfolio. For the definition of the CVaR function, see “Risk Proxy” on page 5-5.

$$\text{minimize}_x \text{CVaR}_\alpha(x)$$

subject to $\bar{y}^T x \geq \text{TargetReturn}$

$$Ax \leq b$$

$$A_{eq}x = b_{eq}$$

$$lb \leq x \leq ub$$

Vector \bar{y} is the mean return vector (the column-wise mean of the scenario matrix Y), so that $\bar{y}^T x$ is the expected return of portfolio x . The first constraint says that the expected return must be at least as good as a target return.

An alternative version of the CVaR portfolio optimization problem maximizes the expected return of the portfolio, subject to a target risk and other linear constraints on the portfolio.

$$\text{maximize}_x \bar{y}^T x$$

subject to $CVaR_\alpha(x) \leq CVaRLimit$

$$Ax \leq b$$

$$A_{eq}x = b_{eq}$$

$$lb \leq x \leq ub$$

The first constraint in this case says that the portfolio CVaR cannot exceed a given CVaR limit.

By default, the CVaR portfolio object uses `fmincon` to solve the CVaR portfolio optimization problems. For information about `fmincon` and quadratic programming algorithms and options, see “Quadratic Programming Algorithms”.

Alternatively, the CVaR portfolio optimization problems can be solved with `'cuttingplane'`, an implementation of Kelley’s cutting-plane method. For more information, see Kelley [45] at “Portfolio Optimization” on page A-10.

References

Kelley, J. E., "The Cutting-Plane Method for Solving Convex Programs," *Journal of the Society for Industrial and Applied Mathematics*, Vol. 8, No. 4, December 1960, pp. 703-712.

Rockafellar, R. T. and S. Uryasev, "Optimization of Conditional Value-at-Risk," *Journal of Risk*, Vol. 2, No. 3, Spring 2000, pp. 21–41.

Rockafellar, R. T. and S. Uryasev, "Conditional Value-at-Risk for General Loss Distributions," *Journal of Banking and Finance*, Vol. 26, 2002, pp. 1443–1471.

See Also

fmincon | linprog | optimoptions | PortfolioCVaR | setOptions

setSolver

Class: PortfolioMAD

Choose main solver and solver options for MAD portfolio optimization

Syntax

```
obj = setSolver(obj,solverType)
obj = setSolver(obj,solverType,varargin)
```

Description

`obj = setSolver(obj,solverType)` chooses the solver for MAD portfolio optimization.

`obj = setSolver(obj,solverType,varargin)` chooses the solver and specifies associated solver options with additional options specified by one or more Name,Value pair arguments or an `optimoptions` object.

After you specify a solver, the `varargin` argument accepts either name-value pair arguments to set options or, for the case of solvers from Optimization Toolbox software, an object created by `optimoptions`.

Tips

You can also use dot notation to choose the solver and specify associated solver options.

```
obj = obj.setSolver(solverType, varargin);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

solverType

Solver to use for MAD portfolio optimization [string]. The supported `solverType` is:

- 'fmincon'

Use `optimoptions` to specify further options for MAD portfolio optimization.

Default: 'fmincon' using the 'sqp' algorithm and 'GradObj' set to 'on'

Name-Value Pair Arguments or optimoptions Object

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'varargin'

Options to control the solver specified in `solverType` as name-value pair arguments or an `optimoptions` object.

Note:

- For the default solver 'fmincon', PortfolioMAD, by default, sets the algorithm to 'sqp' and turns the display off. While all variations of fmincon from Optimization Toolbox are accepted, use of 'sqp' and 'active-set' algorithms for fmincon are recommended and the use of 'interior-point' and 'trust-region-reflective' algorithms are not recommended for MAD portfolio optimization.
 - All fmincon options are supported either as name-value pair arguments or using an `optimoptions` object.
 - `optimoptions` is the default and recommended method to set solver options, however `optimset` is also supported. For details about fmincon and constrained nonlinear optimization algorithms and options, see “Constrained Nonlinear Optimization Algorithms”.
-

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Solver Type as 'fmincon'

Use 'fmincon' as the solverType.

```
p = PortfolioMAD;  
p = setSolver(p, 'fmincon');  
display(p.solverType);
```

```
fmincon
```

Set the Solver Type as 'fmincon' and Use Name-Value Pair Arguments to Set the Algorithm

Use 'fmincon' as the solverType and use name-value pair arguments to set the algorithm to 'sqp' and to turn on the display.

```
p = PortfolioMAD;  
p = setSolver(p, 'fmincon', 'Algorithm', 'sqp', 'Display', 'final');  
display(p.solverOptions.Algorithm);  
display(p.solverOptions.Display);
```

```
sqp
```

```
final
```

Set Solver Type as 'fmincon' and Use an optimoptions Structure to Set the Algorithm

Use 'fmincon' as the solverType and use an optimoptions object to set the algorithm to 'trust-region-reflective' and to turn off the display.

```
p = PortfolioMAD;
options = optimoptions('fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

Set the Solver Type as 'fmincon' and Use an optimoptions Structure to Set the Algorithm and Use of Gradients

Use 'fmincon' as the solverType and use an optimoptions object to set the algorithm to 'active-set' and to set the gradients flag 'on' for 'GradObj' and turn off the display.

```
p = PortfolioMAD;
options = optimoptions('fmincon', 'algorithm', 'active-set', 'display', 'off', 'gradobj', 'on');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
active-set
off
```

- “Choosing and Controlling the Solver” on page 6-92
- “Asset Returns and Scenarios” on page 6-41

References

Rockafellar, R. T. and S. Uryasev, "Optimization of Conditional Value-at-Risk," *Journal of Risk*, Vol. 2, No. 3, Spring 2000, pp. 21–41.

Rockafellar, R. T. and S. Uryasev, "Conditional Value-at-Risk for General Loss Distributions," *Journal of Banking and Finance*, Vol. 26, 2002, pp. 1443–1471.

See Also

fmincon | optimoptions | PortfolioMAD | setOptions

setScenarios

Class: PortfolioCVaR

Set asset returns scenarios by direct matrix for PortfolioCVaR object

Syntax

```
obj = setScenarios(obj,AssetScenarios)
obj = setScenarios(obj,AssetScenarios,varargin)
```

Description

`obj = setScenarios(obj,AssetScenarios)` sets asset returns scenarios by direct matrix.

`obj = setScenarios(obj,AssetScenarios,varargin)` sets asset returns scenarios by direct matrix using the optional input `varargin` with additional options specified by one or more Name,Value pair arguments.

Tips

You can also use dot notation to set asset return scenarios.

```
obj = obj.setScenarios(AssetScenarios, varargin);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

AssetScenarios

Matrix of scenarios for asset returns or prices [matrix]. If the input data are prices, they can be converted into returns with the `DataFormat Name/Value` argument, where the

default format is assumed to be returns. Be careful using price data because portfolio optimization usually requires total returns and not simply price returns.

This method sets up a function handle to indirectly access input `AssetScenarios` without needing to make a copy of the data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

DataFormat

`DataFormat` values are:

- `Returns` — Data in `AssetScenarios` are asset total returns.
- `Prices` — Data in `AssetScenarios` are asset total return prices.

Default: Returns

GetAssetList

`GetAssetList` values are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this method and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is currently `Asset`.

If the `GetAssetList` flag is `false`, nothing happens, which is the default.

Default: false

Output Arguments

obj

Updated CVaR portfolio object [`PortfolioCVaR`].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Asset Returns Scenarios

Given a `PortfolioCVaR` object `p`, use the `setScenarios` method to set asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);
disp(p)
```

PortfolioCVaR with properties:

```
        BuyCost: []
        SellCost: []
        RiskFreeRate: []
ProbabilityLevel: 0.9500
        Turnover: []
        BuyTurnover: []
        SellTurnover: []
        NumScenarios: 20000
            Name: []
        NumAssets: 4
        AssetList: []
        InitPort: []
        AInequality: []
        bInequality: []
            AEquality: []
            bEquality: []
        LowerBound: [4x1 double]
        UpperBound: []
        LowerBudget: 1
        UpperBudget: 1
        GroupMatrix: []
            LowerGroup: []
            UpperGroup: []
                GroupA: []
                GroupB: []
        LowerRatio: []
        UpperRatio: []
```

- “Setting Scenarios Using the PortfolioCVaR Constructor” on page 5-42

See Also

PortfolioCVaR

More About

- “How Stochastic Optimization Works” on page 5-41

setScenarios

Class: PortfolioMAD

Set asset returns scenarios by direct matrix for PortfolioMAD object

Syntax

```
obj = setScenarios(obj,AssetScenarios)
obj = setScenarios(obj,AssetScenarios,varargin)
```

Description

`obj = setScenarios(obj,AssetScenarios)` sets asset returns scenarios by direct matrix.

`obj = setScenarios(obj,AssetScenarios,varargin)` sets asset returns scenarios by direct matrix using the optional input `varargin` with additional options specified by one or more Name,Value pair arguments.

Tips

You can also use dot notation to set asset return scenarios.

```
obj = obj.setScenarios(AssetScenarios, varargin);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

AssetScenarios

Matrix of scenarios for asset returns or prices [matrix]. If the input data are prices, they can be converted into returns with the `DataFormat Name/Value` argument, where the

default format is assumed to be returns. Be careful using price data because portfolio optimization usually requires total returns and not simply price returns.

This method sets up a function handle to indirectly access input `AssetScenarios` without needing to make a copy of the data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

DataFormat

`DataFormat` values are:

- `Returns` — Data in `AssetScenarios` are asset total returns.
- `Prices` — Data in `AssetScenarios` are asset total return prices.

Default: Returns

GetAssetList

`GetAssetList` values are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this method and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is currently `Asset`.

If the `GetAssetList` flag is `false`, nothing happens, which is the default.

Default: false

Output Arguments

obj

Updated MAD portfolio object [**PortfolioMAD**].

Attributes

Access	public
Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Asset Returns Scenarios

Given **PortfolioMAD** object **p**, use the **setScenarios** method to set asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
disp(p)
```

PortfolioMAD with properties:

```
    BuyCost: []
    SellCost: []
RiskFreeRate: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
NumScenarios: 20000
    Name: []
    NumAssets: 4
    AssetList: []
    InitPort: []
AInequality: []
bInequality: []
    AEquality: []
    bEquality: []
LowerBound: [4x1 double]
UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
LowerGroup: []
UpperGroup: []
    GroupA: []
    GroupB: []
LowerRatio: []
UpperRatio: []
```

- “Setting Scenarios Using the PortfolioMAD Constructor” on page 6-42
- “Setting Scenarios Using the setScenarios Method” on page 6-44

See Also

PortfolioMAD

More About

- “How Stochastic Optimization Works” on page 6-41

setTurnover

Class: Portfolio

Set up maximum portfolio turnover constraint

Syntax

```
obj = setTurnover(obj, Turnover)
obj = setTurnover(obj, Turnover, InitPort, NumAssets)
```

Description

`obj = setTurnover(obj, Turnover)` sets up the maximum portfolio turnover constraint.

`obj = setTurnover(obj, Turnover, InitPort, NumAssets)` sets up the maximum portfolio turnover constraint with additional options specified by `InitPort` and `NumAssets`.

Given an upper bound for portfolio turnover in `Turnover` and an initial portfolio in `InitPort`, the turnover constraint requires any portfolio in `Port` to satisfy the following:

$$1' * | \text{Port} - \text{InitPort} | \leq \text{Turnover}$$

Tips

You can also use dot notation to set up the maximum portfolio turnover constraint.

```
obj = obj.setTurnover(Turnover, InitPort, NumAssets);
```

Input Arguments

obj

Portfolio object [Portfolio].

Turnover

Portfolio turnover constraint [scalar].

Note: Turnover must be nonnegative and finite.

InitPort

(Optional) Initial or current portfolio weights [vector].

Note: InitPort must be a finite vector with NumAssets > 0 elements.

If no InitPort is specified, that value is assumed to be 0.

If InitPort is specified as a scalar and NumAssets exists, then InitPort undergoes scalar expansion.

NumAssets

(Optional) Number of assets in the portfolio [scalar].

Note: If it is not possible to obtain a value for NumAssets, it is assumed that NumAssets is 1.

Output Arguments

obj

Updated Portfolio object [Portfolio].

Attributes

Access	public
Static	false

More About

- “Portfolio Set for Portfolio Optimization” on page 4-7

setTurnover

Class: PortfolioCVaR

Set up maximum turnover constraint in PortfolioCVaR object

Syntax

```
obj = setTurnover(obj,Turnover)
obj = setTurnover(obj,Turnover,InitPort,NumAssets)
```

Description

`obj = setTurnover(obj,Turnover)` sets up the maximum turnover constraint for a portfolio.

`obj = setTurnover(obj,Turnover,InitPort,NumAssets)` sets up the maximum turnover constraint for a portfolio with additional options specified by `InitPort` and `NumAssets`.

Given an upper bound for portfolio turnover in `Turnover` and an initial portfolio in `InitPort`, the turnover constraint requires any portfolio in `Port` to satisfy the following:

```
1' * | Port - InitPort | <= Turnover
```

Tips

You can also use dot notation to set up the maximum turnover constraint for a portfolio.

```
obj = obj.setTurnover(Turnover, InitPort, NumAssets);
```

Input Arguments

obj

CVaR portfolio object [`PortfolioCVaR`].

Turnover

Portfolio turnover constraint [scalar].

Note: Turnover must be nonnegative and finite.

InitPort

(Optional) Initial or current portfolio weights [vector].

Note: InitPort must be a finite vector with NumAssets > 0 elements.

If no InitPort is specified, that value is assumed to be 0.

If InitPort is specified as a scalar and NumAssets exists, then InitPort undergoes scalar expansion.

NumAssets

(Optional) Number of assets in the portfolio [scalar].

Note: If it is not possible to obtain a value for NumAssets, it is assumed that NumAssets is 1.

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false

Hidden false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Turnover Constraint for a CVaR Portfolio Object

Given a CVaR portfolio object `p`, to ensure that average turnover is no more than 30% with an initial portfolio of 10 assets in a variable `x0`, use the `setTurnover` method to set the turnover constraint.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];  
p = PortfolioCVaR('InitPort', x0);  
p = setTurnover(p, 0.3);
```

```
disp(p.NumAssets);  
disp(p.Turnover);  
disp(p.InitPort);
```

```
10  
  
0.3000  
  
0.1200  
0.0900  
0.0800  
0.0700  
0.1000  
0.1000  
0.1500  
0.1100  
0.0800  
0.1000
```

- “Working with Average Turnover Constraints” on page 5-72

See Also

PortfolioCVaR | setInitPort | setOneWayTurnover

More About

- “Portfolio Set for Portfolio Optimization” on page 5-7

setTurnover

Class: PortfolioMAD

Set up maximum turnover constraint in PortfolioMAD object

Syntax

```
obj = setTurnover(obj, Turnover)
obj = setTurnover(obj, Turnover, InitPort, NumAssets)
```

Description

`obj = setTurnover(obj, Turnover)` sets up the maximum turnover constraint for a portfolio.

`obj = setTurnover(obj, Turnover, InitPort, NumAssets)` sets up the maximum turnover constraint for a portfolio with additional options specified by `InitPort` and `NumAssets`.

Given an upper bound for portfolio turnover in `Turnover` and an initial portfolio in `InitPort`, the turnover constraint requires any MAD portfolio in `Port` to satisfy the following:

$$1' * | Port - InitPort | \leq Turnover$$

Tips

You can also use dot notation to set up the maximum turnover constraint for a portfolio.

```
obj = obj.setTurnover(Turnover, InitPort, NumAssets);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

Turnover

Portfolio turnover constraint [scalar].

Note: Turnover must be nonnegative and finite.

InitPort

(Optional) Initial or current portfolio weights [vector].

Note: InitPort must be a finite vector with NumAssets > 0 elements.

If no InitPort is specified, that value is assumed to be 0.

If InitPort is specified as a scalar and NumAssets exists, then InitPort undergoes scalar expansion.

NumAssets

(Optional) Number of assets in the portfolio [scalar].

Note: If it is not possible to obtain a value for NumAssets, it is assumed that NumAssets is 1.

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access	public
Static	false

Hidden false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Set Turnover Constraint for a MAD Portfolio Object

Given PortfolioMAD object `p`, to ensure that average turnover is no more than 30% with an initial portfolio of 10 assets in a variable `x0`, use the `setTurnover` method to set the turnover constraint.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];  
p = PortfolioMAD('InitPort', x0);  
p = setTurnover(p, 0.3);
```

```
disp(p.NumAssets);  
disp(p.Turnover);  
disp(p.InitPort);
```

```
10  
  
0.3000  
  
0.1200  
0.0900  
0.0800  
0.0700  
0.1000  
0.1000  
0.1500  
0.1100  
0.0800  
0.1000
```

- “Working with Average Turnover Constraints” on page 6-72

See Also

PortfolioMAD | setInitPort | setOneWayTurnover

More About

- “Portfolio Set for Portfolio Optimization” on page 6-7

simulateNormalScenariosByData

Class: PortfolioCVaR

Simulate multivariate normal asset return scenarios from data for PortfolioCVaR object

Syntax

```
obj = simulateNormalScenariosByData(obj,AssetReturns,  
NumScenarios)
```

```
obj = simulateNormalScenariosByData(obj,AssetReturns,  
NumScenarios,varargin)
```

Description

`obj = simulateNormalScenariosByData(obj,AssetReturns, NumScenarios)` to simulate multivariate normal asset return scenarios for a CVaR portfolio object from data.

`obj = simulateNormalScenariosByData(obj,AssetReturns, NumScenarios,varargin)` to simulate multivariate normal asset return scenarios for a CVaR portfolio object from data using the optional input `varargin` with additional options specified by one or more Name,Value pair arguments.

This method estimates the mean and covariance of asset returns from either price or return data and then uses these estimates to generate the specified number of scenarios with the function `mvnrnd`.

Data can in be either a `NumSamples`-by-`NumAssets` matrix of `NumSamples` prices or returns at a given periodicity for a collection of `NumAssets` assets or a `fints` object with `NumSamples` observations and `NumAssets` time series.

Note: If you want to use the method multiple times and you want to simulate identical scenarios each time the function is called, precede each function call with `rng(seed)` using a specified integer seed.

Tips

You can also use dot notation to simulate multivariate normal asset return scenarios from data for a CVaR portfolio object.

```
obj = obj.simulateNormalScenariosByData(AssetReturns, NumScenarios, varargin);
```

Input Arguments

obj

A CVaR portfolio object [PortfolioCVaR].

AssetReturns

Either a matrix or `fints` object that contains asset data that can be converted into asset returns [NumSamples-by-NumAssets] matrix.

NumScenarios

Number of scenarios to simulate [positive integer].

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

DataFormat

`DataFormat` values are:

- `Returns` — Data in `AssetReturns` are asset total returns.
- `Prices` — Data in `AssetReturns` are asset total return prices.

Default: Returns

MissingData

`MissingData` values are:

- `false` — Do not use ECM algorithm to handle NaN values (just exclude NaN values).
- `true` — Use ECM algorithm to handle NaN values.

Default: `false`

GetAssetList

GetAssetList values are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this method and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is currently `Asset`.

If the `GetAssetList` flag is `false`, nothing happens, which is the default.

Default: `false`

Output Arguments

obj

Updated CVaR portfolio object [`PortfolioCVaR`].

Attributes

Access	<code>public</code>
Static	<code>false</code>
Hidden	<code>false</code>

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Simulate Multivariate Normal Asset Return Scenarios from Data

Given a PortfolioCVaR object `p`, use the `simulateNormalScenariosByData` method to simulate multivariate normal asset return scenarios from data.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

RawData = mvnrnd(m, C, 240);
NumScenarios = 2000;

p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, RawData, NumScenarios)
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);
```

```
disp(p);
```

```
p =
```

```
PortfolioCVaR with properties:
```

```
    BuyCost: []
    SellCost: []
    RiskFreeRate: []
    ProbabilityLevel: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    NumScenarios: 2000
    Name: []
    NumAssets: 4
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
```



```
AEquality: []  
bEquality: []  
LowerBound: []  
UpperBound: []  
LowerBudget: []  
UpperBudget: []  
GroupMatrix: []  
LowerGroup: []  
UpperGroup: []  
GroupA: []  
GroupB: []  
LowerRatio: []  
UpperRatio: []
```

PortfolioCVaR with properties:

```
BuyCost: []  
SellCost: []  
RiskFreeRate: []  
ProbabilityLevel: 0.9000  
Turnover: []  
BuyTurnover: []  
SellTurnover: []  
NumScenarios: 2000  
Name: []  
NumAssets: 4  
AssetList: []  
InitPort: []  
AInequality: []  
bInequality: []  
AEquality: []  
bEquality: []  
LowerBound: [4x1 double]  
UpperBound: []  
LowerBudget: 1  
UpperBudget: 1  
GroupMatrix: []  
LowerGroup: []  
UpperGroup: []  
GroupA: []  
GroupB: []  
LowerRatio: []  
UpperRatio: []
```

Simulate Multivariate Normal Asset Return Scenarios from Market Data

Create a PortfolioCVaR object `p` and use the `simulateNormalScenariosByData` method with market data loaded from `CAPMuniverse.mat` to simulate multivariate normal asset return scenarios.

```
load CAPMuniverse
```

```
p = PortfolioCVaR('AssetList',Assets(1:12));  
disp(p);
```

PortfolioCVaR with properties:

```
    BuyCost: []  
    SellCost: []  
    RiskFreeRate: []  
    ProbabilityLevel: []  
    Turnover: []  
    BuyTurnover: []  
    SellTurnover: []  
    NumScenarios: []  
    Name: []  
    NumAssets: 12  
    AssetList: {1x12 cell}  
    InitPort: []  
    AInequality: []  
    bInequality: []  
    AEquality: []  
    bEquality: []  
    LowerBound: []  
    UpperBound: []  
    LowerBudget: []  
    UpperBudget: []  
    GroupMatrix: []  
    LowerGroup: []  
    UpperGroup: []  
    GroupA: []  
    GroupB: []  
    LowerRatio: []  
    UpperRatio: []
```

Simulate the scenarios from the data for each of the 12 assets from `CAPMuniverse.mat`.

```
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 , 'missingdata', true);
```

- “Simulating Normal Scenarios” on page 5-45

See Also

PortfolioCVAR | rng | simulateNormalScenariosByMoments

More About

- “How Stochastic Optimization Works” on page 5-41

simulateNormalScenariosByData

Class: PortfolioMAD

Estimate mean and covariance of asset returns from data for PortfolioMAD object

Syntax

```
obj = simulateNormalScenariosByData(obj,AssetReturns,...  
NumScenarios)  
obj = simulateNormalScenariosByData(obj,AssetReturns,...  
NumScenarios,varargin)
```

Description

`obj = simulateNormalScenariosByData(obj,AssetReturns,...
NumScenarios)` to estimate mean and covariance of asset returns from data for a PortfolioMAD object.

`obj = simulateNormalScenariosByData(obj,AssetReturns,...
NumScenarios,varargin)` to estimate mean and covariance of asset returns from data for a PortfolioMAD object using the optional input `varargin` with additional options specified by one or more Name,Value pair arguments.

This method estimates the mean and covariance of asset returns from either price or return data and then uses these estimates to generate the specified number of scenarios with the function `mvnrnd`.

Data can in be either a `NumSamples`-by-`NumAssets` matrix of `NumSamples` prices or returns at a given periodicity for a collection of `NumAssets` assets or a `fints` object with `NumSamples` observations and `NumAssets` time series.

Note: If you want to use the method multiple times and you want to simulate identical scenarios each time the function is called, precede each function call with `rng(seed)` using a specified integer seed.

Tips

You can also use dot notation to estimate mean and covariance of asset returns from data for a PortfolioMAD object.

```
obj = obj.simulateNormalScenariosByData(AssetReturns, NumScenarios, varargin);
```

Input Arguments

obj

A MAD portfolio object [PortfolioMAD].

AssetReturns

Either a matrix or `fints` object that contains asset data that can be converted into asset returns [NumSamples-by-NumAssets] matrix.

NumScenarios

Number of scenarios to simulate [positive integer].

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

DataFormat

DataFormat values are:

- **Returns** — Data in AssetReturns are asset total returns.
- **Prices** — Data in AssetReturns are asset total return prices.

Default: Returns

MissingData

MissingData values are:

- `false` — Do not use ECM algorithm to handle NaN values (just exclude NaN values).
- `true` — Use ECM algorithm to handle NaN values.

Default: `false`

GetAssetList

GetAssetList values are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this method and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is currently `Asset`.

If the `GetAssetList` flag is `false`, nothing happens, which is the default.

Default: `false`

Output Arguments

obj

Updated MAD portfolio object [`PortfolioMAD`].

Attributes

Access	<code>public</code>
Static	<code>false</code>
Hidden	<code>false</code>

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Estimate Mean and Covariance of Asset Returns from Data

Given a PortfolioMAD object `p`, use the `simulateNormalScenariosByData` method to simulate multivariate normal asset return scenarios from data.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

RawData = mvnrnd(m, C, 240);
NumScenarios = 2000;

p = PortfolioMAD;
p = simulateNormalScenariosByData(p, RawData, NumScenarios);
p = setDefaultConstraints(p);

disp(p);
```

PortfolioMAD with properties:

```
BuyCost: []
SellCost: []
RiskFreeRate: []
Turnover: []
BuyTurnover: []
SellTurnover: []
NumScenarios: 2000
Name: []
NumAssets: 4
AssetList: []
InitPort: []
AInequality: []
bInequality: []
AEquality: []
bEquality: []
LowerBound: [4x1 double]
UpperBound: []
LowerBudget: 1
```

```
UpperBudget: 1
GroupMatrix: []
LowerGroup: []
UpperGroup: []
    GroupA: []
    GroupB: []
LowerRatio: []
UpperRatio: []
```

Estimate Mean and Covariance of Asset Returns from Market Data

Create a PortfolioMAD object `p` and use the `simulateNormalScenariosByData` method with market data loaded from `CAPMuniverse.mat` to simulate multivariate normal asset return scenarios.

```
load CAPMuniverse
```

```
p = PortfolioMAD('AssetList',Assets(1:12));
disp(p);
```

PortfolioMAD with properties:

```
    BuyCost: []
    SellCost: []
RiskFreeRate: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
NumScenarios: []
    Name: []
    NumAssets: 12
    AssetList: {1x12 cell}
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: []
    UpperBound: []
    LowerBudget: []
    UpperBudget: []
GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
```



```
GroupA: []  
GroupB: []  
LowerRatio: []  
UpperRatio: []
```

Simulate the scenarios from the data for each of the 12 assets from `CAPMuniverse.mat`.

```
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000, 'missingdata', true);
```

- “Simulating Normal Scenarios” on page 6-45

See Also

PortfolioMAD | rng | simulateNormalScenariosByMoments

More About

- “How Stochastic Optimization Works” on page 6-41

simulateNormalScenariosByMoments

Class: PortfolioCVaR

Simulate multivariate normal asset return scenarios from mean and covariance of asset returns for PortfolioCVaR object

Syntax

```
obj = simulateNormalScenariosByMoments(obj,AssetMean,  
AssetCovar,NumScenarios)  
obj = simulateNormalScenariosByMoments(obj,AssetMean,  
AssetCovar,NumScenarios,NumAssets)
```

Description

`obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar,NumScenarios)` to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns or a CVaR portfolio object.

`obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar,NumScenarios,NumAssets)` to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns for a CVaR portfolio object.

Note: This method overwrites existing scenarios associated with the PortfolioCVaR object, and also, possibly, NumScenarios.

If you want to use the method multiple times and you want to simulate identical scenarios each time the function is called, precede each function call with `rng(seed)` using a specified integer seed.

Tips

You can also use dot notation to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns for a CVaR portfolio object.

```
obj = obj.simulateNormalScenariosByMoments(AssetMean, AssetCovar, NumScenarios, NumAssets);
```

Input Arguments

obj

CVaR portfolio object [PortfolioCVaR].

AssetMean

Mean of asset returns [vector]. If **AssetMean** is a scalar and the number of assets is known, scalar expansion occurs. If it is not possible to determine the number of assets, this method assumes that **NumAssets** = 1.

AssetCovar

Covariance of asset returns [matrix]. **AssetCovar** must be a symmetric positive-semidefinite matrix.

- If **AssetCovar** is a scalar and the number of assets is known, a diagonal matrix is formed with the scalar value along the diagonals. If it is not possible to determine the number of assets, this method assumes that **NumAssets** = 1.
- If **AssetCovar** is a vector, a diagonal matrix is formed with the vector along the diagonal.

NumScenarios

Number of scenarios to simulate.

NumAssets

(Optional) Number of assets [scalar].

Output Arguments

obj

Updated CVaR portfolio object [PortfolioCVaR].

Attributes

Access	public
Static	false

Hidden

false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Simulate Multivariate Normal Asset Return Scenarios from Moments

Given PortfolioCVaR object `p`, use the `simulateNormalScenariosByMoments` method to simulate multivariate normal asset return scenarios from moments.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

AssetMean = [.5]
AssetCovar = [.5]
NumScenarios = 100

p = simulateNormalScenariosByMoments(p, AssetMean, AssetCovar, NumScenarios)

AssetMean =

    0.5000

AssetCovar =

    0.5000
```

```
NumScenarios =
```

```
    100
```

```
p =
```

```
PortfolioCVaR with properties:
```

```

    BuyCost: []
    SellCost: []
    RiskFreeRate: []
    ProbabilityLevel: 0.9500
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    NumScenarios: 100
    Name: []
    NumAssets: 4
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: [4x1 double]
    UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
    GroupA: []
    GroupB: []
    LowerRatio: []
    UpperRatio: []

```

- “Simulating Normal Scenarios” on page 5-45

See Also

PortfolioCVaR | rng | simulateNormalScenariosByData

More About

- “How Stochastic Optimization Works” on page 5-41

simulateNormalScenariosByMoments

Class: PortfolioMAD

Simulate multivariate normal asset return scenarios from mean and covariance of asset returns for PortfolioMAD object

Syntax

```
obj = simulateNormalScenariosByMoments(obj,AssetMean,...  
AssetCovar,NumScenarios)  
obj = simulateNormalScenariosByMoments(obj,AssetMean,...  
AssetCovar,NumScenarios,NumAssets)
```

Description

`obj = simulateNormalScenariosByMoments(obj,AssetMean,...
AssetCovar,NumScenarios)` to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns or a MAD portfolio object.

`obj = simulateNormalScenariosByMoments(obj,AssetMean,...
AssetCovar,NumScenarios,NumAssets)` to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns for a MAD portfolio object.

Note: This method overwrites existing scenarios associated with the PortfolioMAD object, and also, possibly, NumScenarios.

If you want to use the method multiple times and you want to simulate identical scenarios each time the function is called, precede each function call with `rng(seed)` using a specified integer seed.

Tips

You can also use dot notation to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns for a MAD portfolio object.

```
obj = obj.simulateNormalScenariosByMoments(AssetMean, AssetCovar, NumScenarios, NumAssets);
```

Input Arguments

obj

MAD portfolio object [PortfolioMAD].

AssetMean

Mean of asset returns [vector]. If **AssetMean** is a scalar and the number of assets is known, scalar expansion occurs. If it is not possible to determine the number of assets, this method assumes that **NumAssets** = 1.

AssetCovar

Covariance of asset returns [matrix]. **AssetCovar** must be a symmetric positive-semidefinite matrix.

- If **AssetCovar** is a scalar and the number of assets is known, a diagonal matrix is formed with the scalar value along the diagonals. If it is not possible to determine the number of assets, this method assumes that **NumAssets** = 1.
- If **AssetCovar** is a vector, a diagonal matrix is formed with the vector along the diagonal.

NumScenarios

Number of scenarios to simulate.

NumAssets

(Optional) Number of assets [scalar].

Output Arguments

obj

Updated MAD portfolio object [PortfolioMAD].

Attributes

Access

public

Static	false
Hidden	false

To learn about attributes of methods, see “Method Attributes” in the MATLAB Object-Oriented Programming documentation.

Examples

Simulate Multivariate Normal Asset Return Scenarios from Moments

Given PortfolioMAD object `p`, use the `simulateNormalScenariosByMoments` method to simulate multivariate normal asset return scenarios from moments.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

AssetMean = [.5]
AssetCovar = [.5]
NumScenarios = 100

p = simulateNormalScenariosByMoments(p, AssetMean, AssetCovar, NumScenarios)

AssetMean =

    0.5000

AssetCovar =
```

0.5000

NumScenarios =

100

p =

PortfolioMAD with properties:

```
    BuyCost: []
    SellCost: []
    RiskFreeRate: []
    Turnover: []
    BuyTurnover: []
    SellTurnover: []
    NumScenarios: 100
    Name: []
    NumAssets: 4
    AssetList: []
    InitPort: []
    AInequality: []
    bInequality: []
    AEquality: []
    bEquality: []
    LowerBound: [4x1 double]
    UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
    GroupA: []
    GroupB: []
    LowerRatio: []
    UpperRatio: []
```

- “Simulating Normal Scenarios” on page 6-45

See Also

PortfolioMAD | rng | simulateNormalScenariosByData

More About

- “How Stochastic Optimization Works” on page 6-41

setfield

Set content of specific field

Syntax

```
newfts = setfield(tsobj, field, V)
```

```
newfts = setfield(tsobj, field, {dates}, V)
```

Description

`setfield` treats the contents of fields in a time series object (`tsobj`) as fields in a structure.

`newfts = setfield(tsobj, field, V)` sets the contents of the specified field to the value `V`. This is equivalent to the syntax `S.field = V`.

`newfts = setfield(tsobj, field, {dates}, V)` sets the contents of the specified field for the specified dates. `dates` can be individual cells of date strings or a cell of a date string range using the `::` operator, for example, `'03/01/99::03/31/99'`. Dates can contain time-of-day information.

Examples

Example 1. Set the closing value for all days to 3890.

```
load dji30short
format bank
myfts1 = setfield(myfts1, 'Close', 3890)
```

```
myfts1 =
```

```
desc: DJI30MAR94.dat
freq: Daily (1)
```

'dates: (20)'	'Open: (20)'	'High: (20)'	'Low: (20)'	'Close: (20)'
'04-Mar-1994'	[3830.90]	[3868.04]	[3800.50]	[3890.00]
'07-Mar-1994'	[3851.72]	[3882.40]	[3824.71]	[3890.00]
'08-Mar-1994'	[3858.48]	[3881.55]	[3822.45]	[3890.00]
'09-Mar-1994'	[3853.97]	[3874.52]	[3817.95]	[3890.00]

```

'10-Mar-1994' [ 3852.57] [ 3865.51] [ 3801.63] [ 3890.00]
'11-Mar-1994' [ 3832.58] [ 3872.83] [ 3806.69] [ 3890.00]
'14-Mar-1994' [ 3870.29] [ 3894.21] [ 3835.96] [ 3890.00]
'15-Mar-1994' [ 3863.41] [ 3888.46] [ 3826.85] [ 3890.00]
'16-Mar-1994' [ 3851.03] [ 3879.53] [ 3819.94] [ 3890.00]
'17-Mar-1994' [ 3853.62] [ 3891.34] [ 3821.66] [ 3890.00]
'18-Mar-1994' [ 3865.42] [ 3911.78] [ 3838.65] [ 3890.00]
'21-Mar-1994' [ 3878.38] [ 3898.25] [ 3838.65] [ 3890.00]
'22-Mar-1994' [ 3865.71] [ 3896.23] [ 3840.66] [ 3890.00]
'23-Mar-1994' [ 3868.88] [ 3901.41] [ 3839.80] [ 3890.00]
'24-Mar-1994' [ 3849.88] [ 3865.42] [ 3792.58] [ 3890.00]
'25-Mar-1994' [ 3827.13] [ 3826.85] [ 3774.73] [ 3890.00]
'28-Mar-1994' [ 3776.46] [ 3793.45] [ 3719.74] [ 3890.00]
'29-Mar-1994' [ 3757.17] [ 3771.86] [ 3689.23] [ 3890.00]
'30-Mar-1994' [ 3688.36] [ 3718.88] [ 3612.36] [ 3890.00]
'31-Mar-1994' [ 3639.71] [ 3673.10] [ 3544.12] [ 3890.00]

```

Example 2. Set values for specific times on specific days.

First create a financial time series containing time-of-day data.

```

dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
        '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates,1),1), ...
                        times]);
myfts = fints(dates_times, [(1:4)'; nan; 6], {'Data1'}, 1, ...
            'My FINTS')

myfts =

```

```

desc: My FINTS
freq: Daily (1)

'dates: (6)'      'times: (6)'      'Data1: (6)'
'01-Jan-2001'    '11:00'          [          1]
'      "      '12:00'          [          2]
'02-Jan-2001'    '11:00'          [          3]
'      "      '12:00'          [          4]
'03-Jan-2001'    '11:00'          [         NaN]
'      "      '12:00'          [          6]

```

Now use `setfield` to replace the data in `myfts` with new data starting at 12:00 on January 1, 2001 and ending at 11:00 on January 3, 2001.

```

S = setfield(myfts, 'Data1', ...
            {'01-Jan-2001 12:00::03-Jan-2001 11:00'}, (102:105))

S =

```

```

desc: My FINTS
freq: Daily (1)

```

```
'dates: (6)'      'times: (6)'      'Data1: (6)'  
'01-Jan-2001'    '11:00'           [      1.00]  
'      "      '12:00'           [     102.00]  
'02-Jan-2001'    '11:00'           [     103.00]  
'      "      '12:00'           [     104.00]  
'03-Jan-2001'    '11:00'           [     105.00]  
'      "      '12:00'           [      6.00]
```

See Also

chfield | fieldnames | getfield | isfield | rmfield

sharpe

Compute Sharpe ratio for one or more assets

Syntax

```
sharpe(Asset)
```

```
sharpe(Asset, Cash)
```

```
Ratio = sharpe(Asset, Cash)
```

Arguments

Asset	NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES observations of asset returns for NUMSERIES asset return series.
Cash	(Optional) Either a scalar return for a riskless asset or a vector of asset returns to be a proxy for a riskless asset. In either case, the return periodicity must be the same as the periodicity of Asset. For example, if Asset is monthly data, then Cash must be monthly returns. If no value is supplied, the default value for Cash returns is 0.

Description

Given NUMSERIES assets with NUMSAMPLES returns for each asset in a NUMSAMPLES-by-NUMSERIES matrix **Asset** and given either a scalar **Cash** asset return or a vector of **Cash** asset returns, the Sharpe ratio is computed for each asset.

The output is **Ratio**, a 1-by-NUMSERIES row vector of Sharpe ratios for each series in **Asset**. Any series in **Asset** with standard deviation of returns equal to 0 will have a NaN value for its Sharpe ratio.

Note: If **Cash** is a vector, **Asset** and **Cash** need not have the same number of returns but must have the same periodicity of returns. The classic Sharpe ratio assumes that **Cash**

is riskless. In reality, a short-term cash rate is not necessarily riskless. NaN values in the data are ignored.

Examples

See “Sharpe Ratio” on page 7-5.

References

William F. Sharpe, "Mutual Fund Performance," *Journal of Business*, Vol. 39, No. 1, Part 2, January 1966, pp. 119-138.

See Also

inforatio | portalpha

size

Number of dates and data series

Syntax

```
szfts = size(tsobj, dim)
```

```
[numRows, numCols] = size(tsobj)
```

Arguments

<code>tsobj</code>	Financial time series object.
<code>dim</code>	(Optional) A scalar that specifies the following dimension: <code>dim = 1</code> returns number of dates (rows). <code>dim = 2</code> returns number of data series (columns).

Description

`szfts = size(tsobj)` returns the number of dates (rows) and the number of data series (columns) in the financial time series object `tsobj`. The result is returned in the vector `szfts`, whose first element is the number of dates and second is the number of data series.

`szfts = size(tsobj, dim)` specifies the dimension you want to extract.

`numRows` returns a scalar representing the number of dates (rows).

`numCols` returns a scalar representing the number of data series (columns).

See Also

`length` | `size`

smoothts

Smooth data

Syntax

```
output = smoothts(input)
```

```
output = smoothts(input, 'b', wsize)
```

```
output = smoothts(input, 'g', wsize, stdev)
```

```
output = smoothts(input, 'e', n)
```

Arguments

input	Financial time series object or a row-oriented matrix. In a row-oriented matrix, each row represents an individual set of observations.
'b', 'g', or 'e'	Smoothing method (essentially the type of filter used). Can be Exponential (e), Gaussian (g), or Box (b). Default = b.
wsize	Window size (scalar). Default = 5.
stdev	Scalar that represents the standard deviation of the Gaussian window. Default = 0.65.
n	For Exponential method, specifies window size or exponential factor, depending upon value. <ul style="list-style-type: none">• $n > 1$ (window size) or period length• $n < 1$ and > 0 (exponential factor: alpha)• $n = 1$ (either window size or alpha) If n is not supplied, the defaults are wsize = 5 and alpha = 0.3333.

Description

`smoothts` smooths the input data using the specified method.

`output = smoothts(input)` smooths the input data using the default Box method with window size, `wsize`, of 5.

`output = smoothts(input, 'b', wsize)` smooths the input data using the Box (simple, linear) method. `wsize` specifies the width of the box to be used.

`output = smoothts(input, 'g', wsize, stdev)` smooths the input data using the Gaussian window method.

`output = smoothts(input, 'e', n)` smooths the input data using the Exponential method. `n` can represent the window size (period length) or alpha. If `n > 1`, `n` represents the window size. If `0 < n < 1`, `n` represents alpha, where

$$\alpha = \frac{2}{wsize + 1}.$$

If `input` is a financial time series object, `output` is a financial time series object identical to `input` except for contents. If `input` is a row-oriented matrix, `output` is a row-oriented matrix of the same length.

See Also

`tsmovavg`

sortfts

Sort financial time series

Syntax

```
sfts = sortfts(tsobj)
sfts = sortfts(tsobj, flag)
sfts = sortfts(tsobj, seriesnames, flag)
[sfts, sidx] = sortfts(...)
```

Arguments

tsobj	Financial time series object.
flag	(Optional) Sort order: flag = 1; increasing order (default) flag = -1; decreasing order
seriesnames	(Optional) String containing a data series name or cell array containing a list of data series names.

Description

`sfts = sortfts(tsobj)` sorts the financial time series object `tsobj` in increasing order based only upon the `'dates'` vector if `tsobj` does not contain time-of-day information. If the object includes time-of-day information, the sort is based upon a combination of the `'dates'` and `'times'` vectors. The `'times'` vector cannot be sorted individually.

`sfts = sortfts(tsobj, flag)` sets the order of the sort. `flag = 1`: increasing date and time order. `flag = -1`: decreasing date and time order.

`sfts = sortfts(tsobj, seriesnames, flag)` sorts the financial time series object `tsobj` based upon the data series name(s) `seriesnames`. The `seriesnames` argument can be a single string containing a data series name or a cell array containing a list of data series names. If the optional `flag` is set to `-1`, the sort is in decreasing order.

`[sfts, sidx] = sortfts(...)` additionally returns the index of the original object `tsobj` sorted based on `'dates'` or specified data series name(s).

See Also

`issorted` | `sort` | `sortrows`

spctkd

Slow stochastics

Syntax

```
[spctk, spctd] = spctkd(fastpctk, fastpctd)
[spctk, spctd] = spctkd([fastpctk fastpctd])
[spctk, spctd] = spctkd(fastpctk, fastpctd, dperiods, dmamethod)
[spctk, spctd] = spctkd([fastpctk fastpctd], dperiods, dmamethod)
skdts = spctkd(tsobj)
skdts = spctkd(tsobj, dperiods, dmamethod)
skdts = spctkd(tsobj, dperiods, dmamethod, ParameterName,
ParameterValue, ...)
```

Arguments

fastpctk	Fast stochastic F%K (vector).
fastpctd	Fast stochastic F%D (vector).
dperiods	(Optional) %D periods. Default = 3.
dmamethod	(Optional) %D moving average method. Default = 'e' (exponential).
tsobj	Financial time series object.

Description

[spctk, spctd] = spctkd(fastpctk, fastpctd) calculates the slow stochastics S%K and S%D. spctk and spctd are column vectors representing the respective slow stochastics. The inputs must be single column-oriented vectors containing the fast stochastics F%K and F%D.

`[spctk, spctd] = spctkd([fastpctk fastpctd])` accepts a two-column matrix as input. The first column contains the fast stochastic F%K values, and the second contains the fast stochastic F%D values.

`[spctk, spctd] = spctkd(fastpctk, fastpctd, dperiods, dmamethod)` calculates the slow stochastics, S%K and S%D, using the value of `dperiods` to set the number of periods and `dmamethod` to indicate the moving average method. The inputs `fastpctk` and `fastpctd` must contain the fast stochastics, F%K and F%D, in column orientation. `spctk` and `spctd` are column vectors representing the respective slow stochastics.

Valid moving average methods for %D are exponential ('e'), triangular ('t'), and modified ('m'). See `tsmovavg` for explanations of these methods.

`[spctk, spctd] = spctkd([fastpctk fastpctd], dperiods, dmamethod)` accepts a two-column matrix rather than two separate vectors. The first column contains the F%K values, and the second contains the F%D values.

`skdts = spctkd(tsobj)` calculates the slow stochastics, S%K and S%D. `tsobj` must contain the fast stochastics, F%K and F%D, in data series named `PercentK` and `PercentD`. The `skdts` output is a financial time series object with the same dates as `tsobj`. Within `tsobj` the two series `SLOWPctK` and `SLOWPctD` represent the respective slow stochastics.

`skdts = spctkd(tsobj, dperiods, dmamethod)` lets you specify the length and the method of the moving average used to calculate S%D values.

`skdts = spctkd(tsobj, dperiods, dmamethod, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `KName`: F%K series name
- `DName`: F%D series name

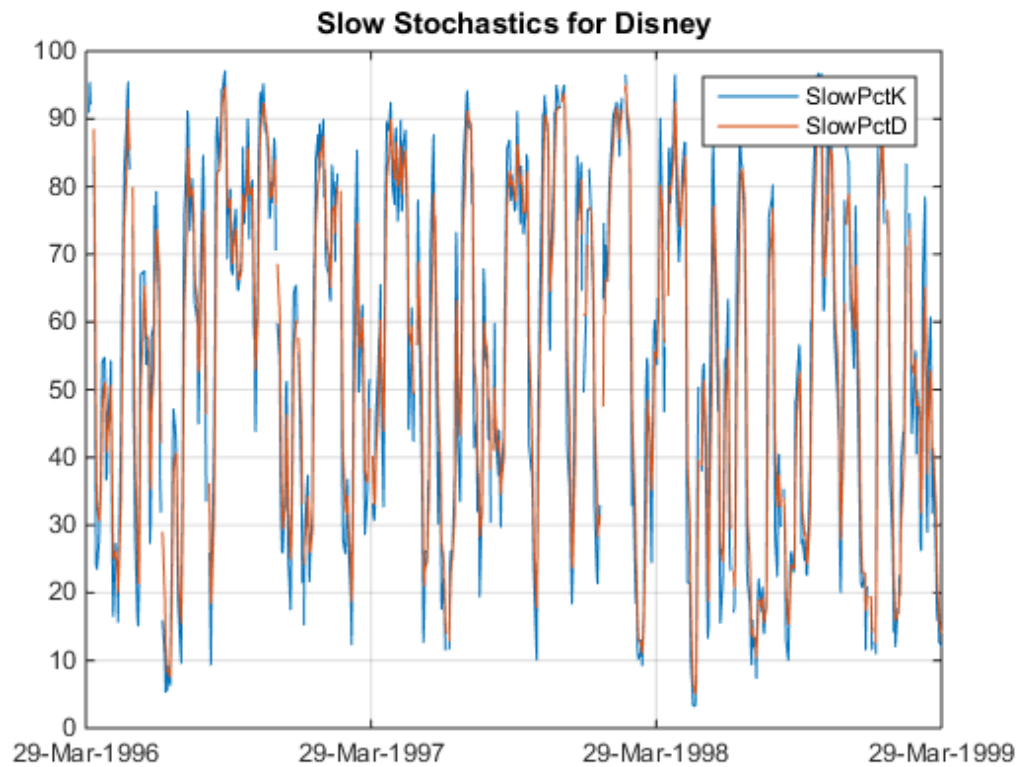
Parameter values are the strings that represent the valid parameter names.

Examples

Calculate the Slow Stochastics

This example shows how to calculate the slow stochastics for Disney stock and plot the results.

```
load disney.mat
dis_FastStoch = fpctkd(dis);
dis_SlowStoch = spctkd(dis_FastStoch);
plot(dis_SlowStoch)
title('Slow Stochastics for Disney')
```

References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 268 - 271.

See Also

fpctkd | stochosc | tsmovavg

std

Standard deviation

Syntax

```
tsstd = std(tsobj)
```

```
tsstd = std(tsobj, flag)
```

Arguments

tsobj	Financial time series object.
flag	(Optional) Normalization factor: flag = 1 normalizes by n (number of observations). flag = 0 normalizes by n-1.

Description

`tsstd = std(tsobj)` computes the standard deviation of each data series in the financial time series object `tsobj` and returns the results in `tsstd`. The `tsstd` output is a structure with field name(s) identical to the data series name(s).

`tsstd = std(tsobj, flag)` normalizes the data as indicated by `flag`.

See Also

`hist` | `mean`

stochosc

Stochastic oscillator

Syntax

```
stosc = stochosc(highp, lowp, closep)
stosc = stochosc([highp lowp closep])
stosc = stochosc(highp, lowp, closep, kperiods, dperiods, dmamethod)
stosc = stochosc([highp lowp closep], kperiods, dperiods, dmamethod)
stoscts = stochosc(tsobj, kperiods, dperiods, dmamethod)
stoscts = stochosc(tsobj, kperiods, dperiods, dmamethod,
ParameterName, ParameterValue, ...)
```

Arguments

highp	High price (vector).
lowp	Low price (vector).
closep	Closing price (vector).
kperiods	(Optional) %K periods. Default = 10.
dperiods	(Optional) %D periods. Default = 3.
damethod	(Optional) %D moving average method. Default = 'e' (exponential).
tsobj	Financial time series object.

Description

`stosc = stochosc(highp, lowp, closep)` calculates the fast stochastics $F\%K$ and $F\%D$ from the stock price data `highp` (high prices), `lowp` (low prices), and `closep` (closing prices). `stosc` is a two-column matrix whose first column is the $F\%K$ values and second is the $F\%D$ values.

`stosc = stochosc([highp lowp closep])` accepts a three-column matrix of high (`highp`), low (`lowp`), and closing prices (`closep`), in that order.

`stosc = stochosc(highp, lowp, closep, kperiods, dperiods, dmamethod)` calculates the fast stochastics `F%K` and `F%D` from the stock price data `highp` (high prices), `lowp` (low prices), and `closep` (closing prices). `kperiods` sets the %K period. `dperiods` sets the %D period. `dmamethod` specifies the %D moving average method. Valid moving average methods for %D are exponential ('e') and triangular ('t'). See `tsmovavg` for explanations of these methods.

`stosc= stochosc([highp lowp closep], kperiods, dperiods, dmamethod)` accepts a three-column matrix of high (`highp`), low (`lowp`), and closing prices (`closep`), in that order.

`stoscts = stochosc(tsobj, kperiods, dperiods, dmamethod)` calculates the fast stochastics `F%K` and `F%D` from the stock price data in the financial time series object `tsobj`. `tsobj` must minimally contain the series `High` (high prices), `Low` (low prices), and `Close` (closing prices). `stoscts` is a financial time series object with similar dates to `tsobj` and two data series named `SOK` and `SOD`.

`stoscts = stochosc(tsobj, kperiods, dperiods, dmamethod, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

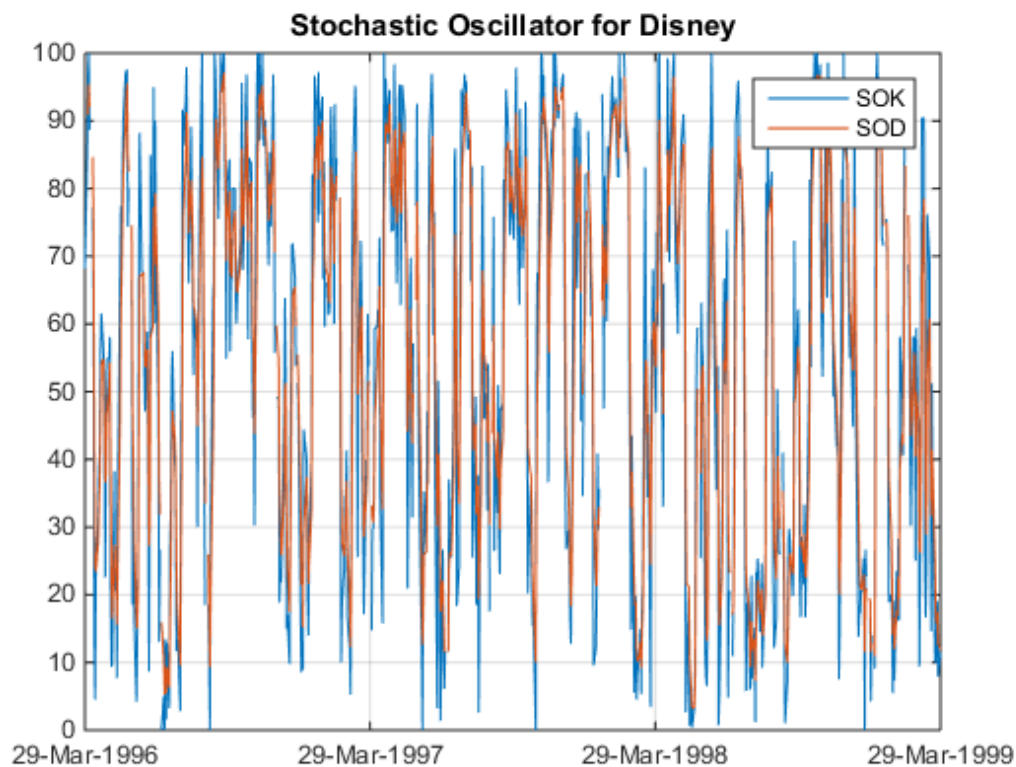
Examples

Compute the Stochastic Oscillator

This example shows how to compute the stochastic oscillator for Disney stock and plot the results.

```
load disney.mat
dis_StochOsc = stochosc(dis);
```

```
plot(dis_StochOsc)  
title('Stochastic Oscillator for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 268 - 271.

See Also

fpctkd | spctkd

subasgn

Content assignment

Syntax

```
subasgn
```

Description

`subasgn` assigns content to a component within a financial time series object. `subasgn` supports integer indexing or date string indexing into the time series object with values assigned to the designated components. *Serial date numbers cannot be used as indices.* To use date string indexing, enclose the date string(s) in a pair of single quotation marks ' '.

You can use integer indexing on the object as in any other MATLAB matrix. It will return the appropriate entry(ies) from the object.

You must specify the component to which you want to assign values. An assigned value must be either a scalar or a column vector.

Examples

Given a time series `myfts` with a default data series name of `series1`,

```
myfts.series1('07/01/98:07/03/98') = [1 2 3]';
```

assigns the values 1, 2, and 3 corresponding to the first three days of July, 1998.

```
myfts('07/01/98:07/05/98')
```

```
ans =
```

```
desc: Data Assignment
freq: Daily (1)

'dates: (5)'    'series1: (5)'
```

```

'01-Jul-1998' [          1]
'02-Jul-1998' [          2]
'03-Jul-1998' [          3]
'04-Jul-1998' [      4561.2]
'05-Jul-1998' [      5612.3]

```

When the financial time series object contains a time-of-day specification, you can assign data to a specific time on a specific day. For example, create a financial time series object called `timeday` containing both dates and times:

```

dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
'02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates,1),1), ...
times]);
timeday = fints(dates_times,(1:6),{'Data1'},1,'My first FINTS')

```

```
timeday =
```

```

desc: My first FINTS
freq: Daily (1)

'dates: (6)'      'times: (6)'      'Data1: (6)'
'01-Jan-2001'    '11:00'           [          1]
'    "    '    '12:00'           [          2]
'02-Jan-2001'    '11:00'           [          3]
'    "    '    '12:00'           [          4]
'03-Jan-2001'    '11:00'           [          5]
'    "    '    '12:00'           [          6]

```

Use integer indexing to assign the value 999 to the first item in the object.

```
timeday(1) = 999
```

```
timeday =
```

```

desc: My first FINTS
freq: Daily (1)

'dates: (6)'      'times: (6)'      'Data1: (6)'
'01-Jan-2001'    '11:00'           [          999]
'    "    '    '12:00'           [          2]
'02-Jan-2001'    '11:00'           [          3]
'    "    '    '12:00'           [          4]
'03-Jan-2001'    '11:00'           [          5]
'    "    '    '12:00'           [          6]

```

For value assignment using date strings, enclose the string in single quotation marks. If a date has multiple times, designating only the date and assigning a value results in every element of that date taking on the assigned value. For example, to assign the value 0.5 to all times-of-day on January 1, 2001, enter

```
timedata('01-Jan-2001') = 0.5
```

The result is

```
timedata =  
  
desc: My first FINTS  
freq: Daily (1)  
  
'dates: (6)'      'times: (6)'      'Data1: (6)'  
'01-Jan-2001'    '11:00'           [ 0.5000]  
'      "      '12:00'           [ 0.5000]  
'02-Jan-2001'    '11:00'           [ 3]  
'      "      '12:00'           [ 4]  
'03-Jan-2001'    '11:00'           [ 5]  
'      "      '12:00'           [ 6]
```

To access the individual components of the financial time series object, use the structure syntax. For example, to assign a range of data to all the data items in the series `Data1`, you can use

```
timedata.Data1 = (0: .1 : .5)'  
  
timedata =  
  
desc: My first FINTS  
freq: Daily (1)  
  
'dates: (6)'      'times: (6)'      'Data1: (6)'  
'01-Jan-2001'    '11:00'           [ 0]  
'      "      '12:00'           [ 0.1000]  
'02-Jan-2001'    '11:00'           [ 0.2000]  
'      "      '12:00'           [ 0.3000]  
'03-Jan-2001'    '11:00'           [ 0.4000]  
'      "      '12:00'           [ 0.5000]
```

See Also

`datestr` | `subsref`

subsref

Subscripted reference

Syntax

subsref

Description

subsref implements indexing for a financial time series object. Integer indexing or date (and time) string indexing is allowed. *Serial date numbers cannot be used as indices.*

To use date string indexing, enclose the date string(s) in a pair of single quotation marks ' '.

You can use integer indexing on the object as in any other MATLAB matrix. It returns the appropriate entry(ies) from the object.

Additionally, **subsref** lets you access the individual components of the object using the structure syntax.

Examples

Create a time series named **myfts**:

```
myfts = fints((datenum('07/01/98'):datenum('07/01/98')+4)',...  
[1234.56; 2345.61; 3456.12; 4561.23; 5612.34], [], 'Daily',...  
'Data Reference');
```

Extract the data for the single day July 1, 1998:

```
myfts('07/01/98')
```

```
ans =
```

```
desc: Data Reference  
freq: Daily (1)
```

```
'dates: (1)'      'series1: (1)'  
'01-Jul-1998'    [      1234.6]
```

Now, extract the data for the range of dates July 1, 1998, through July 5, 1998:

```
myfts('07/01/98::07/03/98')
```

```
ans =  
  desc: Data Reference  
  freq: Daily (1)  
  'dates: (3)'      'series1: (3)'  
  '01-Jul-1998'    [      1234.6]  
  '02-Jul-1998'    [      2345.6]  
  '03-Jul-1998'    [      3456.1]
```

You can use the MATLAB structure syntax to access the individual components of a financial time series object. To get the description field of `myfts`, enter

```
myfts.desc
```

at the command line, which returns

```
ans =  
Data Reference
```

Similarly

```
myfts.series1
```

returns

```
ans =  
  desc: Data Reference  
  freq: Daily (1)  
  'dates: (5)'      'series1: (5)'  
  '01-Jul-1998'    [      1234.6]  
  '02-Jul-1998'    [      2345.6]  
  '03-Jul-1998'    [      3456.1]  
  '04-Jul-1998'    [      4561.2]  
  '05-Jul-1998'    [      5612.3]
```

The syntax for integer indexing is the same as for any other MATLAB matrix. Create a new financial time series object containing both dates and times:

```
dates = ['01-Jan-2001'; '01-Jan-2001'; '02-Jan-2001'; ...
```

```

        '02-Jan-2001'; '03-Jan-2001'; '03-Jan-2001'];
times = ['11:00'; '12:00'; '11:00'; '12:00'; '11:00'; '12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates,1),1), ...
                        times]);
anewfts = fints(dates_times, (1:6), {'Data1'}, 1, 'Another FinTs');

```

Use integer indexing to extract the second and third data items from the object.

```
anewfts(2:3)
```

```
ans =
```

```

desc: Another FinTs
freq: Daily (1)

'dates: (2)'   'times: (2)'   'Data1: (2)'
'01-Jan-2001' '12:00'         [          2]
'02-Jan-2001' '11:00'         [          3]

```

For date or string enclose the indexing string in a pair of single quotation marks.

If there is one date with multiple times, indexing with only the date returns all the times for that specific date:

```
anewfts('01-Jan-2001')
```

```
ans =
```

```

desc: Another FinTs
freq: Daily (1)

'dates: (2)'   'times: (2)'   'Data1: (2)'
'01-Jan-2001' '11:00'         [          1]
'           ' '12:00'         [          2]

```

To specify one specific date and time, index with that date and time:

```
anewfts('01-Jan-2001 12:00')
```

```
ans =
```

```

desc: Another FinTs
freq: Daily (1)

'dates: (1)'   'times: (1)'   'Data1: (1)'
'01-Jan-2001' '12:00'         [          2]

```

To specify a range of dates and times, use the double colon (:) operator:

```
anewfts('01-Jan-2001 12:00::03-Jan-2001 11:00')  
  
ans =  
  
    desc: Another FinTs  
    freq: Daily (1)  
  
    'dates: (4)'    'times: (4)'    'Data1: (4)'  
    '01-Jan-2001'  '12:00'    [          2]  
    '02-Jan-2001'  '11:00'    [          3]  
    '          '    '12:00'    [          4]  
    '03-Jan-2001'  '11:00'    [          5]
```

To request all the dates, times, and data, use the `::` operator without specifying any specific date or time:

```
anewfts('::')
```

See Also

[datestr](#) | [fts2mat](#) | [subsasgn](#)

targetreturn

Portfolio weight accuracy

Syntax

```
return = targetreturn(Universe, Window, Offset, Weights)
```

Arguments

Universe	Number of observations (NUMOBS) by number of assets plus one (NASSETS + 1) array containing total return data for a group of securities. Each row represents an observation. Column 1 contains MATLAB serial date numbers. The remaining columns contain the total return data for each security.
Window	Number of data periods used to calculate frontier.
Offset	Increment in number of periods at which each frontier is generated.
Weights	Number of assets (NASSETS) by number of curves (NCURVES) matrix of asset allocation weights needed to obtain the target rate of return.

Description

`return = targetreturn(Universe, Window, Offset, Weights)` computes target return values for each window of data and given portfolio weights. These values should match the input target return used with `selectreturn`.

See Also

`frontier` | `portopt` | `selectreturn`

taxedrr

After-tax rate of return

Syntax

Return = taxedrr(PreTaxReturn, TaxRate)

Arguments

PreTaxReturn	Nominal rate of return. Enter as a decimal fraction.
TaxRate	Tax rate. Enter as a decimal fraction.

Description

Return = taxedrr(PreTaxReturn, TaxRate) calculates the after-tax rate of return.

Examples

Calculate the After-Tax Rate of Return

This example shows how to calculate the after-tax rate of return, given an investment that has a 12% nominal rate of return and is taxed at a 30% rate.

```
Return = taxedrr(0.12, 0.30)
```

```
Return =
```

```
0.0840
```

See Also

effrr | irr | mirr | nomrr | xirr

tbilldisc2yield

Convert Treasury bill discount to equivalent yield

Syntax

```
[BEYield MMYield] = tbilldisc2yield(Discount, Settle, Maturity)
```

Arguments

Discount	Discount rate of Treasury bills in decimal. The discount rate basis is actual/360.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.

Inputs must either be a scalar or a vector of size equal to the number of Treasury bills (NTBILLS) by 1 or 1-by-NTBILLS.

Description

[BEYield MMYield] = tbilldisc2yield(Yield, Settle, Maturity) converts the discount rate on Treasury bills into their respective money-market or bond-equivalent yields.

BEYield is an NTBILLS-by-1 vector of bond-equivalent yields. The bond-equivalent yield basis is actual/365.

MMYield is an NTBILLS-by-1 vector of money-market yields. The money-market yield basis is actual/360.

Examples

Convert the Discount Rate on Treasury Bills

This example shows how to convert the discount rate on Treasury bills into their respective money-market or bond-equivalent yields, given a Treasury bill with the following characteristics.

```
Discount = 0.0497;  
Settle = '01-Oct-02';  
Maturity = '31-Mar-03';
```

```
[BEYield MMYield] = tbilldisc2yield(Discount, Settle, Maturity)
```

```
BEYield =
```

```
0.0517
```

```
MMYield =
```

```
0.0510
```

- “Computing Treasury Bill Price and Yield”

More About

- “Treasury Bills Defined”

References

This function adheres to *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*, Volume 1, 3rd edition, pp. 44 - 45 (on Treasury bills), and *Money Market and Bond Calculation* by Stigum and Robinson.

See Also

tbillyield2disc | zeroyield

tbillprice

Price Treasury bill

Syntax

Price = tbillprice(Rate, Settle, Maturity, Type)

Arguments

Rate	Bond-equivalent yield, money-market yield, or discount rate in decimal.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.
Type	(Optional) Rate type. Determines how to interpret values entered in Rate . 1 = money market (default). 2 = bond-equivalent. 3 = discount rate.

All arguments must be a scalar or some Treasury bills (NTBILLS) by 1 or 1-by-NTBILLS vector.

Note The bond-equivalent yield basis is actual/365. The money-market yield basis is actual/360. The discount rate basis is actual/360.

Description

Price = tbillprice(Rate, Settle, Maturity, Type) computes the price of a Treasury bill given a yield or discount rate.

Price is an NTBILLS-by-1 vector of T-bill prices for every \$100 face.

Examples

Example 1. Given a Treasury bill with these characteristics, compute the price of the Treasury bill using the bond-equivalent yield as input.

```
Rate = 0.045;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

Type = 2;

Price = tbillprice(Rate, Settle, Maturity, Type)

Price =

    97.8172
```

Example 2. Use `tbillprice` to price a portfolio of Treasury bills.

```
Rate = [0.045; 0.046];
Settle = {'02-Jan-02'; '01-Mar-02'};
Maturity = {'30-June-02'; '30-June-02'};
Type = [2 3];

Price = tbillprice(Rate, Settle, Maturity, Type)

Price =

    97.8408
    98.4539
```

More About

- “Treasury Bills Defined”

References

This function adheres to *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*, Volume 1, 3rd edition, pp. 44 - 45 (on Treasury bills), and *Money Market and Bond Calculation* by Stigum and Robinson.

See Also

tbillyield | zeroprice

Related Examples

- “Computing Treasury Bill Price and Yield”

tbillrepo

Break-even discount of repurchase agreement

Syntax

```
TBEDiscount = tbillrepo(RepoRate, InitialDiscount, PurchaseDate,  
SaleDate, Maturity)
```

Arguments

RepoRate	The annualized, 360-day based repurchase rate, in decimal.
InitialDiscount	Discount on the Treasury bill on the day of purchase, in decimal.
PurchaseDate	Date the Treasury bill is purchased.
SaleDate	Date the Treasury bill repurchase term is due.
Maturity	Treasury bill maturity date.

All arguments must be a scalar or some Treasury bills (NTBILLS) by 1 or a 1-by-NTBILLS vector.

All dates must be in serial date number format.

Description

TBEDiscount = tbillrepo(RepoRate, InitialDiscount, PurchaseDate, SaleDate, Maturity) computes the true break-even discount of a repurchase agreement. TBEDiscount can be a scalar or vector of size NTBills-by-1.

Examples

Compute the True Break-Even Discount

This example shows how to compute the true break-even discount of a Treasury bill repurchase agreement.

```
RepoRate = [0.045; 0.0475];
InitialDiscount = 0.0475;
PurchaseDate = '3-Jan-2002';
SaleDate = '3-Feb-2002';
Maturity = '3-Apr-2002';

TBEDiscount = tbillrepo(RepoRate, InitialDiscount,...
PurchaseDate, SaleDate, Maturity)
```

```
TBEDiscount =
```

```
    0.0491
    0.0478
```

- “Computing Treasury Bill Price and Yield”

More About

- “Treasury Bills Defined”

References

This function adheres to *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*, Volume 1, 3rd edition, pp. 44 - 45 (on Treasury bills), and *Money Market and Bond Calculation* by Stigum and Robinson.

See Also

tbillprice | tbillval01 | tbillyield

tbillval01

Value of one basis point

Syntax

```
[Val01Disc, Val01MMY, Val01BEY] = tbillval01(Settle, Maturity)
```

Arguments

Settle	Settlement date of Treasury bills. Settle must be earlier than Maturity .
Maturity	Maturity date of Treasury bills.

Description

`[Val01Disc, Val01MMY, Val01BEY] = tbillval01(Settle, Maturity)` calculates the value of one basis point of \$100 Treasury bill face value on the discount rate, money-market yield, or bond-equivalent yield.

`Val01Disc` is the value of one basis point of discount rate.

`Val01MMY` is the value of one basis point of money-market yield.

`Val01BEY` is the value of one basis point of bond-equivalent yield.

All outputs are of size equal to the number of Treasury bills (`NTBILLS`) by 1.

Examples

Compute the Value of One Basis Point

This example shows how to compute the value of one basis point, given a Treasury bill with the following settle and maturity dates.

```
Settle = '01-Mar-03';  
Maturity = '30-June-03';  
[Val01Disc, Val01MMY, Val01BEY] = tbillval01(Settle, Maturity)
```

```
Val01Disc =  
    0.0034
```

```
Val01MMY =  
    0.0034
```

```
Val01BEY =  
    0.0033
```

- “Computing Treasury Bill Price and Yield”

More About

- “Treasury Bills Defined”

References

This function adheres to *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*, Volume 1, 3rd edition, pp 108 - 115, on zero coupon instrument pricing.

See Also

tbilldisc2yield | tbillprice | tbillyield | tbillyield2disc

tbillyield

Yield on Treasury bill

Syntax

```
[MMYield, BEYield, Discount] = tbillyield(Price, Settle, Maturity)
```

Arguments

Price	Price of Treasury bills for every \$100 face value.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.

All arguments must be a scalar or some Treasury bills (NTBILLS) by 1 or 1-by-NTBILLS vector.

Description

[MMYield, BEYield, Discount] = tbillyield(Price, Settle, Maturity) computes the yield of U.S. Treasury bills given **Price**, **Settle**, and **Maturity**. **MMYield** is the money-market yields of the Treasury bills. **BEYield** is the bond equivalent yields of the Treasury bills. **Discount** is the discount rates of the Treasury bills.

All outputs are NTBILLS-by-1 vectors.

Note The money-market yield basis is actual/360. The bond-equivalent yield basis is actual/365. The discount rate basis is actual/360.

Examples

Compute the Yield of U.S. Treasury Bills

This example shows how to compute the yield of U.S. Treasury bills, given a Treasury bill with the following characteristics.


```
Price = 98.75;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

[MMYield, BEYield, Discount] = tbillyield(Price, Settle,...
Maturity)
```

```
MMYield =
    0.0252
```

```
BEYield =
    0.0255
```

```
Discount =
    0.0249
```

- “Computing Treasury Bill Price and Yield”

More About

- “Treasury Bills Defined”

References

This function adheres to *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*, Volume 1, 3rd edition, pp. 44 - 45 (on Treasury bills), and *Money Market and Bond Calculation* by Stigum and Robinson.

See Also

tbilldisc2yield | tbillprice | tbillyield2disc | zeroyield

tbillyield2disc

Convert Treasury bill yield to equivalent discount

Syntax

```
Discount = tbillyield2disc(Yield, Settle, Maturity, Type)
```

Arguments

Yield	Yield of Treasury bills in decimal.
Settle	Settlement date. Settle must be earlier than Maturity .
Maturity	Maturity date.
Type	(Optional) Yield type. Determines how to interpret values entered in Yield . 1 = money market (default). 2 = bond-equivalent.

Inputs must either be a scalar or a vector of size equal to the number of Treasury bills (NTBILLS) by 1 or 1-by-NTBILLS.

Note The money-market yield basis is actual/360. The bond-equivalent yield basis is actual/365. The discount rate basis is actual/360.

Description

`Discount = tbillyield2disc(Yield, Settle, Maturity, Type)` converts the yield on some Treasury bills into their respective discount rates.

`Discount` is a NTBILLS-by-1 vector of T-bill discount rates.

Examples

Given a Treasury bill with these characteristics, compute the discount rate on a money-market basis.

```
Yield = 0.0497;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

Discount = tbillyield2disc(Yield, Settle, Maturity)

Discount =

    0.0485
```

Now recompute the discount on a bond-equivalent basis.

```
Discount = tbillyield2disc(Yield, Settle, Maturity, 2)

Discount =

    0.0478
```

More About

- “Treasury Bills Defined”

References

This function adheres to *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*, Volume 1, 3rd edition, pp. 44 - 45 (on Treasury bills), and *Money Market and Bond Calculation* by Stigum and Robinson.

See Also

tbilldisc2yield

Related Examples

- “Computing Treasury Bill Price and Yield”

tbl2bond

Treasury bond parameters given Treasury bill parameters

Syntax

```
[TBondMatrix, Settle] = tbl2bond(TBillMatrix)
```

Arguments

TBillMatrix	Treasury bill parameters. An n -by-5 matrix where each row describes a Treasury bill. n is the number of Treasury bills. Columns are [Maturity DaysMaturity Bid Asked AskYield] where:
Maturity	Maturity date, as a serial date number. Use <code>datenum</code> to convert date strings to serial date numbers.
DaysMaturity	Days to maturity, as an integer. Days to maturity is quoted on a skip-day basis; the actual number of days from settlement to maturity is <code>DaysMaturity + 1</code> .
Bid	Bid bank-discount rate: the percentage discount from face value at which the bill could be bought, annualized on a simple-interest basis. A decimal fraction.
Asked	Asked bank-discount rate, as a decimal fraction.
AskYield	Asked yield: the bond-equivalent yield from holding the bill to maturity, annualized on a simple-interest basis and assuming a 365-day year. A decimal fraction.

Description

`[TBondMatrix, Settle] = tbl2bond(TBillMatrix)` restates U.S. Treasury bill market parameters in U.S. Treasury bond form as zero-coupon bonds. This function makes Treasury bills directly comparable to Treasury bonds and notes.

TBondMatrix	Treasury bond parameters. An N-by-5 matrix where each row describes an equivalent Treasury (zero-coupon) bond. Columns are [CouponRate Maturity Bid Asked AskYield] where
CouponRate	Coupon rate, which is always 0.
Maturity	Maturity date, as a serial date number. This date is the same as the Treasury bill Maturity date.
Bid	Bid price based on \$100 face value.
Asked	Asked price based on \$100 face value.
AskYield	Asked yield to maturity: the effective return from holding the bond to maturity, annualized on a compound-interest basis.

Examples

Restate U.S. Treasury Bill in U.S. Treasury Bond Form

This example shows how to restate U.S. Treasury bill market parameters in U.S. Treasury bond form, given published Treasury bill market parameters for December 22, 1997.

```
TBill = [datenum('jan 02 1998') 10 0.0526 0.0522 0.0530
         datenum('feb 05 1998') 44 0.0537 0.0533 0.0544
         datenum('mar 05 1998') 72 0.0529 0.0527 0.0540];
```

```
TBond = tbl2bond(TBill)
```

```
TBond =
```

```
1.0e+05 *
    0    7.2976    0.0010    0.0010    0.0000
    0    7.2979    0.0010    0.0010    0.0000
    0    7.2982    0.0010    0.0010    0.0000
```

More About

- “Term Structure of Interest Rates” on page 2-38

See Also
tr2bonds

thirdwednesday

Find third Wednesday of month

Syntax

```
[BeginDates, EndDates] = thirdwednesday(Month, Year)
```

Arguments

Month	Month of delivery for Eurodollar futures.
Year	Four-digit year of delivery for Eurodollar futures, in sequence corresponding to a month in the Month input argument.

Inputs can be scalars or n-by-1 vectors.

Description

[BeginDates, EndDates] = thirdwednesday(Month, Year) computes the beginning and end period date for a LIBOR contract (third Wednesdays of delivery months).

BeginDates is the beginning of three-month period contract as specified by Month and Year.

EndDates is the end of three-month period contract as specified by Month and Year.

Notes

- All dates are returned as serial date numbers. Convert to strings using `datestr`.
 - The function returns duplicates if you supply identical months and years.
 - The function supports dates from January 2000 to December 2099.
-

Examples

Find the third Wednesday dates for swaps commencing in the month of October in the years 2002, 2003, and 2004.

```
Months = [10; 10; 10];  
Year = [2002; 2003; 2004];  
[BeginDates, EndDates] = thirdwednesday(Months, Year);
```

```
datestr(BeginDates)
```

```
ans =
```

```
16-Oct-2002  
15-Oct-2003  
20-Oct-2004
```

```
datestr(EndDates)
```

```
ans =
```

```
16-Jan-2003  
15-Jan-2004  
20-Jan-2005
```


thirtytwo2dec

Thirty-second quotation to decimal

Syntax

```
OutNumber = thirtytwo2dec(InNumber, InFraction)
```

Arguments

InNumber	Scalar or vector of input numbers without fractional component.
InFraction	Scalar or vector of fractional portions of each element in InNumber.

Description

`OutNumber = thirtytwo2dec(InNumber, InFraction)` changes the price quotation for a bond or bond future from a fraction with a denominator of 32 to a decimal.

`OutNumber` represents the sum of `InNumber` and `InFraction` expressed as a decimal.

Examples

Change the Price Quotation for a Bond or Bond Future From a Fraction

This example shows how to change the price quotation for a bond or bond future from a fraction with a denominator of 32 to a decimal, given two bonds that are quoted as 101-25 and 102-31.

```
InNumber = [101; 102];  
InFraction = [25; 31];
```

```
OutNumber = thirtytwo2dec(InNumber, InFraction)
```

```
OutNumber =
```

101.7813
102.9688

See Also

dec2thirtytwo

tick2ret

Convert price series to return series

Syntax

```
[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes, Method)
```

Arguments

TickSeries	Number of observations (NUMOBS) by number of assets (NASSETS) matrix of prices of equity assets. Each column is a price series of an individual asset. First row is oldest observation. Last row is most recent. Observations across a given row occur at the same time for all columns.
TickTimes	(Optional) NUMOBS-by-1 increasing vector of observation times associated with the prices in TickSeries. Times are serial date numbers (day units) or decimal numbers in arbitrary units (for example, yearly). If TickTimes is empty or missing, sequential observation times from 1, 2, ... NUMOBS are assumed.
Method	(Optional) Character string indicating the method to convert prices to asset returns. Must be 'Simple' (default) or 'Continuous'. If Method is 'Simple', tick2ret computes simple periodic returns. If Method is 'Continuous', returns are continuously compounded. Case is ignored for Method.

Description

[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes, Method) computes the asset returns realized between NUMOBS observations of prices of NASSETS assets.

RetSeries is a (NUMOBS - 1)-by-NASSETS time series array of asset returns associated with the prices in TickSeries. The *i*th return is quoted for the period TickTimes(*i*) to

`TickTimes(i+1)` and is not normalized by the time increment between successive price observations. If *Method* is unspecified or 'Simple', the returns are:

```
RetSeries(i) = TickSeries(i+1)/TickSeries(i) - 1
```

If *Method* is 'Continuous', the returns are:

```
RetSeries(i) = log[TickSeries(i+1)/TickSeries(i)]
```

`RetIntervals` is a $(\text{NUMOBS}-1)$ -by-1 column vector of interval times between observations. If `TickTimes` is empty or unspecified, all intervals are assumed to have length 1.

Examples

Convert Price Series to Return Series

This example shows how to convert price series to return series, given periodic returns of two stocks observed in the first, second, third, and fourth quarters.

```
TickSeries = [100 80  
              110 90  
              115 88  
              110 91];
```

```
TickTimes = [0  
             6  
             9  
            12];
```

```
[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes)
```

```
RetSeries =
```

```
    0.1000    0.1250  
    0.0455   -0.0222  
   -0.0435    0.0341
```

```
RetIntervals =
```

```
    6
```

3
3

See Also

ewstats | ret2tick

tick2ret (fts)

Convert price series to return series for time series object

Syntax

```
returnFts = tick2ret(priceFts)
```

```
returnFts = tick2ret(priceFts, 'PARAM1', VALUE1,
'PARAM2', VALUE2', ...)
```

Arguments

priceFts	Financial time series object of prices.
'PARAM1 '	(Optional) <i>Method</i> is a character string indicating the method to convert asset returns to prices. The value must be defined as 'Simple' (default) or 'Continuous'. If <i>Method</i> is 'Simple', tick2ret uses simple periodic returns. If <i>Method</i> is 'Continuous', the function uses continuously compounded returns. Case is ignored for <i>Method</i> .

Description

returnFts = tick2ret(priceFts, 'PARAM1', VALUE1, 'PARAM2', VALUE2', ...) generates a financial time series object of returns.

Note: The *i*'th return is quoted for the period PriceSeries(*i*) to PriceSeries(*i*+1) and is not normalized by the time increment between successive price observations.

If *Method* is unspecified or 'Simple', the prices are

$$\text{ReturnSeries}(i) = \text{PriceSeries}(i+1)/\text{PriceSeries}(i) - 1$$

If *Method* is 'Continuous', the prices are

```
ReturnSeries(i) = log[PriceSeries(i+1)/PriceSeries(i)]
```

Examples

Convert Price Series to Return Series for a fints Object

Compute the return series from the following price series:

```
PriceSeries = [100.0000  100.0000
110.0000  112.0000
115.5000  116.4800
109.7250  122.3040]
```

```
PriceSeries =
```

```
  100.0000  100.0000
  110.0000  112.0000
  115.5000  116.4800
  109.7250  122.3040
```

Use the following dates:

```
Dates = {'18-Dec-2000'
'18-Jun-2001'
'17-Sep-2001'
'18-Dec-2001' }
```

```
Dates =
```

```
'18-Dec-2000'
'18-Jun-2001'
'17-Sep-2001'
'18-Dec-2001'
```

The fints object is:

```
p = fints(Dates, PriceSeries)
```

```
p =
```

```
desc: (none)
freq: Unknown (0)

'dates: (4)'      'series1: (4)'    'series2: (4)'
'18-Dec-2000'    [          100]    [          100]
'18-Jun-2001'    [          110]    [          112]
'17-Sep-2001'    [    115.5000]    [    116.4800]
'18-Dec-2001'    [    109.7250]    [    122.3040]
```

`returnFts` is computed as:

```
tick2ret(p)
```

```
ans =
```

```
desc: (none)
freq: Unknown (0)

'dates: (3)'      'series1: (3)'    'series2: (3)'
'18-Jun-2001'    [      0.1000]    [      0.1200]
'17-Sep-2001'    [      0.0500]    [      0.0400]
'18-Dec-2001'    [     -0.0500]    [      0.0500]
```

Note that for n dates in the original time series, there are $(n-1)$ dates returned for `returnFts` from `tick2ret`. The formula for the date output dates is described as: $\text{RetDate}(i) = \text{PriceDate}(i+1)$.

See Also

`portsim` | `ret2tick`

time2date

Dates from time and frequency

Syntax

Dates = time2date(Settle, TFactors, Compounding, Basis, EndMonthRule)

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings.
TFactors	A vector of time factors corresponding to the compounding value. TFactors must be equal to or greater than zero.
Compounding	<p>(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = 2. This argument determines the formula for the discount factors:</p> <p>Compounding = 1, 2, 3, 4, 6, 12</p> <p>Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units; for example, T = F is one year.</p> <p>Compounding = 365</p> <p>Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.</p> <p>Compounding = - 1</p> <p>Disc = $\exp(-T*Z)$, where T is time in years.</p>
Basis	(Optional) Day-count basis of the instrument. A vector of integers.

	<ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
EndMonthRule	<p>(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.</p>

Description

`Dates = time2date(Settle, TFactors, Compounding, Basis, EndMonthRule)` computes dates corresponding to the times occurring beyond the settlement date.

The `time2date` function is the inverse of `date2time`.

Examples

Show that `date2time` and `time2date` are the inverse of each other. First compute the time factors using `date2time`.

```
Settle = '1-Sep-2002';
Dates = datenum(['31-Aug-2005'; '28-Feb-2006'; '15-Jun-2006';
                '31-Dec-2006']);
Compounding = 2;
Basis = 0;
EndMonthRule = 1;
TFactors = date2time(Settle, Dates, Compounding, Basis,...
                    EndMonthRule)
```

```
TFactors =
```

```
    5.9945
    6.9945
    7.5738
    8.6576
```

Now use the calculated `TFactors` in `time2date` and compare the calculated dates with the original set.

```
Dates_calc = time2date(Settle, TFactors, Compounding, Basis,...
                      EndMonthRule)
```

```
Dates_calc =
```

```
    732555
    732736
    732843
    733042
```

```
datestr(Dates_calc)
```

```
ans =
```

```
31-Aug-2005
28-Feb-2006
15-Jun-2006
31-Dec-2006
```

See Also

`cftimes` | `date2time`

times

Financial time series multiplication

Syntax

```
newfts = tsoj_1 .* tsoj_2
```

```
newfts = tsoj .* array
```

```
newfts = array .* tsoj
```

Arguments

<code>tsoj_1, tsoj_2</code>	Pair of financial time series objects.
<code>array</code>	A scalar value or array with the number of rows equal to the number of dates in <code>tsoj</code> and the number of columns equal to the number of data series in <code>tsoj</code> .

Description

The `times` method multiplies element by element the components of one financial time series object by the components of the other. You can also multiply the entire object by an array.

If an object is to be multiplied by another object, both objects must have the same dates and data series names, although the order need not be the same. The order of the data series, when an object is multiplied by another object, follows the order of the first object.

`newfts = tsoj_1 .* tsoj_2` multiplies financial time series objects element by element.

`newfts = tsoj .* array` multiplies a financial time series object element by element by an array.

`newfts = array .* tsoj` and `newfts = array / tsoj` multiplies an array element by element by a financial time series object.

For financial time series objects, the `times` operation is identical to the `mtimes` operation.

See Also

`minus` | `mtimes` | `plus` | `rdivide`

toannual

Convert to annual

Syntax

```
newfts = toannual(oldfts)
```

```
newfts = toannual(oldfts, 'ParameterName', ParameterValue, ...)
```

Arguments

oldfts	Financial time series object.
--------	-------------------------------

Description

`newfts = toannual(oldfts)` converts a financial time series of any frequency to one of an annual frequency. The default end-of-year is the last business day of the December. `toannual` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as '00:00' for those days that did not previously exist in `oldfts`.

Empty ([]) passed as inputs for parameter pair values for `toannual` will trigger the use of the defaults.

`newfts = toannual(oldfts, 'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

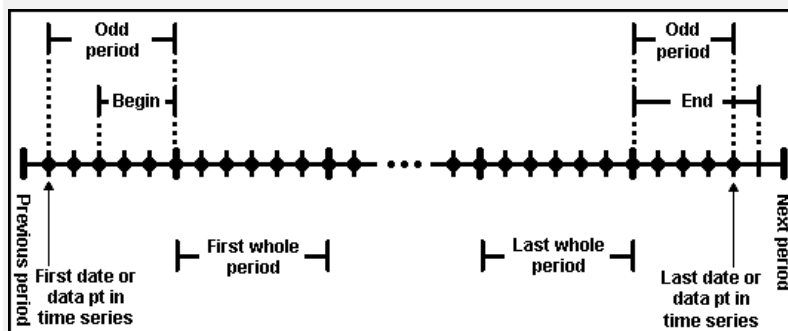
Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values within each year. Data for missing dates are given the value 0.
	Exact	Returns the exact value at the end-of-year date. No data manipulation occurs.

Parameter Name	Parameter Value	Description
	Nearest	(Default) Returns the values located at the end-of-year dates. If there is missing data, Nearest returns the nearest data point preceding the end-of-year date.
	SimpAvg	Returns an averaged annual value that only takes into account dates with data (nonNaN) within each year.
	v21x	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns an averaged end-of-year value using a previous toannual algorithm. This algorithm takes into account all dates and data. For dates that do not contain any data, the data is assumed to be 0.
Note If you set CalcMethod to v21x , settings for all of the following parameter name/parameter value pairs are not supported.		
BusDays	0	Returns a financial time series that ranges from (or between) the first date to the last date in oldfts (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a monthly financial time series that ranges from the first date to the last date in oldfts (excludes NYSE nonbusiness days and holidays and weekends based on AltHolidays and Weekend). If an end-of-month date falls on a nonbusiness day or NYSE holiday, returns the last business day of the month. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ({}).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all annual dates between the start and end dates of <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	Relative	Note: The default is to create a time series with every date at the specified periodicity, which is with <code>DateFilter = Absolute</code> . If you use <code>DateFilter = Relative</code> , the endpoint effects do not apply since only your data defines which dates will appear in the output time series object.
ED	0	Returns only the annual dates that exist in <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> . Annual period ends on the last day or last business day of the month.
	1 - 31	Annual period ends on the last day or last business day of the month. Specifies a particular annual day. Months that do not contain the specified day return the last day (or last business day) of the month (for example, <code>ED = 31</code> does not exist for February.)
EM	1 - 12	(Default) The annual period ends on the last day (or last business day) of the specified month All subsequent annual dates are calculated from this month. Default annual month is December (12).

Parameter Name	Parameter Value	Description
EndPtTol	[Begin, End]	<p>Denotes the minimum number of days that constitute an odd annual period at the endpoints of the time series (before the first time series date and after the last end-of-year date).</p> <p>Begin and End must be -1 or any positive integer greater than or equal to 0.</p> <p>A single value input for 'EndPtTol' is the same as specifying that single value for Begin and End.</p> <p>-1 Exclude odd annual period dates and data from calculations.</p> <p>0 (Default) Include odd annual period dates and data in calculations.</p> <p>n Number of days (any positive integer) that constitute an odd annual period. If there are insufficient days for a complete year, the endpoint data is ignored.</p>

The following diagram is a general depiction of the factors involved in the determination of endpoints for this function.



TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.
----------	-------	--

Parameter Name	Parameter Value	Description
	Last	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
AltHolidays		Vector of dates specifying an alternate set of market closure dates.
	-1	Excludes all holidays.
Weekend		Vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then Weekend = [1 0 0 0 0 0 1].

Examples

Transform Time Series Object from Weekly to Annual Values

This example shows how to transform a time series object from weekly to annual values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdata, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =
```

```
desc: Index
freq: Weekly (2)

'dates: (53)'      'Metric: (53)'
'01-Jan-1999'    [      97.8872]
'08-Jan-1999'    [      97.0847]
'15-Jan-1999'    [     109.6312]
'22-Jan-1999'    [     105.5743]
'29-Jan-1999'    [     108.4028]
'05-Feb-1999'    [     134.4882]
'12-Feb-1999'    [     117.5581]
'19-Feb-1999'    [     106.6683]
'26-Feb-1999'    [     118.2912]
```

'05-Mar-1999'	[105.6835]
'12-Mar-1999'	[128.5836]
'19-Mar-1999'	[115.1746]
'26-Mar-1999'	[131.2854]
'02-Apr-1999'	[130.7116]
'09-Apr-1999'	[123.1684]
'16-Apr-1999'	[107.2975]
'23-Apr-1999'	[91.5625]
'30-Apr-1999'	[78.5738]
'07-May-1999'	[65.2904]
'14-May-1999'	[70.8581]
'21-May-1999'	[72.4807]
'28-May-1999'	[72.9190]
'04-Jun-1999'	[64.3460]
'11-Jun-1999'	[59.8743]
'18-Jun-1999'	[55.0026]
'25-Jun-1999'	[49.4032]
'02-Jul-1999'	[49.9485]
'09-Jul-1999'	[47.8061]
'16-Jul-1999'	[61.0517]
'23-Jul-1999'	[58.9313]
'30-Jul-1999'	[53.9584]
'06-Aug-1999'	[44.8472]
'13-Aug-1999'	[45.0463]
'20-Aug-1999'	[45.1088]
'27-Aug-1999'	[56.4897]
'03-Sep-1999'	[61.2449]
'10-Sep-1999'	[58.1012]
'17-Sep-1999'	[50.8974]
'24-Sep-1999'	[46.5143]
'01-Oct-1999'	[38.0806]
'08-Oct-1999'	[33.6664]
'15-Oct-1999'	[34.2992]
'22-Oct-1999'	[33.4202]
'29-Oct-1999'	[36.9287]
'05-Nov-1999'	[35.1278]
'12-Nov-1999'	[41.8128]
'19-Nov-1999'	[35.8199]
'26-Nov-1999'	[36.9495]
'03-Dec-1999'	[36.2880]
'10-Dec-1999'	[33.8457]
'17-Dec-1999'	[33.3868]
'24-Dec-1999'	[32.7737]
'31-Dec-1999'	[28.5665]

Use `toannual` to obtain the annual aggregate for the `x0` times series.

```
x1 = toannual(x0)
```

```
x1 =
```

```
desc: TOANNUAL: Index  
freq: Annual (6)
```

```
'dates: (1)'      'Metric: (1)'  
'31-Dec-1999'   [      28.5665]
```

See Also

`convertto` | `fints` | `todayly` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly`

todayly

Convert to daily

Syntax

```
newfts = todayly(oldfts)
```

```
newfts = todayly(oldfts, 'ParameterName', ParameterValue, ...)
```

Arguments

oldfts	Financial time series object
--------	------------------------------

Description

`newfts = todayly(oldfts)` converts a financial time series of any frequency to a daily frequency. `todayly` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as '00:00' for those days that did not previously exist in `oldfts`.

Empty (`[]`) passed as inputs for parameter pair values for `todayly` will trigger the use of the defaults.

`newfts = todayly(oldfts, 'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

Parameter Name	Parameter Value	Description
CalcMethod	Exact	Returns the value located at specific dates/times. No data manipulation occurs.
	v21x	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns a five-day business week that starts on Monday and ends on Friday.

Parameter Name	Parameter Value	Description
Note If you set <code>CalcMethod</code> to <code>v21x</code> , settings for all of the following parameter name/parameter value pairs are not supported.		
<code>BusDays</code>	0	Generates a financial time series that ranges from (or between) the first date to the last date in <code>oldfts</code> (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a daily financial time series that ranges from the first date to the last date in <code>oldfts</code> (excludes NYSE nonbusiness days and holidays and weekends based on <code>Altholidays</code> and <code>Weekend</code>). NYSE market closures, holidays, and weekends are observed if <code>Altholidays</code> and <code>Weekend</code> are not supplied or empty (<code>[]</code>).
<code>DateFilter</code>	<code>Absolute</code>	(Default) Displays all daily dates between the start and end dates of <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
Note: The default is to create a time series with every date at the specified periodicity, which is with <code>DateFilter = Absolute</code> . If you use <code>DateFilter = Relative</code> , the endpoint effects do not apply since only your data defines which dates will appear in the output time series object.		
<code>TimeSpec</code>	<code>Relative</code>	Displays only dates that exist in <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	<code>First</code>	Returns only the observation that occurs at the first (earliest) time for a specific date.
	<code>Last</code>	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
<code>Altholidays</code>		Vector of dates specifying an alternate set of market closure dates.
	-1	Excludes all holidays.

Parameter Name	Parameter Value	Description
Weekend		Vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then Weekend = [1 0 0 0 0 0 1].

Examples

Transform Time Series Object from Weekly to Daily Values

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =
```

```
desc: Index
freq: Weekly (2)

'dates: (53)'      'Metric: (53)'
'01-Jan-1999'    [    97.8872]
'08-Jan-1999'    [    97.0847]
'15-Jan-1999'    [   109.6312]
'22-Jan-1999'    [   105.5743]
'29-Jan-1999'    [   108.4028]
'05-Feb-1999'    [   134.4882]
'12-Feb-1999'    [   117.5581]
'19-Feb-1999'    [   106.6683]
'26-Feb-1999'    [   118.2912]
'05-Mar-1999'    [   105.6835]
'12-Mar-1999'    [   128.5836]
'19-Mar-1999'    [   115.1746]
'26-Mar-1999'    [   131.2854]
'02-Apr-1999'    [   130.7116]
'09-Apr-1999'    [   123.1684]
'16-Apr-1999'    [   107.2975]
'23-Apr-1999'    [    91.5625]
'30-Apr-1999'    [    78.5738]
```

'07-May-1999'	[65.2904]
'14-May-1999'	[70.8581]
'21-May-1999'	[72.4807]
'28-May-1999'	[72.9190]
'04-Jun-1999'	[64.3460]
'11-Jun-1999'	[59.8743]
'18-Jun-1999'	[55.0026]
'25-Jun-1999'	[49.4032]
'02-Jul-1999'	[49.9485]
'09-Jul-1999'	[47.8061]
'16-Jul-1999'	[61.0517]
'23-Jul-1999'	[58.9313]
'30-Jul-1999'	[53.9584]
'06-Aug-1999'	[44.8472]
'13-Aug-1999'	[45.0463]
'20-Aug-1999'	[45.1088]
'27-Aug-1999'	[56.4897]
'03-Sep-1999'	[61.2449]
'10-Sep-1999'	[58.1012]
'17-Sep-1999'	[50.8974]
'24-Sep-1999'	[46.5143]
'01-Oct-1999'	[38.0806]
'08-Oct-1999'	[33.6664]
'15-Oct-1999'	[34.2992]
'22-Oct-1999'	[33.4202]
'29-Oct-1999'	[36.9287]
'05-Nov-1999'	[35.1278]
'12-Nov-1999'	[41.8128]
'19-Nov-1999'	[35.8199]
'26-Nov-1999'	[36.9495]
'03-Dec-1999'	[36.2880]
'10-Dec-1999'	[33.8457]
'17-Dec-1999'	[33.3868]
'24-Dec-1999'	[32.7737]
'31-Dec-1999'	[28.5665]

Use `todayly` to obtain the daily aggregate for the `x0` times series.

```
x1 = todayly(x0)
```

```
x1 =
```

```
desc: TODAYLY: Index
```


freq: Daily (1)

'dates: (252)'	'Metric: (252)'
'04-Jan-1999'	[NaN]
'05-Jan-1999'	[NaN]
'06-Jan-1999'	[NaN]
'07-Jan-1999'	[NaN]
'08-Jan-1999'	[97.0847]
'11-Jan-1999'	[NaN]
'12-Jan-1999'	[NaN]
'13-Jan-1999'	[NaN]
'14-Jan-1999'	[NaN]
'15-Jan-1999'	[109.6312]
'19-Jan-1999'	[NaN]
'20-Jan-1999'	[NaN]
'21-Jan-1999'	[NaN]
'22-Jan-1999'	[105.5743]
'25-Jan-1999'	[NaN]
'26-Jan-1999'	[NaN]
'27-Jan-1999'	[NaN]
'28-Jan-1999'	[NaN]
'29-Jan-1999'	[108.4028]
'01-Feb-1999'	[NaN]
'02-Feb-1999'	[NaN]
'03-Feb-1999'	[NaN]
'04-Feb-1999'	[NaN]
'05-Feb-1999'	[134.4882]
'08-Feb-1999'	[NaN]
'09-Feb-1999'	[NaN]
'10-Feb-1999'	[NaN]
'11-Feb-1999'	[NaN]
'12-Feb-1999'	[117.5581]
'16-Feb-1999'	[NaN]
'17-Feb-1999'	[NaN]
'18-Feb-1999'	[NaN]
'19-Feb-1999'	[106.6683]
'22-Feb-1999'	[NaN]
'23-Feb-1999'	[NaN]
'24-Feb-1999'	[NaN]
'25-Feb-1999'	[NaN]
'26-Feb-1999'	[118.2912]
'01-Mar-1999'	[NaN]
'02-Mar-1999'	[NaN]
'03-Mar-1999'	[NaN]

'04-Mar-1999'	[NaN]
'05-Mar-1999'	[105.6835]
'08-Mar-1999'	[NaN]
'09-Mar-1999'	[NaN]
'10-Mar-1999'	[NaN]
'11-Mar-1999'	[NaN]
'12-Mar-1999'	[128.5836]
'15-Mar-1999'	[NaN]
'16-Mar-1999'	[NaN]
'17-Mar-1999'	[NaN]
'18-Mar-1999'	[NaN]
'19-Mar-1999'	[115.1746]
'22-Mar-1999'	[NaN]
'23-Mar-1999'	[NaN]
'24-Mar-1999'	[NaN]
'25-Mar-1999'	[NaN]
'26-Mar-1999'	[131.2854]
'29-Mar-1999'	[NaN]
'30-Mar-1999'	[NaN]
'31-Mar-1999'	[NaN]
'01-Apr-1999'	[NaN]
'05-Apr-1999'	[NaN]
'06-Apr-1999'	[NaN]
'07-Apr-1999'	[NaN]
'08-Apr-1999'	[NaN]
'09-Apr-1999'	[123.1684]
'12-Apr-1999'	[NaN]
'13-Apr-1999'	[NaN]
'14-Apr-1999'	[NaN]
'15-Apr-1999'	[NaN]
'16-Apr-1999'	[107.2975]
'19-Apr-1999'	[NaN]
'20-Apr-1999'	[NaN]
'21-Apr-1999'	[NaN]
'22-Apr-1999'	[NaN]
'23-Apr-1999'	[91.5625]
'26-Apr-1999'	[NaN]
'27-Apr-1999'	[NaN]
'28-Apr-1999'	[NaN]
'29-Apr-1999'	[NaN]
'30-Apr-1999'	[78.5738]
'03-May-1999'	[NaN]
'04-May-1999'	[NaN]
'05-May-1999'	[NaN]

'06-May-1999'	[NaN]
'07-May-1999'	[65.2904]
'10-May-1999'	[NaN]
'11-May-1999'	[NaN]
'12-May-1999'	[NaN]
'13-May-1999'	[NaN]
'14-May-1999'	[70.8581]
'17-May-1999'	[NaN]
'18-May-1999'	[NaN]
'19-May-1999'	[NaN]
'20-May-1999'	[NaN]
'21-May-1999'	[72.4807]
'24-May-1999'	[NaN]
'25-May-1999'	[NaN]
'26-May-1999'	[NaN]
'27-May-1999'	[NaN]
'28-May-1999'	[72.9190]
'01-Jun-1999'	[NaN]
'02-Jun-1999'	[NaN]
'03-Jun-1999'	[NaN]
'04-Jun-1999'	[64.3460]
'07-Jun-1999'	[NaN]
'08-Jun-1999'	[NaN]
'09-Jun-1999'	[NaN]
'10-Jun-1999'	[NaN]
'11-Jun-1999'	[59.8743]
'14-Jun-1999'	[NaN]
'15-Jun-1999'	[NaN]
'16-Jun-1999'	[NaN]
'17-Jun-1999'	[NaN]
'18-Jun-1999'	[55.0026]
'21-Jun-1999'	[NaN]
'22-Jun-1999'	[NaN]
'23-Jun-1999'	[NaN]
'24-Jun-1999'	[NaN]
'25-Jun-1999'	[49.4032]
'28-Jun-1999'	[NaN]
'29-Jun-1999'	[NaN]
'30-Jun-1999'	[NaN]
'01-Jul-1999'	[NaN]
'02-Jul-1999'	[49.9485]
'06-Jul-1999'	[NaN]
'07-Jul-1999'	[NaN]
'08-Jul-1999'	[NaN]

'09-Jul-1999'	[47.8061]
'12-Jul-1999'	[NaN]
'13-Jul-1999'	[NaN]
'14-Jul-1999'	[NaN]
'15-Jul-1999'	[NaN]
'16-Jul-1999'	[61.0517]
'19-Jul-1999'	[NaN]
'20-Jul-1999'	[NaN]
'21-Jul-1999'	[NaN]
'22-Jul-1999'	[NaN]
'23-Jul-1999'	[58.9313]
'26-Jul-1999'	[NaN]
'27-Jul-1999'	[NaN]
'28-Jul-1999'	[NaN]
'29-Jul-1999'	[NaN]
'30-Jul-1999'	[53.9584]
'02-Aug-1999'	[NaN]
'03-Aug-1999'	[NaN]
'04-Aug-1999'	[NaN]
'05-Aug-1999'	[NaN]
'06-Aug-1999'	[44.8472]
'09-Aug-1999'	[NaN]
'10-Aug-1999'	[NaN]
'11-Aug-1999'	[NaN]
'12-Aug-1999'	[NaN]
'13-Aug-1999'	[45.0463]
'16-Aug-1999'	[NaN]
'17-Aug-1999'	[NaN]
'18-Aug-1999'	[NaN]
'19-Aug-1999'	[NaN]
'20-Aug-1999'	[45.1088]
'23-Aug-1999'	[NaN]
'24-Aug-1999'	[NaN]
'25-Aug-1999'	[NaN]
'26-Aug-1999'	[NaN]
'27-Aug-1999'	[56.4897]
'30-Aug-1999'	[NaN]
'31-Aug-1999'	[NaN]
'01-Sep-1999'	[NaN]
'02-Sep-1999'	[NaN]
'03-Sep-1999'	[61.2449]
'07-Sep-1999'	[NaN]
'08-Sep-1999'	[NaN]
'09-Sep-1999'	[NaN]

'10-Sep-1999'	[58.1012]
'13-Sep-1999'	[NaN]
'14-Sep-1999'	[NaN]
'15-Sep-1999'	[NaN]
'16-Sep-1999'	[NaN]
'17-Sep-1999'	[50.8974]
'20-Sep-1999'	[NaN]
'21-Sep-1999'	[NaN]
'22-Sep-1999'	[NaN]
'23-Sep-1999'	[NaN]
'24-Sep-1999'	[46.5143]
'27-Sep-1999'	[NaN]
'28-Sep-1999'	[NaN]
'29-Sep-1999'	[NaN]
'30-Sep-1999'	[NaN]
'01-Oct-1999'	[38.0806]
'04-Oct-1999'	[NaN]
'05-Oct-1999'	[NaN]
'06-Oct-1999'	[NaN]
'07-Oct-1999'	[NaN]
'08-Oct-1999'	[33.6664]
'11-Oct-1999'	[NaN]
'12-Oct-1999'	[NaN]
'13-Oct-1999'	[NaN]
'14-Oct-1999'	[NaN]
'15-Oct-1999'	[34.2992]
'18-Oct-1999'	[NaN]
'19-Oct-1999'	[NaN]
'20-Oct-1999'	[NaN]
'21-Oct-1999'	[NaN]
'22-Oct-1999'	[33.4202]
'25-Oct-1999'	[NaN]
'26-Oct-1999'	[NaN]
'27-Oct-1999'	[NaN]
'28-Oct-1999'	[NaN]
'29-Oct-1999'	[36.9287]
'01-Nov-1999'	[NaN]
'02-Nov-1999'	[NaN]
'03-Nov-1999'	[NaN]
'04-Nov-1999'	[NaN]
'05-Nov-1999'	[35.1278]
'08-Nov-1999'	[NaN]
'09-Nov-1999'	[NaN]
'10-Nov-1999'	[NaN]

'11-Nov-1999'	[NaN]
'12-Nov-1999'	[41.8128]
'15-Nov-1999'	[NaN]
'16-Nov-1999'	[NaN]
'17-Nov-1999'	[NaN]
'18-Nov-1999'	[NaN]
'19-Nov-1999'	[35.8199]
'22-Nov-1999'	[NaN]
'23-Nov-1999'	[NaN]
'24-Nov-1999'	[NaN]
'26-Nov-1999'	[36.9495]
'29-Nov-1999'	[NaN]
'30-Nov-1999'	[NaN]
'01-Dec-1999'	[NaN]
'02-Dec-1999'	[NaN]
'03-Dec-1999'	[36.2880]
'06-Dec-1999'	[NaN]
'07-Dec-1999'	[NaN]
'08-Dec-1999'	[NaN]
'09-Dec-1999'	[NaN]
'10-Dec-1999'	[33.8457]
'13-Dec-1999'	[NaN]
'14-Dec-1999'	[NaN]
'15-Dec-1999'	[NaN]
'16-Dec-1999'	[NaN]
'17-Dec-1999'	[33.3868]
'20-Dec-1999'	[NaN]
'21-Dec-1999'	[NaN]
'22-Dec-1999'	[NaN]
'23-Dec-1999'	[NaN]
'27-Dec-1999'	[NaN]
'28-Dec-1999'	[NaN]
'29-Dec-1999'	[NaN]
'30-Dec-1999'	[NaN]
'31-Dec-1999'	[28.5665]

See Also

`convertto` | `fints` | `toannual` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly`

today

Current date

Syntax

```
Datenum = today
```

Description

`Datenum = today` returns the current date as a serial date number.

Examples

```
Datenum = today
```

returns

```
Datenum =
```

```
    730695
```

See Also

`datenum` | `now` | `datestr`

todecimal

Fractional to decimal conversion

Syntax

```
usddec = todecimal(quote, fracpart)
```

Description

`usddec = todecimal(quote, fracpart)` returns the decimal equivalent, `usddec`, of a security whose price is normally quoted as a whole number and a fraction (`quote`). `fracpart` indicates the fractional base (denominator) with which the security is normally quoted (default = 32).

Examples

In the *Wall Street Journal*, bond prices are quoted in fractional form based on a denominator of 32. For example, if you see the quoted price is 100:05 it means 100 $\frac{5}{32}$. To find the equivalent decimal value, enter

```
usddec = todecimal(100.05)
```

```
usddec =  
    100.1563
```

```
usddec = todecimal(97.04, 16)
```

```
usddec =  
    97.2500
```

Note The convention of using . (period) as a substitute for : (colon) in the input is adopted from Excel software.

See Also

toquoted

tomonthly

Convert to monthly

Syntax

```
newfts = tomonthly(oldfts)
```

```
newfts = tomonthly(oldfts, 'ParameterName', ParameterValue, ...)
```

Arguments

oldfts	Financial time series object.
--------	-------------------------------

Description

`newfts = tomonthly(oldfts)` converts a financial time series of any frequency to a monthly frequency. The default end-of-month day is the last business day of the month. `tomonthly` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as 00:00 for those days that did not previously exist in `oldfts`.

Empty (`[]`) passed as inputs for parameter pair values for `tomonthly` will trigger the use of the defaults.

`newfts = tomonthly(oldfts, 'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values within each month. Data for missing dates are given the value 0.
	Exact	Returns the exact value at the end-of-month date. No data manipulation occurs.

Parameter Name	Parameter Value	Description
	Nearest	(Default) Returns the values located at the end-of-month date. If there is missing data, 'Nearest' returns the nearest data point preceding the end-of-month date.
	SimpAvg	Returns an averaged monthly value that only takes into account dates with data (nonNaN) within each month.
	v21x	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns an averaged end-of-month value using a previous <code>tomonthly</code> algorithm. This algorithm takes into account all dates and data. For dates that do not contain any data, the data is assumed to be 0.
<p>Note If you set <code>CalcMethod</code> to <code>v21x</code>, settings for all of the following parameter name/parameter value pairs are not supported.</p>		
BusDays	0	Generates a monthly financial time series that ranges from the first date to the last date in <code>oldfts</code> (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a monthly financial time series that ranges from the first date to the last date in <code>oldfts</code> (excludes NYSE nonbusiness days and holidays and weekends based on <code>AltHolidays</code> and <code>Weekend</code>). If an end-of-month date falls on a nonbusiness day or NYSE holiday, returns the last business day of the month. NYSE market closures, holidays, and weekends are observed if <code>AltHolidays</code> and <code>Weekend</code> are not supplied or empty (<code>[]</code>).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all monthly dates between the start and end dates of <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	Relative	<p>Note: The default is to create a time series with every date at the specified periodicity, which is with <code>DateFilter = Absolute</code>. If you use <code>DateFilter = Relative</code>, the endpoint effects do not apply since only your data defines which dates will appear in the output time series object.</p> <p>Returns only monthly dates that exist in <code>oldfts</code>. Some dates may be disregarded if <code>BusDays = 1</code>.</p>
ED	0	(Default) The end-of-month date is the last day (or last business day) of the month.
	1 - 31	<p>Returns values on the specified end-of-month day. Months that do not contain the specified end-of-month day return the last day of the month instead (for example, <code>ED = 31</code> does not exist for February).</p> <p>If end-of-month falls on a NYSE non-business day or holiday, the previous business day is returned if <code>BusDays = 1</code>.</p>

Parameter Name	Parameter Value	Description
EndPtTo1	[Begin, End]	<p>Denotes the minimum number of days that constitute an odd month at the end points of the time series (before the first whole period and after the last whole period).</p> <p>Begin and End must be -1 or any positive integer greater than or equal to 0.</p> <p>A single value input for EndPtTo1 is the same as specifying that single value for Begin and End.</p> <p>-1 Do not include odd month dates and data in calculations.</p> <p>0 (Default) Include all odd month dates and data in calculations.</p> <p>n Number of days that constitute an odd month. If the minimum number of days is not met, the odd month dates and data are ignored.</p>
<p>The following diagram is a general depiction of the factors involved in the determination of end points for this function.</p>		
TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.
	Last	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.

Parameter Name	Parameter Value	Description
AltHolidays		Vector of dates specifying an alternate set of market closure dates.
	-1	Excludes all holidays.
Weekend		Vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then Weekend = [1 0 0 0 0 0 1].

Examples

Transform Time Series Object from Weekly to Monthly Values

This example shows how to transform a time series object from weekly to monthly values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =
```

```
desc: Index
freq: Weekly (2)

'dates: (53) '      'Metric: (53) '
'01-Jan-1999'      [      97.8872]
'08-Jan-1999'      [      97.0847]
'15-Jan-1999'      [     109.6312]
'22-Jan-1999'      [     105.5743]
'29-Jan-1999'      [     108.4028]
'05-Feb-1999'      [     134.4882]
'12-Feb-1999'      [     117.5581]
'19-Feb-1999'      [     106.6683]
'26-Feb-1999'      [     118.2912]
'05-Mar-1999'      [     105.6835]
'12-Mar-1999'      [     128.5836]
```

'19-Mar-1999'	[115.1746]
'26-Mar-1999'	[131.2854]
'02-Apr-1999'	[130.7116]
'09-Apr-1999'	[123.1684]
'16-Apr-1999'	[107.2975]
'23-Apr-1999'	[91.5625]
'30-Apr-1999'	[78.5738]
'07-May-1999'	[65.2904]
'14-May-1999'	[70.8581]
'21-May-1999'	[72.4807]
'28-May-1999'	[72.9190]
'04-Jun-1999'	[64.3460]
'11-Jun-1999'	[59.8743]
'18-Jun-1999'	[55.0026]
'25-Jun-1999'	[49.4032]
'02-Jul-1999'	[49.9485]
'09-Jul-1999'	[47.8061]
'16-Jul-1999'	[61.0517]
'23-Jul-1999'	[58.9313]
'30-Jul-1999'	[53.9584]
'06-Aug-1999'	[44.8472]
'13-Aug-1999'	[45.0463]
'20-Aug-1999'	[45.1088]
'27-Aug-1999'	[56.4897]
'03-Sep-1999'	[61.2449]
'10-Sep-1999'	[58.1012]
'17-Sep-1999'	[50.8974]
'24-Sep-1999'	[46.5143]
'01-Oct-1999'	[38.0806]
'08-Oct-1999'	[33.6664]
'15-Oct-1999'	[34.2992]
'22-Oct-1999'	[33.4202]
'29-Oct-1999'	[36.9287]
'05-Nov-1999'	[35.1278]
'12-Nov-1999'	[41.8128]
'19-Nov-1999'	[35.8199]
'26-Nov-1999'	[36.9495]
'03-Dec-1999'	[36.2880]
'10-Dec-1999'	[33.8457]
'17-Dec-1999'	[33.3868]
'24-Dec-1999'	[32.7737]
'31-Dec-1999'	[28.5665]

Use `tomonthly` to obtain the monthly aggregate for the `x0` times series.

```
x1 = tomonthly(x0)
```

```
x1 =
```

```
desc: TOMONTHLY: Index  
freq: Monthly (3)
```

```
'dates: (12)'      'Metric: (12)'  
'29-Jan-1999'     [ 108.4028]  
'26-Feb-1999'     [ 118.2912]  
'31-Mar-1999'     [ 131.2854]  
'30-Apr-1999'     [ 78.5738]  
'28-May-1999'     [ 72.9190]  
'30-Jun-1999'     [ 49.4032]  
'30-Jul-1999'     [ 53.9584]  
'31-Aug-1999'     [ 56.4897]  
'30-Sep-1999'     [ 46.5143]  
'29-Oct-1999'     [ 36.9287]  
'30-Nov-1999'     [ 36.9495]  
'31-Dec-1999'     [ 28.5665]
```

See Also

`convertto` | `fints` | `toannual` | `todayly` | `toquarterly` | `tosemi` | `toweekly`

toquarterly

Convert to quarterly

Syntax

```
newfts = toquarterly(oldfts)
```

```
newfts = toquarterly(oldfts, 'ParameterName', ParameterValue, ...)
```

Arguments

oldfts	Financial time series object
--------	------------------------------

Description

`newfts = toquarterly(oldfts)` converts a financial time series of any frequency to a quarterly frequency. The default quarterly days are the last business day of March, June, September, and December. `toquarterly` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as 00:00 for those days that did not previously exist in `oldfts`.

Empty (`[]`) passed as inputs for parameter pair values for `toquarterly` will trigger the use of the defaults.

`newfts = toquarterly(oldfts, 'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

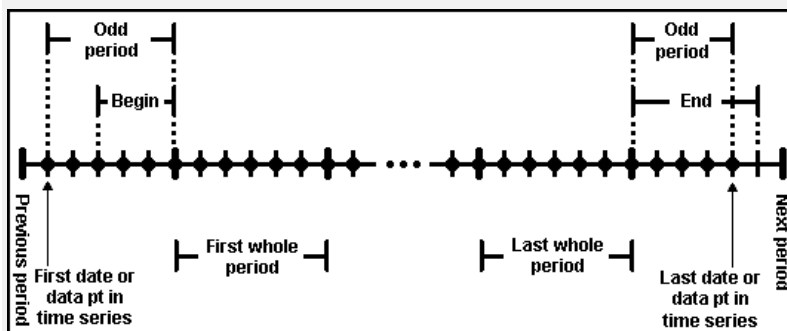
Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values between each quarter. Data for missing dates are given the value 0.

Parameter Name	Parameter Value	Description
	Exact	Returns the exact value at the end-of-quarter date. No data manipulation occurs.
	Nearest	(Default) Returns the values located at the end-of-quarter date. If there is missing data, Nearest returns the nearest data point preceding the end-of-quarter date.
	SimpAvg	Returns an averaged quarterly value that only takes into account dates with data (nonNaN) within each quarter.
	v21x	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns an averaged end-of-quarter value using a previous toquarterly algorithm. This algorithm takes into account all dates and data. For dates that do not contain any data, the data is assumed to be 0.
<p>Note If you set CalcMethod to v21x, settings for all of the following parameter name/parameter value pairs are not supported.</p>		
BusDays	0	Generates a financial time series that ranges from (or between) the first date to the last date in oldfts (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a financial time series that ranges from the first date to the last date in oldfts (excludes NYSE nonbusiness days and holidays and weekends based on AltHolidays and Weekend). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ({}).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all quarterly dates between the start and end dates of <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	Relative	Note: The default is to create a time series with every date at the specified periodicity, which is with <code>DateFilter = Absolute</code> . If you use <code>DateFilter = Relative</code> , the endpoint effects do not apply since only your data defines which dates will appear in the output time series object.
ED	0	Returns only quarterly dates that exist in <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	1 - 31	(Default) The end-of-quarter date is the last day (or last business day) of the quarter. Specifies a particular end-of-quarter day. Months that do not contain the specified end-of-quarter day return the last day of the quarter instead (for example, <code>ED = 31</code> does not exist for February).
EM	1 - 12	Last month of the first quarter. All subsequent quarterly dates are based on this month. The default end-of-first-quarter month is March (3).

Parameter Name	Parameter Value	Description
EndPtTo1	[Begin, End]	<p>Denotes the minimum number of days that constitute a odd quarter at the endpoints of the time series (before the first whole period and after the last whole period).</p> <p>Begin and End must be -1 or any positive integer greater than or equal to 0.</p> <p>A single value input for EndPtTo1 is the same as specifying that single value for Begin and End.</p> <p>-1 Do not include odd quarter dates and data in calculations.</p> <p>0 (Default) Include all odd quarter dates and data in calculations.</p> <p>n Number of days (any positive integer) that constitute an odd quarter. If there are insufficient days for a complete quarter, the odd quarter dates and data are ignored.</p>

The following diagram is a general depiction of the factors involved in the determination of endpoints for this function.



TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.
----------	-------	--

Parameter Name	Parameter Value	Description
	Last	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
AltHolidays		Vector of dates specifying an alternate set of market closure dates.
	-1	Excludes all holidays.
Weekend		Vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then Weekend = [1 0 0 0 0 0 1].

Examples

Transform Time Series Object from Weekly to Quarterly Values

This example shows how to transform a time series object from weekly to quarterly values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =
```

```
desc: Index
freq: Weekly (2)

'dates: (53)'      'Metric: (53)'
'01-Jan-1999'    [      97.8872]
'08-Jan-1999'    [      97.0847]
'15-Jan-1999'    [     109.6312]
'22-Jan-1999'    [     105.5743]
'29-Jan-1999'    [     108.4028]
'05-Feb-1999'    [     134.4882]
'12-Feb-1999'    [     117.5581]
'19-Feb-1999'    [     106.6683]
```

'26-Feb-1999'	[118.2912]
'05-Mar-1999'	[105.6835]
'12-Mar-1999'	[128.5836]
'19-Mar-1999'	[115.1746]
'26-Mar-1999'	[131.2854]
'02-Apr-1999'	[130.7116]
'09-Apr-1999'	[123.1684]
'16-Apr-1999'	[107.2975]
'23-Apr-1999'	[91.5625]
'30-Apr-1999'	[78.5738]
'07-May-1999'	[65.2904]
'14-May-1999'	[70.8581]
'21-May-1999'	[72.4807]
'28-May-1999'	[72.9190]
'04-Jun-1999'	[64.3460]
'11-Jun-1999'	[59.8743]
'18-Jun-1999'	[55.0026]
'25-Jun-1999'	[49.4032]
'02-Jul-1999'	[49.9485]
'09-Jul-1999'	[47.8061]
'16-Jul-1999'	[61.0517]
'23-Jul-1999'	[58.9313]
'30-Jul-1999'	[53.9584]
'06-Aug-1999'	[44.8472]
'13-Aug-1999'	[45.0463]
'20-Aug-1999'	[45.1088]
'27-Aug-1999'	[56.4897]
'03-Sep-1999'	[61.2449]
'10-Sep-1999'	[58.1012]
'17-Sep-1999'	[50.8974]
'24-Sep-1999'	[46.5143]
'01-Oct-1999'	[38.0806]
'08-Oct-1999'	[33.6664]
'15-Oct-1999'	[34.2992]
'22-Oct-1999'	[33.4202]
'29-Oct-1999'	[36.9287]
'05-Nov-1999'	[35.1278]
'12-Nov-1999'	[41.8128]
'19-Nov-1999'	[35.8199]
'26-Nov-1999'	[36.9495]
'03-Dec-1999'	[36.2880]
'10-Dec-1999'	[33.8457]
'17-Dec-1999'	[33.3868]
'24-Dec-1999'	[32.7737]

```
'31-Dec-1999' [ 28.5665]
```

Use `toquarterly` to obtain the quarterly aggregate for the `x0` times series.

```
x1 = toquarterly(x0)
```

```
x1 =
```

```
desc: TOQUARTERLY: Index  
freq: Quarterly (4)
```

```
'dates: (4)' 'Metric: (4)'  
'31-Mar-1999' [ 131.2854]  
'30-Jun-1999' [ 49.4032]  
'30-Sep-1999' [ 46.5143]  
'31-Dec-1999' [ 28.5665]
```

See Also

`convertto` | `fints` | `toannual` | `todayly` | `tomonthly` | `tosemi` | `toweekly`

toquoted

Decimal to fractional conversion

Syntax

```
quote = toquoted(usddec, fracpart)
```

Description

`quote = toquoted(usddec, fracpart)` returns the fractional equivalent, `quote`, of the decimal figure, `usddec`, based on the fractional base (denominator), `fracpart`. The fractional bases are the ones used for quoting equity prices in the United States (denominator 2, 4, 8, 16, or 32). If `fracpart` is not entered, the denominator 32 is assumed.

Examples

A United States equity price in decimal form is 101.625. To convert this to fractional form in eighths of a dollar:

```
quote = toquoted(101.625, 8)
```

```
quote =  
101.05
```

The answer is interpreted as 101 5/8.

Note The convention of using . (period) as a substitute for : (colon) in the output is adopted from Excel software.

See Also

todecimal

tosemi

Convert to semiannual

Syntax

```
newfts = tosemi(oldfts)
```

```
newfts = tosemi(oldfts, 'ParameterName', ParameterValue, ...)
```

Arguments

oldfts	Financial time series object.
--------	-------------------------------

Description

`newfts = tosemi(oldfts)` converts a financial time series of any frequency to a semiannual frequency. The default semiannual days are the last business day of June and December. `tosemi` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as 00:00 for those days that did not previously exist in `oldfts`.

Empty (`[]`) passed as inputs for parameter pair values for `tosemi` will trigger the use of the defaults.

`newfts = tosemi(oldfts, 'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

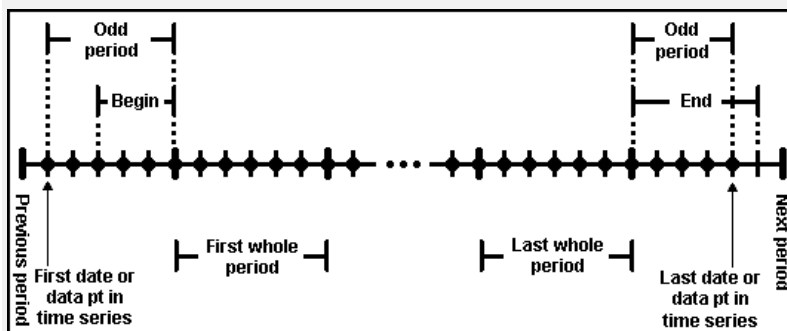
Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values within each semiannual period. Data for missing dates are given the value 0.
	Exact	Returns the exact value at the end-of-period date. No data manipulation occurs.

Parameter Name	Parameter Value	Description
	Nearest	(Default) Returns the values located at the end-of-period date. If there is missing data, Nearest returns the nearest data point preceding the end-of-period date.
	SimpAvg	Returns an averaged semiannual value that only takes into account dates with data (nonNaN) within each semiannual period.
	v21x	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns an averaged end-of-period value using a previous tosemi algorithm. This algorithm takes into account all dates and data. For dates that do not contain any data, the data is assumed to be 0.
<hr/> <p>Note If you set CalcMethod to v21x, settings for all of the following parameter name/parameter value pairs are not supported.</p> <hr/>		
BusDays	0	Generates a financial time series that ranges from (or between) the first date to the last date in oldfts (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a financial time series that ranges from the first date to the last date in oldfts (excludes NYSE nonbusiness days and holidays and weekends based on AltHolidays and Weekend). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ({}).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all semiannual dates between the start and end dates of <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	Relative	Note: The default is to create a time series with every date at the specified periodicity, which is with <code>DateFilter = Absolute</code> . If you use <code>DateFilter = Relative</code> , the endpoint effects do not apply since only your data defines which dates will appear in the output time series object.
ED	0	Returns only semiannual dates that exist in <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
	1 - 31	(Default) The end-of-period date is the last day (or last business day) of the semiannual period. Specifies a particular end-of-period day. Months that do not contain the specified end-of-period day return the last day of the semiannual period instead (for example, <code>ED = 31</code> does not exist for February).
EM	1 - 12	End month of the first semiannual period. All subsequent period dates are based on this month. The default end of period months are June (6) and December (12).

Parameter Name	Parameter Value	Description
EndPtTol	[Begin, End]	<p>Denotes the minimum number of days that constitute an odd semiannual period at the endpoints of the time series (before the first whole period and after the last whole period).</p> <p>Begin and End must be -1 or any positive integer greater than or equal to 0.</p> <p>A single value input for EndPtTol is the same as specifying that single value for Begin and End.</p> <p>-1 Do not include odd period dates and data in calculations.</p> <p>0 (Default) Include all odd period dates and data in calculations.</p> <p>n Number of days (any positive integer) that constitute an odd period. If there are insufficient days for a complete semiannual period, the odd period dates and data are ignored.</p>

The following diagram is a general depiction of the factors involved in the determination of endpoints for this function.



TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.
----------	-------	--

Parameter Name	Parameter Value	Description
	Last	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
AltHolidays		Vector of dates specifying an alternate set of market closure dates.
	-1	Excludes all holidays.
Weekend		Vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then Weekend = [1 0 0 0 0 0 1].

Examples

Transform Time Series Object from Weekly to Semiannual Values

This example shows how to transform a time series object from weekly to semiannual values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =
```

```
desc: Index
freq: Weekly (2)

'dates: (53)'      'Metric: (53)'
'01-Jan-1999'     [      97.8872]
'08-Jan-1999'     [      97.0847]
'15-Jan-1999'     [     109.6312]
'22-Jan-1999'     [     105.5743]
'29-Jan-1999'     [     108.4028]
'05-Feb-1999'     [     134.4882]
'12-Feb-1999'     [     117.5581]
'19-Feb-1999'     [     106.6683]
```

'26-Feb-1999'	[118.2912]
'05-Mar-1999'	[105.6835]
'12-Mar-1999'	[128.5836]
'19-Mar-1999'	[115.1746]
'26-Mar-1999'	[131.2854]
'02-Apr-1999'	[130.7116]
'09-Apr-1999'	[123.1684]
'16-Apr-1999'	[107.2975]
'23-Apr-1999'	[91.5625]
'30-Apr-1999'	[78.5738]
'07-May-1999'	[65.2904]
'14-May-1999'	[70.8581]
'21-May-1999'	[72.4807]
'28-May-1999'	[72.9190]
'04-Jun-1999'	[64.3460]
'11-Jun-1999'	[59.8743]
'18-Jun-1999'	[55.0026]
'25-Jun-1999'	[49.4032]
'02-Jul-1999'	[49.9485]
'09-Jul-1999'	[47.8061]
'16-Jul-1999'	[61.0517]
'23-Jul-1999'	[58.9313]
'30-Jul-1999'	[53.9584]
'06-Aug-1999'	[44.8472]
'13-Aug-1999'	[45.0463]
'20-Aug-1999'	[45.1088]
'27-Aug-1999'	[56.4897]
'03-Sep-1999'	[61.2449]
'10-Sep-1999'	[58.1012]
'17-Sep-1999'	[50.8974]
'24-Sep-1999'	[46.5143]
'01-Oct-1999'	[38.0806]
'08-Oct-1999'	[33.6664]
'15-Oct-1999'	[34.2992]
'22-Oct-1999'	[33.4202]
'29-Oct-1999'	[36.9287]
'05-Nov-1999'	[35.1278]
'12-Nov-1999'	[41.8128]
'19-Nov-1999'	[35.8199]
'26-Nov-1999'	[36.9495]
'03-Dec-1999'	[36.2880]
'10-Dec-1999'	[33.8457]
'17-Dec-1999'	[33.3868]
'24-Dec-1999'	[32.7737]

```
'31-Dec-1999' [ 28.5665]
```

Use `tosemi` to obtain the semiannual aggregate for the `x0` times series.

```
x1 = tosemi(x0)
```

```
x1 =
```

```
desc: TOSEMI: Index  
freq: Semiannual (5)  
  
'dates: (2)' 'Metric: (2)'  
'30-Jun-1999' [ 49.4032]  
'31-Dec-1999' [ 28.5665]
```

See Also

`convertto` | `fints` | `toannual` | `todayly` | `tomonthly` | `toquarterly` | `toweekly`

totalreturnprice

Total return price time series

Syntax

```
Return = totalreturnprice(Price, Action, Dividend)
```

Arguments

Price	Number of observations (NUMOBS)-by-2 matrix of price data. Column 1 contains MATLAB serial date numbers. Column 2 contains price values.
Action	NUMOBS-by-2 matrix of price data. Column 1 contains MATLAB serial date numbers. Column 2 contains split ratios.
Dividend	NUMOBS-by-2 matrix of price data. Column 1 contains MATLAB serial date numbers. Column 2 contains dividend payouts.

The number of observations (NUMOBS) for the three input arguments will differ from each other.

Description

`Return = totalreturnprice(Price, Action, Dividend)` generates a total return price time series given price data, action or split data, and dividend data.

Return is NUMOBS-by-2 array of price data, where NUMOBS reflects the number of observations of price data. Column 1 contains MATLAB serial date numbers. Column 2 contains total return price values.

See Also

`periodicreturns`

towekly

Convert to weekly

Syntax

```
newfts = towekly(oldfts)
```

```
newfts = towekly(oldfts, 'ParameterName', ParameterValue, ...)
```

Arguments

oldfts	Financial time series object.
--------	-------------------------------

Description

`newfts = towekly(oldfts)` converts a financial time series of any frequency to a weekly frequency. The default weekly days are Fridays or the last business day of the week. `towekly` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as 00:00 for those days that did not previously exist in `oldfts`.

Empty (`[]`) passed as inputs for parameter pair values for `towekly` will trigger the use of the defaults.

`newfts = towekly(oldfts, 'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

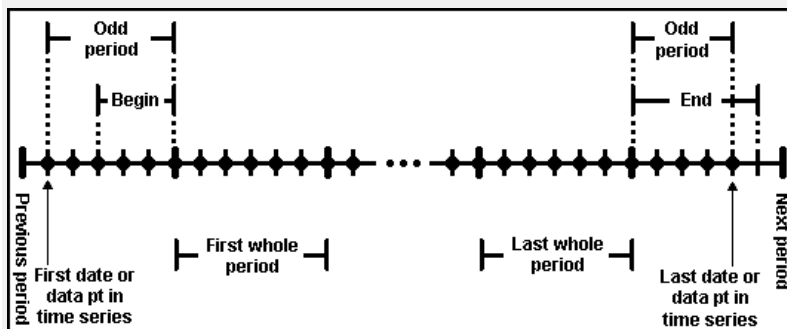
Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values within each week. Data for missing dates are given the value 0.
	Exact	Returns the exact value at the end-of-week dates. No data manipulation occurs.

Parameter Name	Parameter Value	Description
	Nearest	(Default) Returns the values located at the end-of-week dates. If there is missing data, Nearest returns the nearest data point preceding the end-of-week date.
	SimpAvg	Returns an averaged weekly value that only takes into account dates with data (nonNaN) within each week.
	v21x	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns an averaged end-of-weekly value using a previous toquarterly algorithm. This algorithm takes into account all dates and data. For dates that do not contain any data, the data is assumed to be 0.
<p>Note If you set CalcMethod to v21x, settings for all of the following parameter name/parameter value pairs are not supported.</p>		
BusDays	0	Generates a financial time series that ranges from (or between) the first date to the last date in oldfts (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a financial time series that ranges from the first date to the last date in oldfts (excludes NYSE nonbusiness days and holidays and weekends based on AltHolidays and Weekend). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ({}).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all weekly dates between the start and end dates of <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> . Note: The default is to create a time series with every date at the specified periodicity, which is with <code>DateFilter = Absolute</code> . If you use <code>DateFilter = Relative</code> , the endpoint effects do not apply since only your data defines which dates will appear in the output time series object.
	Relative	Returns only end-of-week dates that exist in <code>oldfts</code> . Some dates may be disregarded if <code>BusDays = 1</code> .
EndPtTol	[Begin, End]	Denotes the minimum number of days that constitute a odd week at the endpoints of the time series (before the first whole period and after the last whole period). Begin and End must be -1 or any positive integer greater than or equal to 0. A single value input for <code>EndPtTol</code> is the same as specifying that single value for <code>Begin</code> and <code>End</code> . -1 Do not include odd week dates and data in calculations. 0 (Default) Include all odd week dates and data in calculations. n Number of days (any positive integer) that constitute an odd week. If there are insufficient days for a complete week, the odd week dates and data are ignored.

Parameter Name	Parameter Value	Description
----------------	-----------------	-------------

The following diagram is a general depiction of the factors involved in the determination of endpoints for this function.



EOW	0 - 6	Specifies the end-of-week day: <ul style="list-style-type: none"> • 0 Friday (default) • 1 Saturday • 2 Sunday • 3 Monday • 4 Tuesday • 5 Wednesday • 6 Thursday
TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.
	Last	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
AltHolidays		Vector of dates specifying an alternate set of market closure dates.
Weekend	-1	Excludes all holidays.
		Vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then Weekend = [1 0 0 0 0 0 1].

Examples

Transform Time Series Object from Quarterly to Weekly Values

This example shows how to transform a time series object from quarterly to weekly values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a quarterly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'q', 'Index')
```

```
x0 =
```

```
desc: Index
freq: Quarterly (4)

'dates: (53)'      'Metric: (53)'
'01-Jan-1999'     [    97.8872]
'08-Jan-1999'     [    97.0847]
'15-Jan-1999'     [   109.6312]
'22-Jan-1999'     [   105.5743]
'29-Jan-1999'     [   108.4028]
'05-Feb-1999'     [   134.4882]
'12-Feb-1999'     [   117.5581]
'19-Feb-1999'     [   106.6683]
'26-Feb-1999'     [   118.2912]
'05-Mar-1999'     [   105.6835]
'12-Mar-1999'     [   128.5836]
'19-Mar-1999'     [   115.1746]
'26-Mar-1999'     [   131.2854]
'02-Apr-1999'     [   130.7116]
'09-Apr-1999'     [   123.1684]
'16-Apr-1999'     [   107.2975]
'23-Apr-1999'     [    91.5625]
'30-Apr-1999'     [    78.5738]
'07-May-1999'     [    65.2904]
'14-May-1999'     [    70.8581]
'21-May-1999'     [    72.4807]
'28-May-1999'     [    72.9190]
'04-Jun-1999'     [    64.3460]
'11-Jun-1999'     [    59.8743]
```

```

'18-Jun-1999' [ 55.0026]
'25-Jun-1999' [ 49.4032]
'02-Jul-1999' [ 49.9485]
'09-Jul-1999' [ 47.8061]
'16-Jul-1999' [ 61.0517]
'23-Jul-1999' [ 58.9313]
'30-Jul-1999' [ 53.9584]
'06-Aug-1999' [ 44.8472]
'13-Aug-1999' [ 45.0463]
'20-Aug-1999' [ 45.1088]
'27-Aug-1999' [ 56.4897]
'03-Sep-1999' [ 61.2449]
'10-Sep-1999' [ 58.1012]
'17-Sep-1999' [ 50.8974]
'24-Sep-1999' [ 46.5143]
'01-Oct-1999' [ 38.0806]
'08-Oct-1999' [ 33.6664]
'15-Oct-1999' [ 34.2992]
'22-Oct-1999' [ 33.4202]
'29-Oct-1999' [ 36.9287]
'05-Nov-1999' [ 35.1278]
'12-Nov-1999' [ 41.8128]
'19-Nov-1999' [ 35.8199]
'26-Nov-1999' [ 36.9495]
'03-Dec-1999' [ 36.2880]
'10-Dec-1999' [ 33.8457]
'17-Dec-1999' [ 33.3868]
'24-Dec-1999' [ 32.7737]
'31-Dec-1999' [ 28.5665]

```

Use `tweekly` to obtain the weekly aggregate for the `x0` times series.

```
x1 = tweekly(x0)
```

```
x1 =
```

```

desc: TOWEEKLY: Index
freq: Weekly (2)

'dates: (53)' 'Metric: (53)'
'31-Dec-1998' [ NaN]
'08-Jan-1999' [ 97.0847]
'15-Jan-1999' [ 109.6312]

```

'22-Jan-1999'	[105.5743]
'29-Jan-1999'	[108.4028]
'05-Feb-1999'	[134.4882]
'12-Feb-1999'	[117.5581]
'19-Feb-1999'	[106.6683]
'26-Feb-1999'	[118.2912]
'05-Mar-1999'	[105.6835]
'12-Mar-1999'	[128.5836]
'19-Mar-1999'	[115.1746]
'26-Mar-1999'	[131.2854]
'01-Apr-1999'	[NaN]
'09-Apr-1999'	[123.1684]
'16-Apr-1999'	[107.2975]
'23-Apr-1999'	[91.5625]
'30-Apr-1999'	[78.5738]
'07-May-1999'	[65.2904]
'14-May-1999'	[70.8581]
'21-May-1999'	[72.4807]
'28-May-1999'	[72.9190]
'04-Jun-1999'	[64.3460]
'11-Jun-1999'	[59.8743]
'18-Jun-1999'	[55.0026]
'25-Jun-1999'	[49.4032]
'02-Jul-1999'	[49.9485]
'09-Jul-1999'	[47.8061]
'16-Jul-1999'	[61.0517]
'23-Jul-1999'	[58.9313]
'30-Jul-1999'	[53.9584]
'06-Aug-1999'	[44.8472]
'13-Aug-1999'	[45.0463]
'20-Aug-1999'	[45.1088]
'27-Aug-1999'	[56.4897]
'03-Sep-1999'	[61.2449]
'10-Sep-1999'	[58.1012]
'17-Sep-1999'	[50.8974]
'24-Sep-1999'	[46.5143]
'01-Oct-1999'	[38.0806]
'08-Oct-1999'	[33.6664]
'15-Oct-1999'	[34.2992]
'22-Oct-1999'	[33.4202]
'29-Oct-1999'	[36.9287]
'05-Nov-1999'	[35.1278]
'12-Nov-1999'	[41.8128]
'19-Nov-1999'	[35.8199]

'26-Nov-1999'	[36.9495]
'03-Dec-1999'	[36.2880]
'10-Dec-1999'	[33.8457]
'17-Dec-1999'	[33.3868]
'23-Dec-1999'	[NaN]
'31-Dec-1999'	[28.5665]

See Also

`convertto` | `fints` | `toannual` | `todayly` | `tomonthly` | `toquarterly` | `tosemi`

tr2bonds

Term-structure parameters given Treasury bond parameters

Syntax

```
[Bonds, Prices, Yields] = tr2bonds(TreasuryMatrix, Settle)
```

Arguments

TreasuryMatrix	Treasury bond parameters. An n-by-5 matrix, where each row describes a Treasury bond. Columns are [CouponRate Maturity Bid Asked AskYield] where:
CouponRate	Coupon rate, as a decimal fraction.
Maturity	Maturity date, as a serial date number. Use <code>datenum</code> to convert date strings to serial date numbers.
Bid	Bid price based on \$100 face value.
Asked	Asked price based on \$100 face value.
AskYield	Asked yield to maturity, as a decimal fraction.
Settle	(Optional) Date string or serial date number of the settlement date for the analysis.

Description

[Bonds, Prices, Yields] = `tr2bonds(TreasuryMatrix, Settle)` returns term-structure parameters (bond information, prices, and yields) sorted by ascending maturity date, given Treasury bond parameters. The formats of the output matrix and vectors meet requirements for input to the `zbtprice` and `zbtyield` zero-curve bootstrapping functions.

Bonds	Coupon bond information. An n-by-6 matrix where each row describes a bond. Columns are [Maturity CouponRate Face Period Basis EndMonthRule] where:
-------	--

Maturity	Maturity date of the bond, as a serial date number. Use <code>datestr</code> to convert serial date numbers to date strings.
CouponRate	Coupon rate of the bond, as a decimal fraction.
Face	Redemption or face value of the bond, always 100.
Period	Coupons per year of the bond, always 2.
Basis	Day-count basis of the bond, possible values include: <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 For more information, see basis .
EndMonthRule	End-of-month flag, always 1, meaning that a bond's coupon payment date is always the last day of the month.
Prices	Prices. Column vector containing the price of each bond in bonds , respectively. The number of rows (n) matches the number of rows in bonds .
Yields	Yields. Column vector containing the yield to maturity of each bond in bonds , respectively. The number of rows (n) matches the number of rows in bonds . If Settle is input, Yields is computed as a semiannual yield to maturity. If Settle is not input, the quoted input yields will be used.

Examples

Return Term-Structure Parameters Given Treasury Bond Parameters

This example shows how to return term-structure parameters (bond information, prices, and yields) sorted by ascending maturity date, given Treasury bond market parameters for December 22, 1997.

```
Matrix =[0.0650 datenum('15-apr-1999') 101.03125 101.09375 0.0564
         0.05125 datenum('17-dec-1998') 99.4375 99.5 0.0563
         0.0625 datenum('30-jul-1998') 100.3125 100.375 0.0560
         0.06125 datenum('26-mar-1998') 100.09375 100.15625 0.0546];
```

```
[Bonds, Prices, Yields] = tr2bonds(Matrix)
```

```
Bonds =
```

```
1.0e+05 *
```

7.2984	0.0000	0.0010	0.0000	0	0.0000
7.2997	0.0000	0.0010	0.0000	0	0.0000
7.3011	0.0000	0.0010	0.0000	0	0.0000
7.3022	0.0000	0.0010	0.0000	0	0.0000

```
Prices =
```

```
100.1563
100.3750
99.5000
101.0938
```

```
Yields =
```

```
0.0546
0.0560
0.0563
0.0564
```

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

tbl2bond | zbtprice | zbtyield

transprob

Estimate transition probabilities from credit ratings data

Syntax

```
[transMat, sampleTotals, idTotals] = transprob(data)
[transMat, sampleTotals, idTotals] = transprob(data,
Name, Value)
```

Description

`[transMat, sampleTotals, idTotals] = transprob(data)` constructs a transition matrix from historical data of credit ratings.

`[transMat, sampleTotals, idTotals] = transprob(data, Name, Value)` constructs a transition matrix from historical data of credit ratings with additional options specified by one or more Name, Value pair arguments.

Input Arguments

data

Using `transprob` to estimate transition probabilities given credit ratings historical data (i.e. credit migration data), the `data` input can be either of the following:

- A preprocessed data structure obtained using `transprobprep`. This data structure contains the fields 'idStart', 'numericDates', 'numericRatings', and 'ratingsLabels'.

or

- An `nRecords-by-3` cell array containing the historical credit ratings data of the form:

```
'00010283' '10-Nov-1984' 'CCC'
'00010283' '12-May-1986' 'B'
'00010283' '29-Jun-1988' 'CCC'
'00010283' '12-Dec-1991' 'D'
```

```
'00013326' '09-Feb-1985' 'A'  
'00013326' '24-Feb-1994' 'AA'  
'00013326' '10-Nov-2000' 'BBB'  
'00014413' '23-Dec-1982' 'B'
```

where each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). Column 3 is the rating assigned to the corresponding ID on the corresponding date. All information corresponding to the same ID must be stored in contiguous rows. Sorting this information by date is not required, but recommended for efficiency. IDs, dates and ratings are usually stored in string format, but they can also be entered in numeric format.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

'algorithm'

Estimation algorithm, in string format. Valid values are `duration` or `cohort`.

Default: `duration`

'endDate'

End date of the estimation time window, in string or numeric format. The `endDate` cannot be a date before the `startDate`.

Default: Latest date in `data`

'labels'

Cell array of size `nRatings-by-1`, or `1-by-nRatings`, containing the credit-rating scale. It must be consistent with the ratings labels used in the third column of `data`.

Default: { 'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D' }

'snapsPerYear'

Integer indicating the number of credit-rating snapshots per year to be considered for the estimation. Valid values are 1, 2, 3, 4, 6, 12. This parameter is only used with the `cohort` algorithm.

Default: 1 — One snapshot per year

'startDate'

Start date of the estimation time window, in string or numeric format.

Default: Earliest date in data

'transInterval'

Length of the transition interval, in years.

Default: 1 — One year transition probabilities

Output Arguments

transMat

Matrix of transition probabilities in percent. The size of the transition matrix is nRatings-by-nRatings.

sampleTotals

Structure with fields:

- **totalsVec** — A vector of size 1-by-nRatings.
- **totalsMat** — A matrix of size nRatings-by-nRatings.
- **algorithm** — A string with values 'duration' or 'cohort'.

For the 'duration' algorithm, **totalsMat**(i,j) contains the total transitions observed out of rating i into rating j (all the diagonal elements are zero). The total time spent on rating i is stored in **totalsVec**(i). For example, if there are three rating categories, Investment Grade (IG), Speculative Grade (SG) and Default (D), and the following information:

Total time spent	IG	SG	D
in rating:	4859.09	1503.36	1162.05

Transitions		IG	SG	D
out of (row)	IG	0	89	7

```
into (column):      SG  202   0  32
                   D   0   0   0
```

Then

```
totals.totalsVec = [4859.09 1503.36 1162.05]
totals.totalsMat = [  0  89   7
                   202  0  32
                   0   0  0]
```

totals.algorithm = 'duration'

For the 'cohort' algorithm, `totalsMat(i,j)` contains the total transitions observed from rating i to rating j , and `totalsVec(i)` is the initial count in rating i . For example, given the following information:

```
Initial count      IG      SG      D
in rating:         4808   1572   1145
```

```
Transitions       IG      SG      D
from (row)        IG  4721    80    7
to (column):      SG   193   1347   32
                  D    0     0   1145
```

Then

```
totals.totalsVec = [4808  1572  1145]
totals.totalsMat = [4721    80    7
                   193   1347   32
                   0     0   1145]
totals.algorithm = 'cohort'
```

idTotals

Struct array of size `nIDs`-by-1, where `nIDs` is the number of distinct IDs in column 1 of `data` when this is a cell array or, equivalently, equal to the length of the `idStart` field minus 1 when `data` is a preprocessed data structure. For each ID in the sample, `idTotals` contains one structure with the following fields:

- `totalsVec` — A sparse vector of size 1-by-`nRatings`.
- `totalsMat` — A sparse matrix of size `nRatings`-by-`nRatings`.
- `algorithm` — A string with values 'duration' or 'cohort'.

These fields contain the same information described for the output `sampleTotals`, but at an ID level. For example, for 'duration', `idTotals(k).totalsVec` contains the total time that the k -th company spent on each rating.

Examples

Construct a Transition Matrix From Historical Data of Credit Ratings

Using the historical credit rating input data from `Data_TransProb.mat` display the first ten rows and compute the transition matrix:

```
load Data_TransProb
data(1:10,:)

% Estimate transition probabilities with default settings
transMat = transprob(data)
```

ans =

```
'00010283'    '10-Nov-1984'    'CCC'
'00010283'    '12-May-1986'    'B'
'00010283'    '29-Jun-1988'    'CCC'
'00010283'    '12-Dec-1991'    'D'
'00013326'    '09-Feb-1985'    'A'
'00013326'    '24-Feb-1994'    'AA'
'00013326'    '10-Nov-2000'    'BBB'
'00014413'    '23-Dec-1982'    'B'
'00014413'    '20-Apr-1988'    'BB'
'00014413'    '16-Jan-1998'    'B'
```

transMat =

Columns 1 through 7

```
93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
 1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
 0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
 0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
 0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
 0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
 0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0
```

Column 8

```
0.0017
```

```
0.0396
0.0753
0.2193
0.7050
2.4399
12.7167
100.0000
```

Using the historical credit rating input data from `Data_TransProb.mat`, compute the transition matrix using the `cohort` algorithm:

```
%Estimate transition probabilities with 'cohort' algorithm
transMatCoh = transprob(data, 'algorithm', 'cohort')
```

```
transMatCoh =
```

```
Columns 1 through 7
```

```
93.1345    5.9335    0.7456    0.1553    0.0311         0         0
 1.7359   92.9198    4.5446    0.6046    0.1560         0         0
 0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0
 0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421
 0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873
      0         0    0.0761    0.7230   7.9909    86.1872    2.7397
      0         0         0    0.3094    1.8561    4.5630   80.8971
      0         0         0         0         0         0         0
```

```
Column 8
```

```
0
0.0390
0.0725
0.2103
0.6409
2.2831
12.3743
100.0000
```

Using the historical credit rating data with ratings investment grade ('IG'), speculative grade ('SG'), and default ('D'), from `Data_TransProb.mat` display the first ten rows and compute the transition matrix:


```
dataIGSG(1:10,:)
transMatIGSG = transprob(dataIGSG, 'labels', {'IG', 'SG', 'D'})
```

```
ans =
```

```
'00011253'    '04-Apr-1983'    'IG'
'00012751'    '17-Feb-1985'    'SG'
'00012751'    '19-May-1986'    'D'
'00014690'    '17-Jan-1983'    'IG'
'00012144'    '21-Nov-1984'    'IG'
'00012144'    '25-Mar-1992'    'SG'
'00012144'    '07-May-1994'    'IG'
'00012144'    '23-Jan-2000'    'SG'
'00012144'    '20-Aug-2001'    'IG'
'00012937'    '07-Feb-1984'    'IG'
```

```
transMatIGSG =
```

```
98.6719    1.2020    0.1261
 3.5781    93.3318    3.0901
      0         0 100.0000
```

Using the historical credit rating data with numeric ratings for investment grade (1), speculative grade (2), and default (3), from `Data_TransProb.mat` display the first ten rows and compute the transition matrix:

```
dataIGSGnum(1:10,:)
transMatIGSGnum = transprob(dataIGSGnum, 'labels', {1,2,3})
```

```
ans =
```

```
'00011253'    '04-Apr-1983'    [1]
'00012751'    '17-Feb-1985'    [2]
'00012751'    '19-May-1986'    [3]
'00014690'    '17-Jan-1983'    [1]
'00012144'    '21-Nov-1984'    [1]
'00012144'    '25-Mar-1992'    [2]
'00012144'    '07-May-1994'    [1]
'00012144'    '23-Jan-2000'    [2]
'00012144'    '20-Aug-2001'    [1]
'00012937'    '07-Feb-1984'    [1]
```

```
transMatIGSGnum =  
    98.6719    1.2020    0.1261  
    3.5781    93.3318    3.0901  
         0         0 100.0000
```

More About

Cohort Estimation

The cohort algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time. If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates.

Duration Estimation

Unlike the cohort method, the duration algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur. There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time.

Algorithms

Cohort Estimation

The algorithm first determines a sequence t_0, \dots, t_K of snapshot dates. The elapsed time, in years, between two consecutive snapshot dates t_{k-1} and t_k is equal to $1 / ns$, where ns is the number of snapshots per year. These $K + 1$ dates determine K transition periods.

The algorithm computes N_i^n , the number of transition periods in which obligor n starts at rating i . These are added up over all obligors to get N_i , the number of obligors in the sample that start a period at rating i . The number periods in which obligor n starts at rating i and ends at rating j , or migrates from i to j , denoted by N_{ij}^n , is also computed.

These are also added up to get N_{ij} , the total number of migrations from i to j in the sample.

The estimate of the transition probability from i to j in one period, denoted by P_{ij} , is given by:

$$P_{ij} = \frac{N_{ij}}{N_i}$$

These probabilities are arranged in a one-period transition matrix P_0 , where the i,j entry in P_0 is P_{ij} .

If the number of snapshots per year ns is 4 (quarterly snapshots), the probabilities in P_0 are 3-month (or 0.25-year) transition probabilities. You may, however, be interested in 1-year or 2-year transition probabilities. The latter time interval is called the transition interval, Δt , and it is used to convert P_0 into the final transition matrix, P , according to the formula:

$$P = P_0^{ns\Delta t}$$

For example, if $ns = 4$ and $\Delta t = 2$, P contains the 2-year transition probabilities estimated from quarterly snapshots.

Note: For the cohort algorithm, optional output arguments `idTotals` and `sampleTotals` from `transprob` contain the following information:

- `idTotals(n).totalsVec = (N_i^n)_{\forall i}`
- `idTotals(n).totalsMat = (N_{i,j}^n)_{\forall ij}`
- `idTotals(n).algorithm = 'cohort'`
- `sampleTotals.totalsVec = (N_i)_{\forall i}`
- `sampleTotals.totalsMat = (N_{i,j})_{\forall ij}`
- `sampleTotals.algorithm = 'cohort'`

For efficiency, the vectors and matrices in `idTotals` are stored as sparse arrays.

Duration Estimation

The algorithm computes T_i^n , the total time that obligor n spends in rating i within the estimation time window. These quantities are added up over all obligors to get T_i , the total time spent in rating i , collectively, by all obligors in the sample. The algorithm also computes T_{ij}^n , the number times that obligor n migrates from rating i to rating j , with i not equal to j , within the estimation time window. And it also adds them up to get T_{ij} , the total number of migrations, by all obligors in the sample, from the rating i to j , with i not equal to j .

To estimate the transition probabilities, the duration algorithm first needs to compute a generator matrix Λ . Each off-diagonal entry of this matrix is an estimate of the transition rate out of rating i into rating j , and is given by:

$$\lambda_{ij} = \frac{T_{ij}}{T_i}, i \neq j$$

The diagonal entries are computed as:

$$\lambda_{ii} = -\sum_{j \neq i} \lambda_{ij}$$

With the generator matrix and the transition interval Δt (e.g., $\Delta t = 2$ corresponds to 2-year transition probabilities), the transition matrix is obtained as $P = \exp(\Delta t \Lambda)$, where *exp* denotes matrix exponentiation (`expm` in MATLAB).

Note: For the duration algorithm, optional output arguments `idTotals` and `sampleTotals` from `transprob` contain the following information:

- `idTotals(n).totalsVec = (T_i^n) \forall i`

- `idTotals(n).totalsMat = (Ti,jn)ij`
- `idTotals(n).algorithm = 'duration'`
- `sampleTotals.totalsVec = (Ti)i`
- `sampleTotals.totalsMat = (Ti,j)ij`
- `sampleTotals.algorithm = 'duration'`

For efficiency, the vectors and matrices in `idTotals` are stored as sparse arrays.

- “Estimation of Transition Probabilities” on page 8-21

References

Hanson, S., T. Schuermann, "Confidence Intervals for Probabilities of Default," *Journal of Banking & Finance*, Elsevier, vol. 30(8), pages 2281-2301, August 2006.

Löffler, G., P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.

Schuermann, T., "Credit Migration Matrices," in E. Melnick, B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment*, Wiley, 2008.

See Also

| `transprobbytals` | `transprobprep`

transprobytotals

Estimate transition probabilities using `totals` structure input

Syntax

```
[transMat, sampleTotals] = transprobytotals(totals)
[transMat, sampleTotals] = transprobytotals(totals,
Name, Value)
```

Description

`[transMat, sampleTotals] = transprobytotals(totals)` estimates transition probabilities using a `totals` structure input.

`[transMat, sampleTotals] = transprobytotals(totals, Name, Value)` estimates transition probabilities using a `totals` structure input with additional options specified by one or more `Name, Value` pair arguments.

`transprobytotals` is useful for removing outlier information, obtaining bootstrapped confidence intervals, or computing transition probability estimates for different periodicity parameters (1-year transitions, 2-year transitions, etc.) in an efficient manner.

Input Arguments

`totals`

This can be:

- `totalsVec` — A sparse vector of size 1-by-`nRatings1`.
- `totalsMat` — A sparse matrix of size `nRatings1`-by-`nRatings2` with `nRatings1` ≤ `nRatings2`.
- `algorithm` — A string with values 'duration' or 'cohort'.

For the 'duration' algorithm, `totalsMat(i,j)` contains the total transitions observed out of rating *i* into rating *j* (all the diagonal elements are 0). The total time spent on

rating i is stored in `totalsVec(i)`. For example, you have three rating categories, Investment Grade (IG), Speculative Grade (SG) and Default (D), and the following information:

```
Total time spent      IG      SG      D
in rating:           4859.09  1503.36  1162.05
```

```
Transitions
out of (row)      IG  SG  D
into (column):   SG 202  0  32
                  D   0  0  0
```

Then:

```
totals.totalsVec = [4859.09  1503.36  1162.05]
totals.totalsMat = [  0  89  7
                   202  0  32
                   0  0  0]
totals.algorithm = 'duration'
```

For the 'cohort' algorithm, `totalsMat(i,j)` contains the total transitions observed from rating i to rating j , and `totalsVec(i)` is the initial count in rating i . For example, given the following information:

```
Initial count      IG      SG      D
in rating:         4808  1572  1145
```

```
Transitions
from (row)      IG  SG  D
to (column):   SG 193 1347  32
                D   0  0  1145
```

Then:

```
totals.totalsVec = [4808  1572  1145]
totals.totalsMat = [4721  80  7
                   193 1347  32
                   0  0  1145]
totals.algorithm = 'cohort'
```

Common totals structures are the optional output arguments from `transprob`:

- `sampleTotals` — A single structure summarizing the totals information for the whole dataset.
- `idTotals` — A struct array with the totals information at the ID level.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

'snapsPerYear'

Integer indicating the number of credit-rating snapshots per year to be considered for the estimation. Values are 1, 2, 3, 4, 6, or 12. This argument is only used with the `cohort` algorithm.

Default: 1 — One snapshot per year

'transInterval'

Length of the transition interval, in years.

Default: 1 — One-year transition probabilities

Output Arguments

`transMat`

Matrix of transition probabilities in percent. The size of the transition matrix is `nRatings1-by-nRatings2`.

`sampleTotals`

Structure with fields:

- `totalsVec` — A vector of size `1-by-nRatings1`.
- `totalsMat` — A matrix of size `nRatings1-by-nRatings2` with `nRatings1 ≤ nRatings2`.
- `algorithm` — A string with values 'duration' or 'cohort'.

If `totals` is a struct array, `sampleTotals` contains the aggregated information. That is, `sampleTotals.totalsVec` is the sum of `totals(k).totalsVec` over all k , and similarly for `totalsMat`. When `totals` is itself a single structure, `sampleTotals` and `totals` are the same.

Examples

Estimate Transition Probabilities Using a totals Structure Input

Use historical credit rating input data from `Data_TransProb.mat` and `transprob` to generate input for `transprobytotals`:

```
load Data_TransProb
```

```
% Call TRANSPROB with three output arguments
[transMat, sampleTotals, idTotals] = transprob(data);
transMat
```

```
transMat =
```

```
Columns 1 through 7
```

```
93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
 1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
 0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
 0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
 0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
 0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
 0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0
```

```
Column 8
```

```
0.0017
0.0396
0.0753
0.2193
0.7050
2.4399
12.7167
100.0000
```

Suppose companies 4 and 27 are outliers and you want to remove them from the pre-processed `idTotals` struct array and estimate the new transition probabilities.

```
idTotals([4 27]) = [];
[transMat1, sampleTotals1] = transprobytotals(idTotals);
```

```
transMat1
```

```
transMat1 =
```

```
Columns 1 through 7
```

```
93.1172    5.8427    0.8231    0.1763    0.0377    0.0012    0.0001
 1.6213   93.1501    4.3584    0.6614    0.1631    0.0055    0.0004
 0.1239    2.9027   92.2297    4.0628    0.5367    0.0661    0.0028
 0.0236    0.2313    5.0070   90.1825    3.7986    0.4734    0.0642
 0.0216    0.1134    0.6357    5.7959   88.9866    3.4497    0.2920
 0.0010    0.0062    0.1081    0.8697    7.3367   86.7217    2.5171
 0.0002    0.0011    0.0120    0.2591    1.4340    4.3034   81.3027
         0         0         0         0         0         0         0
```

```
Column 8
```

```
0.0017
0.0397
0.0753
0.2193
0.7050
2.4395
12.6875
100.0000
```

Obtain the 1-year, 2-year, 3-year, 4-year, and 5-year default probabilities, without the outlier information (i.e., using `sampleTotals1`).

```
DefProb = zeros(7,5);
for t = 1:5
    transMatTemp = transprobytotals(sampleTotals1, 'transInterval', t);
    DefProb(:,t) = transMatTemp(1:7,8);
end
DefProb
```

```
DefProb =
```

```
0.0017    0.0070    0.0159    0.0285    0.0450
0.0397    0.0828    0.1299    0.1813    0.2377
0.0753    0.1606    0.2567    0.3640    0.4831
0.2193    0.4675    0.7430    1.0445    1.3700
```

0.7050	1.4668	2.2759	3.1232	4.0000
2.4395	4.9282	7.4071	9.8351	12.1847
12.6875	23.1184	31.7177	38.8282	44.7266

More About

Cohort Estimation

The `cohort` algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time. If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates. For more information, see “Algorithms” on page 18-1416.

Duration Estimation

Unlike the `cohort` algorithm, the `duration` algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur. There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time. For more information, see “Algorithms” on page 18-1416.

- “Estimation of Transition Probabilities” on page 8-21

References

Hanson, S., T. Schuermann, "Confidence Intervals for Probabilities of Default," *Journal of Banking & Finance*, Elsevier, vol. 30(8), pages 2281–2301, August 2006.

Löffler, G., P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.

Schuermann, T., "Credit Migration Matrices," in E. Melnick, B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment*, Wiley, 2008.

See Also

| [transprob](#) | [transprobgrouptotals](#)

transprobfromthresholds

Convert from credit quality thresholds to transition probabilities

Syntax

```
trans = transprobfromthresholds(thresh)
```

Description

`trans = transprobfromthresholds(thresh)` transforms credit quality thresholds into transition probabilities.

Input Arguments

thresh

M-by-N matrix of credit quality thresholds. In each row, the first element must be `Inf` and the entries must satisfy the following monotonicity condition:

$$\text{thresh}(i,j) \geq \text{thresh}(i,j+1), \text{ for } 1 \leq j < N$$

The M-by-N input `thresh` and the M-by-N output `trans` are related as follows. The thresholds `thresh(i,j)` are critical values of a standard normal distribution z , such that:

$$\text{trans}(i,N) = P[z < \text{thresh}(i,N)],$$

$$\text{trans}(i,j) = P[z < \text{thresh}(i,j)] - P[z < \text{thresh}(i,j+1)], \text{ for } 1 \leq j < N$$

Any given row in the output matrix `TRANS` determines a probability distribution over a discrete set of N ratings 'R1', ..., 'RN', so that for any row i `TRANS(i,j)` is the probability of migrating into 'Rj'. `TRANS` can be a standard transition matrix, with $M \leq N$, in which case row i contains the transition probabilities for issuers with rating 'Ri'. But `TRANS` does not have to be a standard transition matrix. `TRANS` can contain individual transition probabilities for a set of M -specific issuers, with $M > N$.

For example, suppose there are only $N=3$ ratings, 'High', 'Low', and 'Default', with these credit quality thresholds:

	High	Low	Default
High	Inf	-2.0814	-3.1214
Low	Inf	2.4044	-1.7530

The matrix of transition probabilities is then:

	High	Low	Default
High	98.13	1.78	0.09
Low	0.81	95.21	3.98

This means the probability of default for 'High' is equivalent to drawing a standard normal random number smaller than -3.1214 , or 0.09%. The probability that a 'High' will end up the period with a rating of 'Low' or lower is equivalent to drawing a standard normal random number smaller than -2.0814 , or 1.87%. From here, the probability of ending with a 'Low' rating is:

$$P[z < -2.0814] - P[z < -3.1214] = 1.87\% - 0.09\% = 1.78\%$$

And the probability of ending with a 'High' rating is:

$$100\% - 1.87\% = 98.13\%$$

where 100% is the same as:

$$P[z < \text{Inf}]$$

Output Arguments

trans

M-by-N matrix with transition probabilities, in percent.

Examples

Transform Credit Quality Thresholds Into Transition Probabilities

Use historical credit rating input data from `Data_TransProb.mat`, estimate transition probabilities with default settings.

```
load Data_TransProb
```

```
% Estimate transition probabilities with default settings
transMat = transprob(data)
```

transMat =

Columns 1 through 7

93.1170	5.8428	0.8232	0.1763	0.0376	0.0012	0.0001
1.6166	93.1518	4.3632	0.6602	0.1626	0.0055	0.0004
0.1237	2.9003	92.2197	4.0756	0.5365	0.0661	0.0028
0.0236	0.2312	5.0059	90.1846	3.7979	0.4733	0.0642
0.0216	0.1134	0.6357	5.7960	88.9866	3.4497	0.2919
0.0010	0.0062	0.1081	0.8697	7.3366	86.7215	2.5169
0.0002	0.0011	0.0120	0.2582	1.4294	4.2898	81.2927
0	0	0	0	0	0	0

Column 8

0.0017
 0.0396
 0.0753
 0.2193
 0.7050
 2.4399
 12.7167
 100.0000

Obtain the credit quality thresholds.

thresh = transprobt thresholds(transMat)

thresh =

Columns 1 through 7

Inf	-1.4846	-2.3115	-2.8523	-3.3480	-4.0083	-4.1276
Inf	2.1403	-1.6228	-2.3788	-2.8655	-3.3166	-3.3523
Inf	3.0264	1.8773	-1.6690	-2.4673	-2.9800	-3.1631
Inf	3.4963	2.8009	1.6201	-1.6897	-2.4291	-2.7663
Inf	3.5195	2.9999	2.4225	1.5089	-1.7010	-2.3275
Inf	4.2696	3.8015	3.0477	2.3320	1.3838	-1.6491
Inf	4.6241	4.2097	3.6472	2.7803	2.1199	1.5556
Inf	Inf	Inf	Inf	Inf	Inf	Inf

Column 8

```

-4.1413
-3.3554
-3.1736
-2.8490
-2.4547
-1.9703
-1.1399
  Inf

```

Recover the transition probabilities.

```
trans = transprobfromthresholds(thresh)
```

```
trans =
```

```
Columns 1 through 7
```

```

93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
 1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
 0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
 0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
 0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
 0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
 0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
      0          0          0          0          0          0          0

```

```
Column 8
```

```

0.0017
0.0396
0.0753
0.2193
0.7050
2.4399
12.7167
100.0000

```

More About

- “Credit Quality Thresholds” on page 8-69

References

Gupton, G. M., C. C. Finger, and M. Bhatia, *CreditMetrics*, Technical Document, RiskMetrics Group, Inc., 2007.

See Also

| `transprob` | `transprobytotals` | `transprobt thresholds`

transprobgrouptotals

Aggregate credit ratings information into fewer rating categories

Syntax

```
totalsGrouped = transprobgrouptotals(totals,groupingEdges)
```

Description

`totalsGrouped = transprobgrouptotals(totals,groupingEdges)` aggregates the credit ratings information stored in the `totals` input into fewer ratings categories, which are defined by the `groupingEdges` argument.

Input Arguments

totals

Structure, or a struct array of length `nTotals`, with fields:

- `totalsVec` — A vector of size 1-by-`nRatings1`.
- `totalsMat` — A matrix of size `nRatings1`-by-`nRatings2`, with `nRatings1` ≤ `nRatings2`.
- `algorithm` — A string with values 'duration' or 'cohort'.

For the 'duration' algorithm, `totalsMat(i,j)` contains the total transitions observed out of rating *i* into rating *j* (all the diagonal elements are 0). The total time spent on rating *i* is stored in `totalsVec(i)`. For example, if there are three rating categories, Investment Grade (IG), Speculative Grade (SG) and Default (D), and the following information:

Total time spent	IG	SG	D
in rating:	4859.09	1503.36	1162.05

Transitions		IG	SG	D
out of (row)	IG	0	89	7
into (column):	SG	202	0	32

Then:

```
totals.totalsVec = [4859.09 1503.36 1162.05]
totals.totalsMat = [ 0  89  7
                   202 0 32
                   0  0  0]
```

```
totals.algorithm = 'duration'
```

For the 'cohort' algorithm, `totalsMat(i,j)` contains the total transitions observed from rating i to rating j , and `totalsVec(i)` is the initial count in rating i . For example, given the following information:

Initial count	IG	SG	D
in rating:	4808	1572	1145

Transitions		IG	SG	D
from (row)	IG	4721	80	7
to (column):	SG	193	1347	32
	D	0	0	1145

Then:

```
totals.totalsVec = [4808 1572 1145]
totals.totalsMat = [4721  80  7
                   193 1347 32
                   0   0 1145]
totals.algorithm = 'cohort'
```

Note: Common totals structures are the optional output arguments from `transprob`:

- `sampleTotals` — A single structure summarizing the totals information for the whole dataset.
- `idTotals` — A struct array with the totals information at the ID level.

For more information, see “Algorithms” on page 18-1416.

groupingEdges

Numeric array with increasing, positive integers to indicate how to group credit ratings into categories.

This table illustrates how to group a list of whole ratings into investment grade (IG) and speculative grade (SG) categories. Eight ratings are in the original list. Ratings 1 to 4 are

IG, ratings 5 to 7 are SG, and rating 8 is a category of its own. In this example, the array of grouping edges is [4 7 8].

Original ratings:	'AAA'	'AA'	'A'	'BBB'	'BB'	'B'	'CCC'	'D'
Relative ordering:	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Grouped ratings:		'IG'				'SG'		'D'
Grouping edges:				(4)			(7)	(8)

In general, if `groupingEdges` has K elements $\text{edge1} < \text{edge2} < \dots < \text{edgeK}$, ratings 1 to edge1 (inclusive) are grouped in the first category, ratings $\text{edge1}+1$ to edge2 in the second category, and so forth.

Regarding the last element, edgeK :

- If $n\text{Ratings1}$ equals $n\text{Ratings2}$, then edgeK must equal $n\text{Ratings1}$. This leads to K groups, and $n\text{RatingsGrouped1} = n\text{RatingsGrouped2} = K$.
- If $n\text{Ratings1} < n\text{Ratings2}$, then either:
 - edgeK equals $n\text{Ratings1}$, in which case ratings $\text{edgeK}+1, \dots, n\text{Ratings2}$ will be treated as categories of their own. This results in $K+(n\text{Ratings2}-\text{edgeK})$ groups, with $n\text{RatingsGrouped1} = K$ and $n\text{RatingsGrouped2} = K + (n\text{Ratings2} - \text{edgeK})$; or
 - edgeK equals $n\text{Ratings2}$, in which case there must be a j th edge element, edge_j , such that edge_j equals $n\text{Ratings1}$. This leads to K groups, and $n\text{RatingsGrouped1} = j$ and $n\text{RatingsGrouped2} = K$.

Output Arguments

totalsGrouped

Structure, or a struct array of length `nTotals`, with fields:

- `totalsVec` — A vector of size 1-by- $n\text{RatingsGrouped1}$.
- `totalsMat` — A matrix of size $n\text{RatingsGrouped1}$ -by- $n\text{RatingsGrouped2}$.
- `algorithm` — A string, 'duration' or 'cohort'.

`nRatingsGrouped1` and `nRatingsGrouped2` are defined in the description of `groupingEdges`. Each structure contains aggregated information by categories, based

on the information provided in the corresponding structure in `totals`, according to the grouping of ratings defined by `groupingEdges` and consistent with the `algorithm` choice.

Following the examples in the description of the `totals` input, suppose `IG` and `SG` are grouped into a single `ND` (Not-Defaulted) category, using the edges `[2 3]`. For the `'cohort'` algorithm, the output is:

```
totalsGrouped.totalsVec = [6380  1145]
totalsGrouped.totalsMat = [6341    39
                          0    1145]
totalsGrouped.algorithm = 'cohort'
and for the 'duration' algorithm:
```

```
totalsGrouped.totalsVec = [6362.45  1162.05]
totalsGrouped.totalsMat = [0  39
                          0  0]
totalsGrouped.algorithm = 'duration'
```

Examples

Aggregate the Credit Ratings Information Stored in the `totals` Input

Use historical credit rating input data from `Data_TransProb.mat`. Load input data from file `Data_TransProb.mat`.

```
load Data_TransProb
```

```
% Call TRANSPROB with two output arguments
[transMat, sampleTotals] = transprob(data);
transMat
```

```
transMat =
```

```
Columns 1 through 7
```

```
93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
 1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
 0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
 0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
 0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
```

```

0.0010    0.0062    0.1081    0.8697    7.3366    86.7215    2.5169
0.0002    0.0011    0.0120    0.2582    1.4294    4.2898    81.2927
      0          0          0          0          0          0          0

```

Column 8

```

0.0017
0.0396
0.0753
0.2193
0.7050
2.4399
12.7167
100.0000

```

Group into investment grade (ratings 1-4) and speculative grade (ratings 5-7); note, the default is the last rating (number 8).

```

edges = [4 7 8];
sampleTotalsGrp = transprobgrouptotals(sampleTotals,edges);

```

```

% Transition matrix at investment grade / speculative grade level
transMatIGSG = transprobbytotals(sampleTotalsGrp)

```

transMatIGSG =

```

98.5336    1.3608    0.1056
 3.9155    92.9692    3.1153
      0          0 100.0000

```

Obtain the 1-year, 2-year, 3-year, 4-year, and 5-year default probabilities at investment grade and speculative grade level.

```

DefProb = zeros(2,5);
for t = 1:5
transMatTemp = transprobbytotals(sampleTotalsGrp, 'transInterval',t);
DefProb(:,t) = transMatTemp(1:2,3);
end
DefProb

```

DefProb =

0.1056	0.2521	0.4359	0.6537	0.9027
3.1153	6.0157	8.7179	11.2373	13.5881

More About

Cohort Estimation

The `cohort` algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time. If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates. For more information, see “Algorithms” on page 18-1416.

Duration Estimation

Unlike the `cohort` algorithm, the `duration` algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur. There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time. For more information, see “Algorithms” on page 18-1416.

- “Estimation of Transition Probabilities” on page 8-21

References

Hanson, S., T. Schuermann, "Confidence Intervals for Probabilities of Default," *Journal of Banking & Finance*, Elsevier, vol. 30(8), pages 2281–2301, August 2006.

Löffler, G., P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.

Schuermann, T., "Credit Migration Matrices," in E. Melnick, B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment*, Wiley, 2008.

See Also

| `transprob` | `transprobbytals`

transprobbrep

Preprocess credit ratings data to estimate transition probabilities

Syntax

```
[prepData] = transprobbrep(data)
[prepData] = transprobbrep(data, Name, Value)
```

Description

[prepData] = transprobbrep(data) preprocesses credit ratings historical data (i.e., credit migration data) for the subsequent estimation of transition probabilities.

[prepData] = transprobbrep(data, Name, Value) preprocesses credit ratings historical data (i.e., credit migration data) for the subsequent estimation of transition probabilities with additional options specified by one or more Name, Value pair arguments.

Input Arguments

data

Historical input data for credit ratings. Cell array of size nRecords-by-3 containing the credit ratings. Each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). The assigned credit rating corresponds to the associated ID on the associated date. All information corresponding to the same ID must be stored in contiguous rows. Sorting this information by date is not required but is recommended. IDs, dates, and ratings are usually stored in string format, but they can also be entered in numeric format. Here is an example with all of the information in string format:

```
'00010283'    '10-Nov-1984'    'CCC'
'00010283'    '12-May-1986'    'B'
'00010283'    '29-Jun-1988'    'CCC'
'00010283'    '12-Dec-1991'    'D'
'00013326'    '09-Feb-1985'    'A'
'00013326'    '24-Feb-1994'    'AA'
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, . . . , `NameN`, `ValueN`.

'labels'

Cell array of size `nRatings`-by-1, or 1-by-`nRatings`, containing the credit-rating scale. It must be consistent with the ratings labels used in the third column of `data`.

Default: { 'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D' }

Output Arguments

`prepData`

Structure with the following fields:

- `idStart` — Array of size $(nIDs+1)$ -by-1, where $nIDs$ is the number of distinct IDs in column 1 of `data`. This array summarizes where the credit ratings information corresponding to each company starts and ends. The dates and ratings corresponding to company j in `data` are stored from row `idStart(j)` to row `idStart(j+1)-1` of `numericDates` and `numericRatings`.
- `numericDates` — Array of size $nRecords$ -by-1, containing the dates in column 2 of `data`, in numeric format.
- `numericRatings` — Array of size $nRecords$ -by-1, containing the ratings in column 3 of `data`, mapped into numeric format.
- `ratingsLabels` — Cell array of size 1-by-`nRatings`, containing the credit rating scale.

Examples

Aggregate the Credit Ratings Information Stored in the totals Input

Use historical credit rating input data from `Data_TransProb.mat`. Load input data from file `Data_TransProb.mat`.


```

load Data_TransProb

% Call TRANSPROB with two output arguments
[transMat, sampleTotals] = transprob(data);
transMat

transMat =

Columns 1 through 7

    93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
     1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
     0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
     0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
     0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
     0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
     0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

Column 8

     0.0017
     0.0396
     0.0753
     0.2193
     0.7050
     2.4399
    12.7167
   100.0000

```

Group into investment grade (ratings 1-4) and speculative grade (ratings 5-7); note, the default is the last rating (number 8).

```

edges = [4 7 8];
sampleTotalsGrp = transprobgrouptotals(sampleTotals,edges);

% Transition matrix at investment grade / speculative grade level
transMatIGSG = transprobbytotals(sampleTotalsGrp)

```

```

transMatIGSG =

    98.5336    1.3608    0.1056

```

```
3.9155    92.9692    3.1153
         0         0 100.0000
```

Obtain the 1-year, 2-year, 3-year, 4-year, and 5-year default probabilities at investment grade and speculative grade level.

```
DefProb = zeros(2,5);
for t = 1:5
transMatTemp = transprobbytotals(sampleTotalsGrp, 'transInterval',t);
DefProb(:,t) = transMatTemp(1:2,3);
end
DefProb
```

```
DefProb =
```

```
0.1056    0.2521    0.4359    0.6537    0.9027
3.1153    6.0157    8.7179   11.2373   13.5881
```

More About

- “Estimation of Transition Probabilities” on page 8-21

See Also

| [transprobbytotals](#) | [transprob](#)

transprobtothresholds

Convert from transition probabilities to credit quality thresholds

Syntax

```
thresh = transprobtothresholds(trans)
```

Description

`thresh = transprobtothresholds(trans)` transforms transition probabilities into credit quality thresholds.

Input Arguments

trans

M-by-N matrix with transition probabilities, in percent. Entries cannot be negative and cannot exceed 100, and all rows must add up to 100.

Any given row in the M-by-N input matrix `trans` determines a probability distribution over a discrete set of N ratings. If the ratings are 'R1', ..., 'RN', then for any row *i* `trans(i,j)` is the probability of migrating into 'Rj'. If `trans` is a standard transition matrix, then $M \# N$ and row *i* contains the transition probabilities for issuers with rating 'Ri'. But `trans` does not have to be a standard transition matrix. `trans` can contain individual transition probabilities for a set of M-specific issuers, with $M > N$.

The credit quality thresholds `thresh(i,j)` are critical values of a standard normal distribution *z*, such that:

$$\text{trans}(i,N) = P[z < \text{thresh}(i,N)],$$

$$\text{trans}(i,j) = P[z < \text{thresh}(i,j)] - P[z < \text{thresh}(i,j+1)], \text{ for } 1 \leq j < N$$

This implies that `thresh(i,1) = Inf`, for all *i*. For example, suppose there are only $N=3$ ratings, 'High', 'Low', and 'Default', with the following transition probabilities:

```
High   Low   Default
```

```
High 98.13 1.78 0.09
Low  0.81 95.21 3.98
```

The matrix of credit quality thresholds is:

	High	Low	Default
High	Inf	-2.0814	-3.1214
Low	Inf	2.4044	-1.7530

This means the probability of default for 'High' is equivalent to drawing a standard normal random number smaller than -3.1214 , or 0.09%. The probability that a 'High' will end up the period with a rating of 'Low' or lower is equivalent to drawing a standard normal random number smaller than -2.0814 , or 1.87%. From here, the probability of ending with a 'Low' rating is:

$$P[z < -2.0814] - P[z < -3.1214] = 1.87\% - 0.09\% = 1.78\%$$

And the probability of ending with a 'High' rating is:

$$100\% - 1.87\% = 98.13\%$$

where 100% is the same as:

$$P[z < \text{Inf}]$$

Output Arguments

thresh

M-by-N matrix of credit quality thresholds.

Examples

Transform Transition Probabilities Into Credit Quality Thresholds

Use historical credit rating input data from `Data_TransProb.mat`. Load input data from file `Data_TransProb.mat`.

```
load Data_TransProb
```

```
% Estimate transition probabilities with default settings
transMat = transprob(data)
```

```
transMat =
```

```
Columns 1 through 7
```

93.1170	5.8428	0.8232	0.1763	0.0376	0.0012	0.0001
1.6166	93.1518	4.3632	0.6602	0.1626	0.0055	0.0004
0.1237	2.9003	92.2197	4.0756	0.5365	0.0661	0.0028
0.0236	0.2312	5.0059	90.1846	3.7979	0.4733	0.0642
0.0216	0.1134	0.6357	5.7960	88.9866	3.4497	0.2919
0.0010	0.0062	0.1081	0.8697	7.3366	86.7215	2.5169
0.0002	0.0011	0.0120	0.2582	1.4294	4.2898	81.2927
0	0	0	0	0	0	0

```
Column 8
```

```
0.0017
0.0396
0.0753
0.2193
0.7050
2.4399
12.7167
100.0000
```

Obtain the credit quality thresholds.

```
thresh = transprobtothresholds(transMat)
```

```
thresh =
```

```
Columns 1 through 7
```

Inf	-1.4846	-2.3115	-2.8523	-3.3480	-4.0083	-4.1276
Inf	2.1403	-1.6228	-2.3788	-2.8655	-3.3166	-3.3523
Inf	3.0264	1.8773	-1.6690	-2.4673	-2.9800	-3.1631
Inf	3.4963	2.8009	1.6201	-1.6897	-2.4291	-2.7663
Inf	3.5195	2.9999	2.4225	1.5089	-1.7010	-2.3275
Inf	4.2696	3.8015	3.0477	2.3320	1.3838	-1.6491
Inf	4.6241	4.2097	3.6472	2.7803	2.1199	1.5556
Inf	Inf	Inf	Inf	Inf	Inf	Inf

```
Column 8
```

-4.1413
-3.3554
-3.1736
-2.8490
-2.4547
-1.9703
-1.1399
Inf

More About

- “Credit Quality Thresholds” on page 8-69

References

Gupton, G. M., C. C. Finger, and M. Bhatia, *CreditMetrics*, Technical Document, RiskMetrics Group, Inc., 2007.

See Also

| `transprob` | `transprobbytotals` | `transprobfromthresholds`

ts2func

Convert time series arrays to 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 three-dimensional array.
-------	---

Optional Input Arguments

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:

- 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:

Times	Vector of monotonically increasing observation times associated with the time series input array <code>Array</code> . If you do not specify a value for this argument, <code>Times</code> is a zero-based, unit-increment vector of the same length as that of the dimension of <code>Array</code> associated with time (see <code>TimeDimension</code>).
TimeDimension	Positive scalar integer that specifies which dimension of the input time series array <code>Array</code> is associated with time. The value of this argument cannot be greater than the number of dimensions of <code>Array</code> . If you do not specify a value for this argument, the default is 1, indicating that time is associated with the rows of <code>Array</code> .
StateDimension	Positive scalar integer that specifies which dimension of the input time series array <code>Array</code> is associated with the NVARs state variables. <code>StateDimension</code> cannot be greater than the number of dimensions of <code>Array</code> . If you do not specify a value for this argument, <code>ts2func</code> assigns <code>StateDimension</code> the first dimension of <code>Array</code> that is not already associated with time (the state vector X_t is associated with the first available dimension of <code>Array</code> not already assigned to <code>TimeDimension</code>).
Deterministic	A scalar, logical flag to indicate whether the output function is a deterministic function of time alone. If <code>Deterministic</code> is true, the output function <code>F</code> is a deterministic function of time, $F(t)$, and the only input it will accept is a scalar, real-valued time t . If <code>Deterministic</code> is false, the output function <code>F</code> will accept two inputs, a scalar, real-valued time t followed by an NVARs-by-1 state vector $X(t)$. The default is false, and <code>F</code> is a callable function of time and state, $F(t,X)$.

Output Arguments

F	Callable function $F(t)$ of a real-valued scalar observation time t . You can invoke <code>F</code> with a second input (such as an NVARs-by-1 state vector X), which is a placeholder that <code>ts2func</code> ignores. For example, while $F(t)$ and $F(t,X)$ produce identical results, the latter directly supports SDE simulation methods. If the optional input argument
----------	--

	<p>Deterministic is true, F is a deterministic function of time, $F(t)$, and the only input it will accept is a scalar, real-valued time t. Otherwise, if Deterministic is false (the default), F will accept a scalar, real-valued time t followed by an NVARs-by-1 state vector $X(t)$.</p>
--	---

More About

Algorithms

- 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 three-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 $F(t) = F(\text{Times}(1))$.
- 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.
- The output function **F** is always a deterministic function of time, $F(t)$, and may always be called with a single input regardless of the **Deterministic** flag. The distinction is that when **Deterministic** is false, the function **F** may also be called with a second input, an NVARs-by-1 state vector $X(t)$, which is a placeholder and ignored. While $F(t)$ and $F(t,X)$ produce identical results, the former specifically indicates that the function is a deterministic function of time, and may offer significant performance benefits in some situations.

References

- 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.
- Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `simulate`

tsaccel

Acceleration between times

Syntax

```
acc = tsaccel(data, nTimes, datatype)
```

```
accts = tsaccel(tsobj, nTimes, datatype)
```

Arguments

<code>data</code>	Data series.
<code>nTimes</code>	(Optional) Number of times. Default = 12.
<code>datatype</code>	(Optional) Indicates whether <code>data</code> contains the data itself or the momentum of the data: 0 = Data contains the data itself (default). 1 = Data contains the momentum of the data.
<code>tsobj</code>	Name of an existing financial time series object.

Description

Acceleration is the difference of two momentums separated by some number of periods.

`acc = tsaccel(data, nTimes, datatype)` calculates the acceleration of a data series, essentially the difference of the current momentum with the momentum some number of periods ago. If `nTimes` is specified, `tsaccel` calculates the acceleration of a data series data with time distance of `nTimes` times.

`accts = tsaccel(tsobj, nTimes, datatype)` calculates the acceleration of the data series in the financial time series object `tsobj`, essentially the difference of the current momentum with the momentum some number of periods ago. Each data series in

`tsobj` is treated individually. `accts` is a financial time series object with similar dates and data series names as `tsobj`.

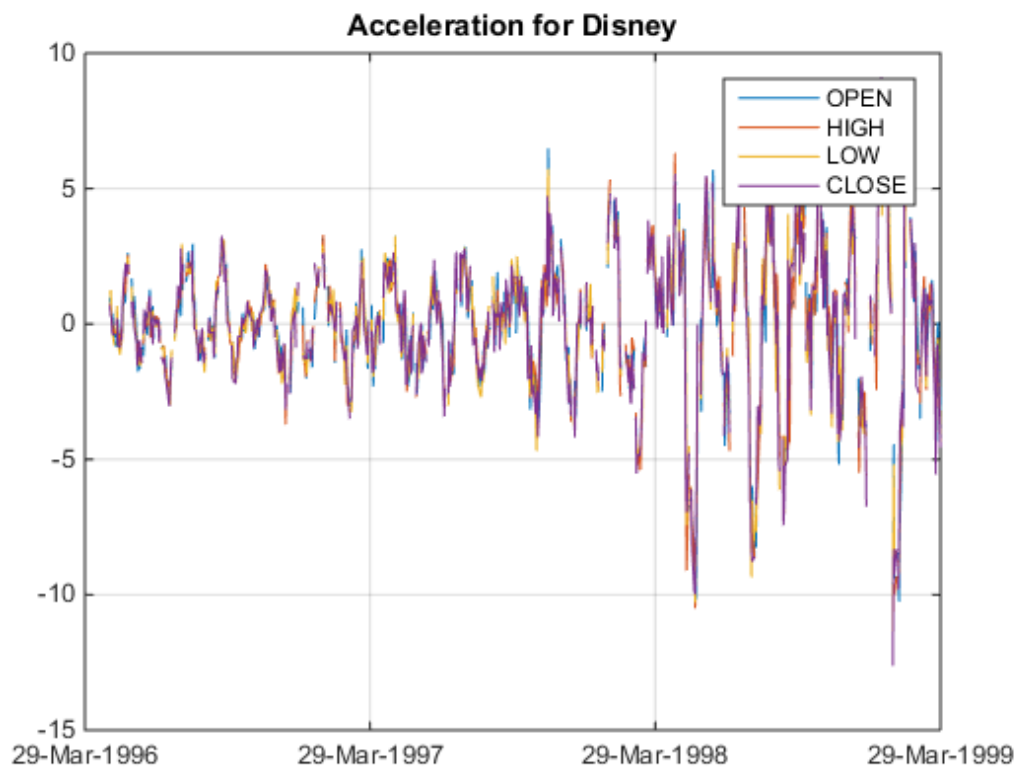
Note, to compute a quantity over n periods, you must specify $n+1$ for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Calculate the Acceleration of a Data Series

This example shows how to calculate the acceleration of a data series for Disney stock and plot the results.

```
load disney.mat
dis = rmfield(dis, 'VOLUME'); % remove VOLUME field
dis_Accel = tsaccel(dis);
plot(dis_Accel)
title('Acceleration for Disney')
```



References

Kaufman, P. J., *The New Commodity Trading Systems and Methods*, New York: John Wiley & Sons, 1987.

See Also

tsmom

tsmom

Momentum between times

Syntax

```
mom = tsmom(data, nTimes)
```

```
momts = tsmom(tsobj, nTimes)
```

Arguments

<code>data</code>	Data series. Column-oriented vector or matrix.
<code>nTimes</code>	(Optional) Number of times. Default = 12.
<code>tsobj</code>	Financial time series object.

Description

Momentum is the difference between two prices (data points) separated by a number of times.

`mom = tsmom(data, nTimes)` calculates the momentum of a data series `data`. If `nTimes` is specified, `tsmom` uses that value instead of the default 12.

`momts = tsmom(tsobj, nTimes)` calculates the momentum of all data series in the financial time series object `tsobj`. Each data series in `tsobj` is treated individually. `momts` is a financial time series object with similar dates and data series names as `tsobj`. If `nTimes` is specified, `tsmom` uses that value instead of the default 12.

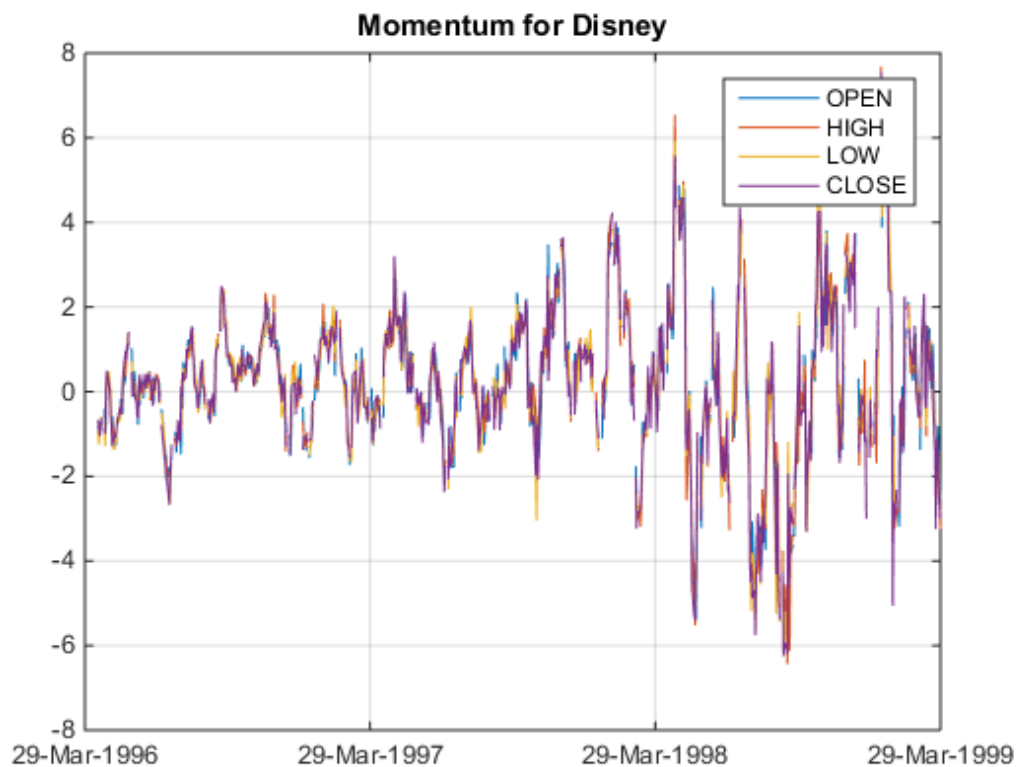
Note, to compute a quantity over n periods, you must specify $n+1$ for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Calculate the Momentum of a Data Series

This example shows how to calculate the momentum of a data series for Disney stock and plot the results.

```
load disney.mat
dis = rmfield(dis, 'VOLUME'); % remove VOLUME field
dis_Mom = tsmom(dis);
plot(dis_Mom)
title('Momentum for Disney')
```



References

Kaufman, P. J., *The New Commodity Trading Systems and Methods*, John Wiley and Sons, New York, 1987.

See Also

tsaccel

tsmovavg

Moving average

Syntax

```
output = tsmovavg(tsobj, 's', lag, dim)
output = tsmovavg(vector, 's', lag, dim)

output = tsmovavg(tsobj, 'e', timeperiod, dim)
output = tsmovavg(vector, 'e', timeperiod, dim)

output = tsmovavg(tsobj, 't', numperiod, dim)
output = tsmovavg(vector, 't', numperiod, dim)

output = tsmovavg(tsobj, 'w', weights, dim)
output = tsmovavg(vector, 'w', weights, dim)

output = tsmovavg(tsobj, 'm', numperiod, dim)
output = tsmovavg(vector, 'm', numperiod, dim)
```

Description

`output = tsmovavg(tsobj, 's', lag, dim)` returns the simple moving average for financial time series object, `tsobj`. `lag` indicates the number of previous data points used with the current data point when calculating the moving average.

`output = tsmovavg(vector, 's', lag, dim)` returns the simple moving average for a vector. `lag` indicates the number of previous data points used with the current data point when calculating the moving average.

`output = tsmovavg(tsobj, 'e', timeperiod, dim)` returns the exponential weighted moving average for financial time series object, `tsobj`. The exponential moving average is a weighted moving average, where `timeperiod` specifies the time period. Exponential moving averages reduce the lag by applying more weight to recent prices. For example, a 10-period exponential moving average weights the most recent price by 18.18%. ($2 / (\text{timeperiod} + 1)$).

`output = tsmovavg(vector, 'e', timeperiod, dim)` returns the exponential weighted moving average for a vector. The exponential moving average is a weighted

moving average, where `timeperiod` specifies the time period. Exponential moving averages reduce the lag by applying more weight to recent prices. For example, a 10-period exponential moving average weights the most recent price by 18.18%. ($2 / (\text{timeperiod} + 1)$).

`output = tsmovavg(tsobj, 't', numperiod, dim)` returns the triangular moving average for financial time series object, `tsobj`. The triangular moving average double-smooths the data. `tsmovavg` calculates the first simple moving average with window width of $\text{ceil}(\text{numperiod} + 1) / 2$. Then it calculates a second simple moving average on the first moving average with the same window size.

`output = tsmovavg(vector, 't', numperiod, dim)` returns the triangular moving average for a vector. The triangular moving average double-smooths the data. `tsmovavg` calculates the first simple moving average with window width of $\text{ceil}(\text{numperiod} + 1) / 2$. Then it calculates a second simple moving average on the first moving average with the same window size.

`output = tsmovavg(tsobj, 'w', weights, dim)` returns the weighted moving average for the financial time series object, `tsobj`, by supplying weights for each element in the moving window. The length of the weight vector determines the size of the window. If larger weight factors are used for more recent prices and smaller factors for previous prices, the trend is more responsive to recent changes.

`output = tsmovavg(vector, 'w', weights, dim)` returns the weighted moving average for the vector by supplying weights for each element in the moving window. The length of the weight vector determines the size of the window. If larger weight factors are used for more recent prices and smaller factors for previous prices, the trend is more responsive to recent changes.

`output = tsmovavg(tsobj, 'm', numperiod, dim)` returns the modified moving average for the financial time series object, `tsobj`. The modified moving average is similar to the simple moving average. Consider the argument `numperiod` to be the lag of the simple moving average. The first modified moving average is calculated like a simple moving average. Subsequent values are calculated by adding the new price and subtracting the last average from the resulting sum.

`output = tsmovavg(vector, 'm', numperiod, dim)` returns the modified moving average for the vector. The modified moving average is similar to the simple moving average. Consider the argument `numperiod` to be the lag of the simple moving average. The first modified moving average is calculated like a simple moving average. Subsequent values are calculated by adding the new price and subtracting the last average from the resulting sum.

Examples

Compute Five Forms of Moving Averages Using a Financial Time Series Object

Load the financial time series object, `dis` for Disney stock and look at the weekly data for this time series.

```
load disney.mat
weekly = toweekly(dis);
dates = (weekly.dates);
price = fts2mat(weekly.CLOSE);
```

Set the `|lag|` input argument for the window size for the moving average.

```
window_size = 12;
```

Calculate the simple moving average.

```
simple = tsmovavg(price, 's', window_size, 1);
```

Calculate the simple moving average.

```
simple = tsmovavg(price, 's', window_size, 1);
```

Calculate the exponential weighted moving average moving average.

```
exp = tsmovavg(price, 'e', window_size, 1);
```

Calculate the triangular moving average moving average.

```
tri = tsmovavg(price, 't', window_size, 1);
```

Calculate the weighted moving average moving average.

```
semi_gaussian = [0.026 0.045 0.071 0.1 0.12 0.138];
semi_gaussian = [semi_gaussian fliplr(semi_gaussian)];
weighted = tsmovavg(price, 'w', semi_gaussian, 1);
```

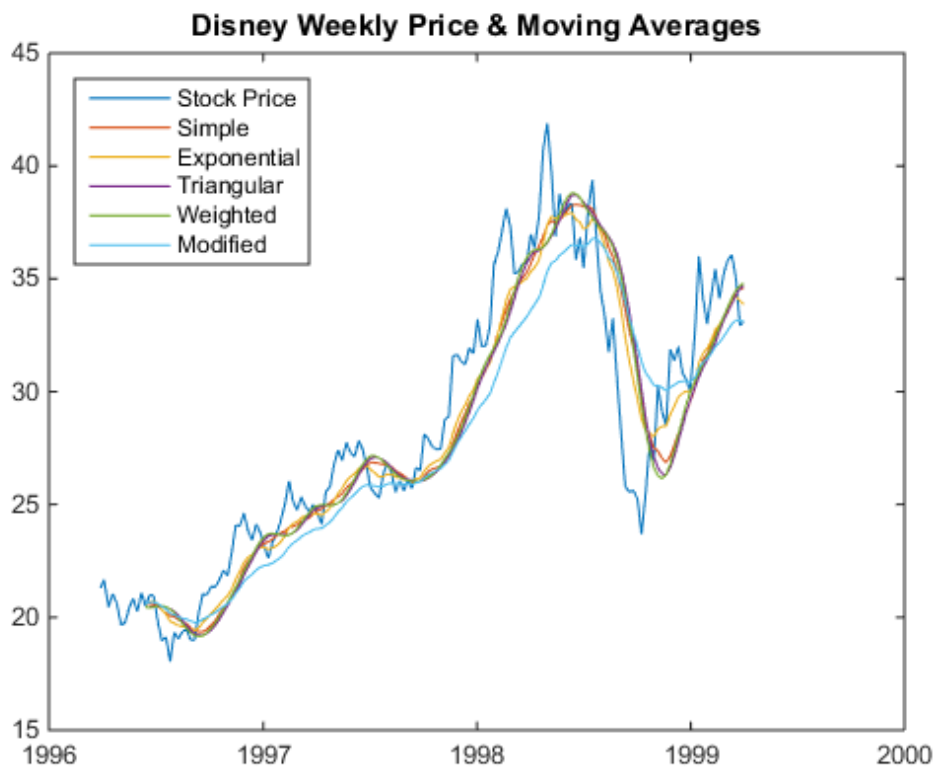
Calculate the modified moving average moving average.

```
modif = tsmovavg(price, 'm', window_size, 1);
```

Plot the results for the five moving average calculations for Disney stock.

```
plot(dates, price, ...
     dates, simple, ...
```

```
    dates,exp,...  
    dates,tri,...  
    dates,weighted,...  
    dates,modif)  
datetick  
legend('Stock Price', 'Simple', 'Exponential', 'Triangular', 'Weighted', ...  
      'Modified', 'Location', 'NorthWest')  
title('Disney Weekly Price & Moving Averages')
```



Input Arguments

tobj — Financial time series object
object

Financial time series object specified using a time series object created using `fints`.

lag — Number of previous data points

nonnegative integer

Number of previous data points specified as a nonnegative integer. `Lag` indicates the window size or number of periods of the moving average.

vector — Set of observations

vector or matrix

Set of observations specified as a vector or matrix.

dim — dimension to operate along

positive integer with value 1 or 2

Dimension to operate along, specified as a positive integer with a value of 1 or 2. `dim` is an optional input argument, and if it is not included as an input, the default value 2 is assumed. The default of `dim = 2` indicates a row-oriented matrix, where each row is a variable and each column is an observation.

If `dim = 1`, the input is assumed to be a column vector or column-oriented matrix, where each column is a variable and each row an observation.

timeperiod — Length of time period

nonnegative integer

Length of time period specified as a nonnegative integer.

numperiod — Number of periods considered

nonnegative integer

Number of periods considered specified as a nonnegative integer.

weights — Weights for each element in the moving window

vector of weights

Weights for each element in the window specified as a vector of weights.

Output Arguments

output — Moving average calculation

vector or matrix

Moving average calculation returned as a vector or matrix. The output returned from `tsmovavg` is identical in format to the input.

References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 184-192.

See Also

`mean` | `peravg`

typprice

Typical price

Syntax

```
tprc = typprice(highp, lowp, closep)
```

```
tprc = typprice([highp lowp closep])
```

```
tprcts = typprice(tsobj)
```

```
tprcts = typprice(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

highp	High price (vector).
lowp	Low price (vector).
closep	Closing price (vector).
tsobj	Financial time series object.

Description

`tprc = typprice(highp, lowp, closep)` calculates the typical prices `tprc` from the high (`highp`), low (`lowp`), and closing (`closep`) prices. The typical price is the average of the high, low, and closing prices for each period.

`tprc = typprice([highp lowp closep])` accepts a three-column matrix as the input rather than two individual vectors. The columns of the matrix represent the high, low, and closing prices, in that order.

`tprcts = typprice(tsobj)` calculates the typical prices from the stock data contained in the financial time series object `tsobj`. The object must contain, at least, the High, Low, and Close data series. The typical price is the average of the closing price

plus the high and low prices. `tprcts` is a financial time series object of the same dates as `tsobj` containing the data series `TypPrice`.

`tprcts = typprice(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Typical Price

This example shows how to compute the typical price for Disney stock and plot the results.

```
load disney.mat
dis_Typ = typprice(dis);
plot(dis_Typ)
title('Typical Price for Disney')
```




References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 291 - 292.

See Also

medprice | wclose

ugarch

Univariate GARCH(P,Q) parameter estimation with Gaussian innovations

Syntax

[Kappa, Alpha, Beta] = ugarch(U, P, Q)

Arguments

U	Single column vector of random disturbances, that is, the residuals or innovations ($\#_t$), of an econometric model representing a mean-zero, discrete-time stochastic process. The innovations time series U is assumed to follow a GARCH(P,Q) process. Note: The latest value of residuals is the last element of vector U .
P	Nonnegative, scalar integer representing a model order of the GARCH process. P is the number of lags of the conditional variance. P can be zero; when P = 0, a GARCH(0,Q) process is actually an ARCH(Q) process.
Q	Positive, scalar integer representing a model order of the GARCH process. Q is the number of lags of the squared innovations.

Description

[Kappa, Alpha, Beta] = ugarch(U, P, Q) computes estimated univariate GARCH(P,Q) parameters with Gaussian innovations.

Kappa is the estimated scalar constant term ($[[\text{KAPPA}]]$) of the GARCH process.

Alpha is a **P**-by-1 vector of estimated coefficients, where **P** is the number of lags of the conditional variance included in the GARCH process.

Beta is a **Q**-by-1 vector of estimated coefficients, where **Q** is the number of lags of the squared innovations included in the GARCH process.

The time-conditional variance, σ_t^2 , of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = K + \sum_{i=1}^P \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^Q \beta_j \varepsilon_{t-j}^2,$$

where α represents the argument **Alpha**, β represents **Beta**, and the GARCH(P, Q) coefficients $\{K, \alpha, \beta\}$ are subject to the following constraints.

$$\begin{aligned} \sum_{i=1}^P \alpha_i + \sum_{j=1}^Q \beta_j &< 1 \\ K &> 0 \\ \alpha_i &\geq 0 \quad i = 1, 2, \dots, P \\ \beta_j &\geq 0 \quad j = 1, 2, \dots, Q. \end{aligned}$$

Note that \mathbf{U} is a vector of residuals or innovations ($\#_t$) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, $\#_t$ and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t,$$

where $\{v_t\}$ is an independent, identically distributed (iid) sequence $\sim N(0,1)$.

Note The Econometrics Toolbox software provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information, see the Econometrics Toolbox documentation or the financial products Web page at <http://www.mathworks.com/products/finprod/>.

Examples

See `ugarchsim` for an example of a GARCH(P,Q) process.

References

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

See Also

ugarchsim

ugarch1lf

Log-likelihood objective function of univariate GARCH(P,Q) processes with Gaussian innovations

Syntax

LogLikelihood = ugarch1lf(Parameters, U, P, Q)

Arguments

Parameters	(1 + P + Q)-by-1 column vector of GARCH(P,Q) process parameters. The first element is the scalar constant term $[\text{KAPPA}]$ of the GARCH process; the next P elements are coefficients associated with the P lags of the conditional variance terms; the next Q elements are coefficients associated with the Q lags of the squared innovations terms.
U	Single column vector of random disturbances, that is, the residuals or innovations ($\#_t$), of an econometric model representing a mean-zero, discrete-time stochastic process. The innovations time series U is assumed to follow a GARCH(P,Q) process. Note: The latest value of residuals is the last element of vector U.
P	Nonnegative, scalar integer representing a model order of the GARCH process. P is the number of lags of the conditional variance. P can be zero; when P = 0, a GARCH(0,Q) process is actually an ARCH(Q) process.
Q	Positive, scalar integer representing a model order of the GARCH process. Q is the number of lags of the squared innovations.

Description

LogLikelihood = ugarch1lf(Parameters, U, P, Q) computes the log-likelihood objective function of univariate GARCH(P,Q) processes with Gaussian innovations.

`LogLikelihood` is a scalar value of the GARCH(P,Q) log-likelihood objective function given the input arguments. This function is meant to be optimized via the `fmincon` function of the Optimization Toolbox software.

`fmincon` is a minimization routine. To maximize the log-likelihood function, the `LogLikelihood` output parameter is actually the negative of what is formally presented in most time series or econometrics references.

The time-conditional variance, σ_t^2 , of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = K + \sum_{i=1}^P \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^Q \beta_j \varepsilon_{t-j}^2,$$

where α represents the argument `Alpha`, and β represents `Beta`.

\mathbf{U} is a vector of residuals or innovations ($\#_t$) representing a mean-zero, discrete time stochastic process. Although σ_t^2 is generated via the equation above, $\#_t$ and σ_t^2 are related as

$$\varepsilon_t = \sigma_t \nu_t,$$

where $\{\nu_t\}$ is an independent, identically distributed (iid) sequence $\sim N(0,1)$.

Since `ugarch11f` is really just a helper function, no argument checking is performed. This function is not meant to be called directly from the command line.

Note The Econometrics Toolbox software provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information, see the Econometrics Toolbox documentation or the financial products Web page at <http://www.mathworks.com/products/finprod/>.

See Also

`ugarch` | `ugarchsim`

ugarchpred

Forecast conditional variance of univariate GARCH(P,Q) processes

Syntax

[VarianceForecast, H] = ugarchpred(U, Kappa, Alpha, Beta, NumPeriods)

Arguments

U	Single column vector of random disturbances, that is, the residuals or innovations ($\#_t$), of an econometric model representing a mean-zero, discrete-time stochastic process. The innovations time series U is assumed to follow a GARCH(P,Q) process. Note: The latest value of residuals is the last element of vector U.
Kappa	Scalar constant term [[KAPPA]] of the GARCH process.
Alpha	P-by-1 vector of coefficients, where P is the number of lags of the conditional variance included in the GARCH process. Alpha can be an empty matrix, in which case P is assumed 0; when $P = 0$, a GARCH(0,Q) process is actually an ARCH(Q) process.
Beta	Q-by-1 vector of coefficients, where Q is the number of lags of the squared innovations included in the GARCH process.
NumPeriods	Positive, scalar integer representing the forecast horizon of interest, expressed in periods compatible with the sampling frequency of the input innovations column vector U.

Description

[VarianceForecast, H] = ugarchpred(U, Kappa, Alpha, Beta, NumPeriods) forecasts the conditional variance of univariate GARCH(P,Q) processes.

VarianceForecast is a number of periods (**NUMPERIODS**)-by-1 vector of the minimum mean-square error forecast of the conditional variance of the innovations time series vector **U** (that is, $\{\#_t\}$). The first element contains the 1-period-ahead forecast, the second element contains the 2-period-ahead forecast, and so on. Thus, if a forecast horizon greater than 1 is specified (**NUMPERIODS** > 1), the forecasts of all intermediate horizons are returned as well. In this case, the last element contains the variance forecast of the specified horizon, **NumPeriods** from the most recent observation in **U**.

H is a vector of the conditional variances (σ_t^2) corresponding to the innovations vector **U**. It is inferred from the innovations **U**, and is a reconstruction of the “past” conditional variances, whereas the **VarianceForecast** output represents the projection of conditional variances into the “future.” This sequence is based on setting pre-sample values of σ_t^2 to the unconditional variance of the $\{\#_t\}$ process. **H** is a single column vector of the same length as the input innovations vector **U**.

The time-conditional variance, σ_t^2 , of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = K + \sum_{i=1}^P \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^Q \beta_j \varepsilon_{t-j}^2,$$

where a represents the argument **Alpha**, β represents **Beta**, and the GARCH(P,Q) coefficients $\{K, a, \beta\}$ are subject to the following constraints.

$$\begin{aligned} \sum_{i=1}^P \alpha_i + \sum_{j=1}^Q \beta_j &< 1 \\ K &> 0 \\ \alpha_i &\geq 0 \quad i = 1, 2, \dots, P \\ \beta_j &\geq 0 \quad j = 1, 2, \dots, Q. \end{aligned}$$

Note that **U** is a vector of residuals or innovations ($\{\#_t\}$) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, $\#_t$ and σ_t^2 are related as

$$\varepsilon_t = \sigma_t \nu_t,$$

where $\{v_t\}$ is an independent, identically distributed (iid) sequence $\sim N(0,1)$.

Note The Econometrics Toolbox software provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information, see the Econometrics Toolbox documentation or the financial products Web page at <http://www.mathworks.com/products/finprod/>.

Examples

See `ugarchsim` for an example of forecasting the conditional variance of a univariate GARCH(P,Q) process.

See Also

`ugarch` | `ugarchsim`

ugarchsim

Simulate univariate GARCH(P,Q) process with Gaussian innovations

Syntax

[U, H] = ugarchsim(Kappa, Alpha, Beta, NumSamples)

Arguments

Kappa	Scalar constant term [[KAPPA]] of the GARCH process.
Alpha	P-by-1 vector of coefficients, where P is the number of lags of the conditional variance included in the GARCH process. Alpha can be an empty matrix, in which case P is assumed 0; when P = 0, a GARCH(0,Q) process is actually an ARCH(Q) process.
Beta	Q-by-1 vector of coefficients, where Q is the number of lags of the squared innovations included in the GARCH process.
NumSamples	Positive, scalar integer indicating the number of samples of the innovations U and conditional variance H (see below) to simulate.

Description

[U, H] = ugarchsim(Kappa, Alpha, Beta, NumSamples) simulates a univariate GARCH(P,Q) process with Gaussian innovations.

U is a number of samples (NUMSAMPLES)-by-1 vector of innovations ($\#_t$), representing a mean-zero, discrete-time stochastic process. The innovations time series U is designed to follow the GARCH(P,Q) process specified by the inputs Kappa, Alpha, and Beta.

H is a NUMSAMPLES-by-1 vector of the conditional variances (σ_t^2) corresponding to the innovations vector U. Note that U and H are the same length, and form a “matching” pair of vectors. As shown in the following equation, σ_t^2 (that is, H(t)) represents the time series inferred from the innovations time series $\{\#_t\}$ (that is, U).

The time-conditional variance, σ_t^2 , of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = K + \sum_{i=1}^P \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^Q \beta_j \varepsilon_{t-j}^2,$$

where α represents the argument **Alpha**, β represents **Beta**, and the GARCH(P,Q) coefficients $\{K, \alpha, \beta\}$ are subject to the following constraints.

$$\begin{aligned} \sum_{i=1}^P \alpha_i + \sum_{j=1}^Q \beta_j &< 1 \\ K &> 0 \\ \alpha_i &\geq 0 \quad i = 1, 2, \dots, P \\ \beta_j &\geq 0 \quad j = 1, 2, \dots, Q. \end{aligned}$$

Note that \mathbf{U} is a vector of residuals or innovations ($\#_t$) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, $\#_t$ and σ_t^2 are related as

$$\varepsilon_t = \sigma_t \nu_t,$$

where $\{\nu_t\}$ is an independent, identically distributed (iid) sequence $\sim N(0,1)$.

The output vectors \mathbf{U} and \mathbf{H} are designed to be steady-state sequences in which transients have arbitrarily small effect. The (arbitrary) metric used by `ugarchsim` strips the first N samples of \mathbf{U} and \mathbf{H} such that the sum of the GARCH coefficients, excluding **Kappa**, raised to the N th power, does not exceed 0.01.

$$0.01 = (\text{sum}(\text{Alpha}) + \text{sum}(\text{Beta}))^N$$

Thus

$$N = \log(0.01) / \log((\text{sum}(\text{Alpha}) + \text{sum}(\text{Beta})))$$

Note The Econometrics Toolbox software provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see

the Econometrics Toolbox documentation or the financial products Web page at <http://www.mathworks.com/products/finprod/>.

Examples

This example simulates a GARCH(P,Q) process with $P = 2$ and $Q = 1$.

```
% Set the random number generator seed for reproducibility.
rng('default')

% Set the simulation parameters of GARCH(P,Q) = GARCH(2,1) process.
Kappa = 0.25;      %a positive scalar.
Alpha = [0.2 0.1]'; %a column vector of nonnegative numbers (P = 2).
Beta = 0.4;       % Q = 1.
NumSamples = 500; % number of samples to simulate.

% Now simulate the process.
[U , H] = ugarchsim(Kappa, Alpha, Beta, NumSamples);

% Estimate the process parameters.
P = 2;    % Model order P (P = length of Alpha).
Q = 1;    % Model order Q (Q = length of Beta).
[k, a, b] = ugarch(U , P , Q);
disp(' ')
disp(' Estimated Coefficients:')
disp(' -----')
disp([k; a; b])
disp(' ')

% Forecast the conditional variance using the estimated
% coefficients.
NumPeriods = 10; % Forecast out to 10 periods.
[VarianceForecast, H1] = ugarchpred(U, k, a, b, NumPeriods);
disp(' Variance Forecasts:')
disp(' -----')
disp(VarianceForecast)
disp(' ')
```

When the above code is executed, the screen output looks like the display shown.

```
Diagnostic Information

Number of variables: 4

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

Constraints
Nonlinear constraints: do not exist
```

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

Algorithm selected
 medium-scale: SQP, Quasi-Newton, line-search

End diagnostic information

Iter	F-count	f(x)	Max constraint	Line search steplength	Directional derivative	First-order optimality	Procedure
0	5	625.735	-0.05				
1	12	611.683	-0.05614	0.25	-146	411	
2	18	609.396	-0.0375	0.5	-98.1	94.8	
3	24	600.17	-0.07969	0.5	-61.9	56.1	
4	31	598.818	-0.05977	0.25	-24	27.9	
5	38	598.308	-0.04482	0.25	-12.4	42.9	
6	43	598.247	0	1	-15.7	23.9	
7	48	598.02	0	1	-22.3	3.59	
8	57	598.01	-0.02759	0.0625	-2.03	3.89	
9	62	597.999	-0.01749	1	-1.48	0.249	
10	67	597.999	-0.01638	1	-0.15	0.0365	
11	72	597.999	-0.0164	1	-0.0269	0.00774	Hessian modified

Local minimum possible. Constraints satisfied.

fmincon stopped because the predicted change in the objective function is less than the default value of the function tolerance and constraints are satisfied to within the default value of the constraint tolerance.

No active inequalities.

Estimated Coefficients:

 0.2868
 0.1984
 0.0164
 0.4076

Variance Forecasts:

 0.4130
 0.5453
 0.6240
 0.6740
 0.7055
 0.7254
 0.7380
 0.7460
 0.7510
 0.7542

References

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

See Also

ugarch | ugarchpred

uicalendar

Graphical calendar

Syntax

```
uicalendar('PARAM1', VALUE1, 'PARAM2', VALUE2', ...)
```

Arguments

'BusDays '	<p>Values are:</p> <ul style="list-style-type: none"> • 0 — (Default) Standard calendar without nonbusiness day indicators. • 1 — Marks NYSE nonbusiness days in red.
'BusDaySelect '	<p>Values are:</p> <ul style="list-style-type: none"> • 0 — Only allow selection of business days. Nonbusiness days are determined from the following parameters: <ul style="list-style-type: none"> • 'BusDays ' • 'Holiday ' • 'Weekend ' • 1 — (Default) Allows selections of business and nonbusiness days.
'DateBoxColor '	[date R G B] : Sets the color of the date squares to the specified [R G B] color.
'DateStrColor '	[date R G B] : Sets the color of the numeric date number in the date square to the specified [R G B] color.
'DestinationUI '	<p>Values are:</p> <ul style="list-style-type: none"> • H — Scalar or vector of the destination object's handles. The default UI property that is populated with the date(s) is 'string'.

	<ul style="list-style-type: none"> • <code>{H, {Prop}}</code> — Cell array of handles and the destination object's UI properties. <code>H</code> must be a scalar or vector and <code>Prop</code> must be a single property string or a cell array of property strings.
'Holiday'	Sets the specified holiday dates into the calendar. The corresponding date string of the holiday will appear Red. The <code>Date(s)</code> must be a scalar or vector of datenums.
'InitDate'	<p>Values are:</p> <ul style="list-style-type: none"> • <code>Datenum</code> — Numeric date value specifying the initial start date when the calendar is initialized. The default date is <code>TODAY</code>. • <code>Datestr</code> — Date string value specifying the initial start date when the calendar is initialized. <code>Datestr</code> must include a Year, Month, and Day (for example, <code>01-Jan-2006</code>).
'InputDateFormat'	<code>Format</code> — Sets the format of initial start date, <code>InitDate</code> . See ' <code>help datestr</code> ' for date format values.
'OutputDateFormat'	<code>Format</code> — Sets the format of output date string. See ' <code>help datestr</code> ' for date format values.
'OutputDateStyle'	<p>Values are:</p> <ul style="list-style-type: none"> • <code>0</code> — (Default) Returns a single date string or a cell array (row) of date string. For example, <code>{ '01-Jan-2001, 02-Jan-2001, ... '}</code>. • <code>1</code> — Returns a single date string or a cell (column) array of date strings. For example, <code>{ '01-Jan-2001; 02-Jan-2001; ... '}</code>. • <code>2</code> — Returns a string representation of a row vector of datenums. For example, <code>' [732758, 732759, 732760, 732761] '</code>. • <code>3</code> — Returns a string representation of a column vector of datenums. For example, <code>' [732758; 732759; 732760; 732761] '</code>.

'SelectionType'	<p>Values are:</p> <ul style="list-style-type: none"> • 0 — Allows multiple date selections. • 1 — (Default) Allows only a single date selection.
'Weekend'	<p>DayOfWeek — Sets the specified days of the week as weekend days. Weekend days are marked in red. DayOfWeek can be a vector containing the following numeric values:</p> <ul style="list-style-type: none"> • 1 — Sunday • 2 — Monday • 3 — Tuesday • 4 — Wednesday • 5 — Thursday • 6 — Friday • 7 — Saturday <p>Also this value can be a vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days then WEEKEND = [1 0 0 0 0 0 1].</p>
'WindowStyle'	<p>Values are:</p> <ul style="list-style-type: none"> • Normal — (Default) Standard figure properties. • Modal — Modal figures remain stacked above all normal figures and the MATLAB Command Window.

Description

`uicalendar('PARAM1', VALUE1, 'PARAM2', VALUE2, ...)` supports a customizable graphical calendar that interfaces with `uicontrols`. `uicalendar` populates `uicontrols` with user-selected dates.

Examples

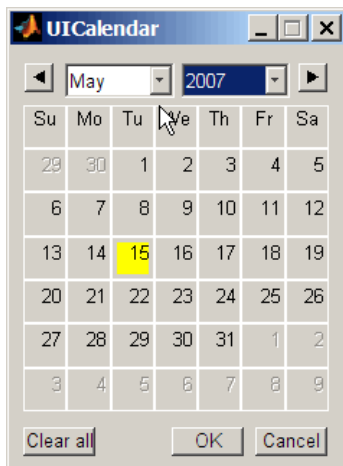
Create a uicontrol:

```
textH1 = uicontrol('style', 'edit', 'position', [10 10 100 20]);
```

Call UICalendar:

```
uicalendar('DestinationUI', {textH1, 'string'})
```

Select a date and click 'OK'.



For more information on using `uicalendar` with an application, see “Example of Using UICalendar with an Application” on page 15-5.

See Also

holidays

uminus

Unary minus of financial time series object

Syntax

uminus

Description

uminus implements unary minus for a financial time series object.

See Also

uplus

uplus

Unary plus of financial time series object

Syntax

`uplus`

Description

`uplus` implements unary plus for a financial time series object.

See Also

`uminus`

var

Variance

Syntax

`y = var(X)`

`y = var(X, 1)`

`y = var(X, W)`

`y = var(X, W, DIM)`

Arguments

<code>X</code>	Financial times series object.
<code>W</code>	Weight vector used in calculating variance.
<code>DIM</code>	Dimension of <code>X</code> used in calculating variance.

Description

`var` supports financial time series objects based on the MATLAB `var` function. See `var` in the MATLAB documentation.

`y = var(X)`, if `X` is a financial time series object and returns the variance of each series.

`var` normalizes `y` by $N - 1$ if $N > 1$, where N is the sample size. This is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples. For $N = 1$, `y` is normalized by N .

`y = var(X, 1)` normalizes by N and produces the second moment of the sample about its mean. `var(X, 0)` is the same as `var(X)`.

`y = var(X, W)` computes the variance using the weight vector `W`. The length of `W` must equal the length of the dimension over which `var` operates, and its elements must be

nonnegative. `var` normalizes W to sum to 1. Use a value of 0 for W to use the default normalization by $N - 1$, or use a value of 1 to use N .

`y = var(X, W, DIM)` takes the variance along the dimension `DIM` of X .

Examples

The variance is the square of the standard deviation. Consider if

```
f = fints((today:today+1)', [4 -2 1; 9 5 7])
```

then

```
var(f, 0, 1)
```

is

```
[12.5 24.5 18.0]
```

and

```
var(f, 0, 2)
```

is

```
[9.0; 4.0]
```

See Also

`corrcoef` | `cov` | `mean` | `std`

vertcat

Concatenate financial time series objects vertically

Syntax

```
vertcat
```

Description

`vertcat` implements vertical concatenation of financial time series objects. `vertcat` essentially adds data points to a time series object. Objects to be vertically concatenated must not have any duplicate dates and/or times or any overlapping dates and/or times. The description fields are concatenated as well. They are separated by `||`.

Examples

Create two financial time series objects with daily frequencies:

```
myfts = fints((today:today+4)', (1:5)', 'DataSeries', 'd');
yourfts = fints((today+5:today+9)', (11:15)', 'DataSeries', 'd');
```

Use `vertcat` to concatenate them vertically:

```
newfts1 = [myfts; yourfts]
```

```
newfts1 =
```

```
desc:  ||
freq:  Daily (1)

'dates: (10)'      'DataSeries: (10)'
'11-Dec-2001'     [          1]
'12-Dec-2001'     [          2]
'13-Dec-2001'     [          3]
'14-Dec-2001'     [          4]
'15-Dec-2001'     [          5]
'16-Dec-2001'     [         11]
'17-Dec-2001'     [         12]
```

```
'18-Dec-2001' [ 13]
'19-Dec-2001' [ 14]
'20-Dec-2001' [ 15]
```

Create two financial time series objects with different frequencies:

```
myfts = fints((today:today+4)', (1:5)', 'DataSeries', 'd');
hisfts = fints((today+5:7:today+34)', (11:15)', 'DataSeries', ...
'w');
```

Concatenate these two objects vertically:

```
newfts2 = [myfts; hisfts]
```

```
newfts2 =
  desc:  ||
  freq: Unknown (0)

  'dates: (10)'   'DataSeries: (10)'
  '11-Dec-2001' [ 1]
  '12-Dec-2001' [ 2]
  '13-Dec-2001' [ 3]
  '14-Dec-2001' [ 4]
  '15-Dec-2001' [ 5]
  '16-Dec-2001' [ 11]
  '23-Dec-2001' [ 12]
  '30-Dec-2001' [ 13]
  '06-Jan-2002' [ 14]
  '13-Jan-2002' [ 15]
```

If all frequency indicators are the same, the new object has the same frequency indicator. However, if one of the concatenated objects has a different `freq` from the other(s), the frequency of the resulting object is set to `Unknown (0)`. In these examples, `newfts1` has `Daily` frequency, while `newfts2` has `Unknown (0)` frequency.

See Also

`horzcat`

volarea

Price and volume chart

Syntax

volarea(X)

Arguments

X	M-by-3 matrix where the first column contains date numbers, the second column is the asset price, and the third column is the volume.
---	---

Description

volarea(X) plots asset date, price, and volume on a single axis.

Examples

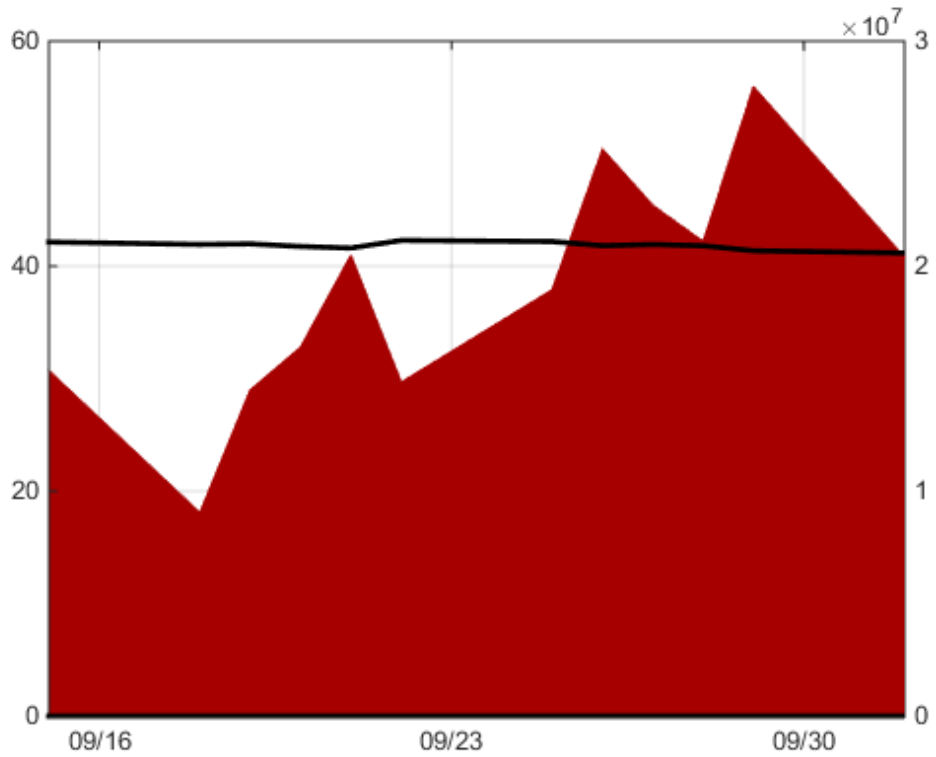
Plot Asset Date, Price, and Volume on a Single Axis

This example shows how to plot asset date, price, and volume on a single axis, given asset X as an M-by-3 matrix of date numbers, asset price, and volume.

```
X = [...
733299.00      41.99    15045445.00; ...
733300.00      42.14    15346658.00; ...
733303.00      41.93     9034397.00; ...
733304.00      41.98    14486275.00; ...
733305.00      41.75    16389872.00; ...
733306.00      41.61    20475208.00; ...
733307.00      42.29    14833200.00; ...
733310.00      42.19    18945176.00; ...
733311.00      41.82    25188101.00; ...
```

```
733312.00      41.93  22689878.00;...
733313.00      41.81  21084723.00;...
733314.00      41.37  27963619.00;...
733317.00      41.17  20385033.00;...
733318.00      42.02  27783775.00];
```

volarea(X)



See Also

[bolling](#) | [kagi](#) | [candle](#) | [highlow](#) | [linebreak](#) | [movavg](#) | [pointfig](#) | [priceandvol](#) | [renko](#)

volroc

Volume rate of change

Syntax

```
vroc = volroc(tvolume nTimes)
```

```
vrocts = volroc(tsobj, nTimes)
```

```
vrocts = volroc(tsobj, nTimes, ParameterName, ParameterValue)
```

Arguments

<code>tvolume</code>	Volume traded.
<code>nTimes</code>	(Optional) Time difference. Default = 12.
<code>tsobj</code>	Financial time series object.

Description

`vroc = volroc(tvolume nTimes)` calculates the volume rate of change, `vroc`, from the volume traded data `tvolume`. If `nTimes` is specified, the volume rate of change is calculated between the current volume and the volume `nTimes` ago.

`vrocts = volroc(tsobj, nTimes)` calculates the volume rate of change, `vrocts`, from the financial time series object `tsobj`. The `vrocts` output is a financial time series object with similar dates as `tsobj` and a data series named `VolumeROC`. If `nTimes` is specified, the volume rate of change is calculated between the current volume and the volume `nTimes` ago.

`vrocts = volroc(tsobj, nTimes, ParameterName, ParameterValue)` specifies the name for the required data series when it is different from the default name. The valid parameter name is

- `VolumeName`: volume traded series name

The parameter value is a string that represents the valid parameter name.

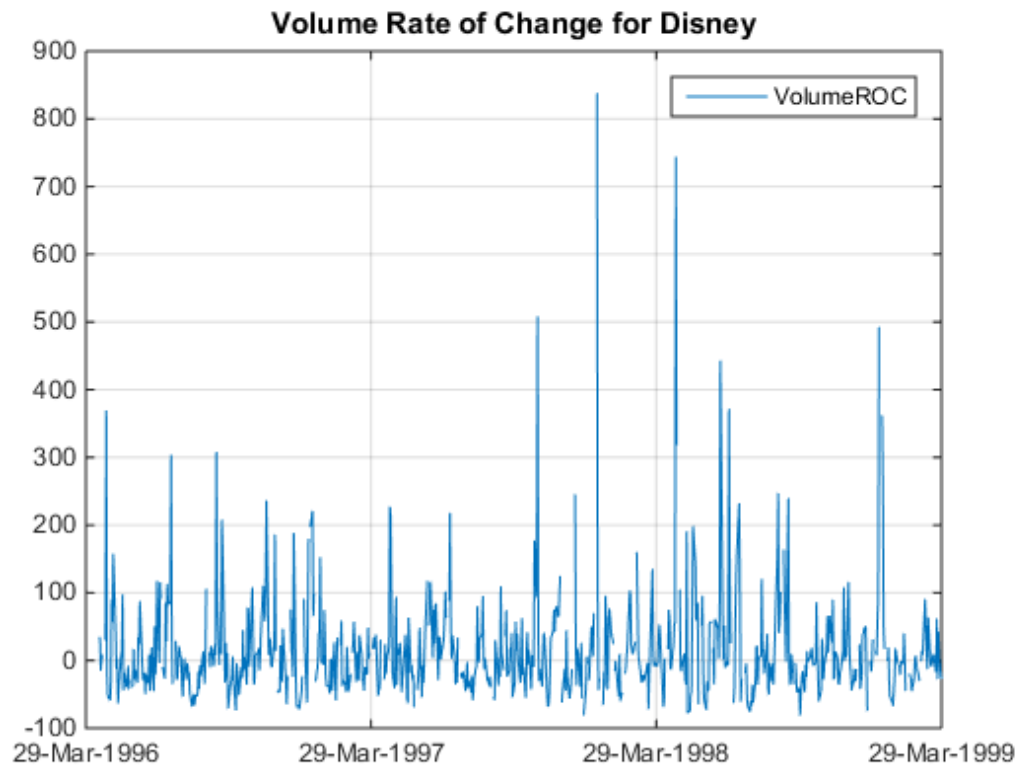
Note, to compute a quantity over n periods, you must specify $n+1$ for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Compute the Volume Rate of Change

This example shows how to compute the volume rate of change for Disney stock and plot the results.

```
load disney.mat
dis_VolRoc = volroc(dis);
plot(dis_VolRoc)
title('Volume Rate of Change for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 310 - 311.

See Also

prcroc

wclose

Weighted close

Syntax

```
wcls = wclose(highp, lowp, closep)
```

```
wcls = wclose([highp lowp closep])
```

```
wclsts = wclose(tsoobj)
```

```
wclsts = wclose(tsoobj, ParameterName, ParameterValue, ...)
```

Arguments

<code>highp</code>	High price (vector).
<code>lowp</code>	Low price (vector).
<code>closep</code>	Closing price (vector).
<code>tsoobj</code>	Financial time series object.

Description

The weighted close price is the average of twice the closing price plus the high and low prices.

`wcls = wclose(highp, lowp, closep)` calculates the weighted close prices `wcls` based on the high (`highp`), low (`lowp`), and closing (`closep`) prices per period.

`wcls = wclose([highp lowp closep])` accepts a three-column matrix consisting of the high, low, and closing prices, in that order.

`wclsts = wclose(tsoobj)` computes the weighted close prices for a set of stock price data contained in the financial time series object `tsoobj`. The object must contain the

high, low, and closing prices needed for this function. The function assumes that the series are named **High**, **Low**, and **Close**. All three are required. `wclsts` is a financial time series object of the same dates as `tsobj` and contains the data series named `WCclose`.

`wclsts = wclose(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Weighted Closing Prices

This example shows how to compute the weighted closing prices for Disney stock and plot the results.

```
load disney.mat
dis_Wclose = wclose(dis);
plot(dis_Wclose)
title('Weighted Closing Prices for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 312 - 313.

See Also

medprice | typprice

weeknum

Week in year

Syntax

[N]= weeknum(D)

[N] = weeknum(D, W, E)

Arguments

D	Serial date number or a date string.
W	(Optional) A numeric representation of the day a week begins. The week start values and their corresponding day are: <ul style="list-style-type: none"> • 1 (default) — Sunday • 2 — Monday • 3 — Tuesday • 4 — Wednesday • 5 — Thursday • 6 — Friday • 7 — Saturday
E	(Optional) Indicates if the week of the year display is in the European standard. The European standard considers first week of year to be first week longer than 3 days, offset by the given week's start day. Set to 1 to use the European standard. The default setting is 0.

Description

[N]= weeknum(D) returns the week of the year given D, a serial date number or a date string.

[N] = weeknum(D, W, E) returns the week of the year given D, a serial date number or a date string, W, a numeric representation of the day a week begins, and when E is set to 1, the week of the year is in the European standard.

The weeknum function considers the week containing January 1 to be the first week of the year.

Examples

You can determine the week of the year using a serial date number

```
N = weeknum(728647)
```

```
N =
```

```
52
```

or a date string

```
N = weeknum('19-Dec-1994')
```

```
N =
```

```
52
```

The first week of the year must have at least 4 days in it. For example, January 8, 2004 was a Thursday. The European standard is used because the first week of the year is the first week longer than 3 days.

```
weeknum('08-jan-2004',1,1)
```

```
ans =
```

```
2
```

You can use weeknum with datenum:

```
weeknum(datenum('01-Jan-2004'):datenum('08-Jan-2004'))
```

```
ans =
```

```
1 1 1 2 2 2 2 2
```

The default start day of the week is Sunday. Every day after, and including the first Sunday of the year (04-Jan-2004), returns 2 denoting the second week. In this case, the first of week of the year started before January 1, 2004.

You can also use `weeknum` with `datenum` and specify a W value of 5 to indicate that the weeks start on Thursday:

```
weeknum(datenum('01-Jan-2004'):datenum('08-Jan-2004'),5)
```

```
ans =
```

```
     1     1     1     1     1     1     1     2
```

The first week of the year that has 4 or more days, based on the specified start day, is considered week one (even if this isn't the first week in the calendar). Any day falling in (or before) this week is given a week number of 1.

See Also

`datenum` | `datestr` | `datevec` | `day`

weights2holdings

Portfolio values and weights into holdings

Syntax

```
Holdings = weights2holdings(Values, Weights, Prices)
```

Arguments

Values	Scalar or number of portfolios (NPORTS) vector containing portfolio values.
Weights	NPORTS by number of assets (NASSETS) matrix with portfolio weights. The weights sum to the value of a Budget constraint, which is usually 1. (See <code>holdings2weights</code> for information about budget constraints.)
Prices	NASSETS vector of prices.

Description

`Holdings = weights2holdings(Values, Weights, Prices)` converts portfolio values and weights into portfolio holdings.

`Holdings` is a NPORTS-by-NASSETS matrix containing the holdings of NPORTS portfolios that contain NASSETS assets.

Note This function does not create round-lot positions. `Holdings` are floating-point values.

See Also

`holdings2weights`

willad

Williams Accumulation/Distribution line

Syntax

```
wadl = willad(highp, lowp, closep)
```

```
wadl = willad([highp lowp closep])
```

```
wadlts = willad(tsobj)
```

```
wadlts = willad(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

highp	High price (vector)
lowp	Low price (vector)
closep	Closing price (vector)
tsobj	Time series object

Description

`wadl = willad(highp, lowp, closep)` computes the Williams Accumulation/Distribution line for a set of stock price data. The prices needed for this function are the high (`highp`), low (`lowp`), and closing (`closep`) prices. All three are required.

`wadl = willad([highp lowp closep])` accepts a three-column matrix of prices as input. The first column contains the high prices, the second contains the low prices, and the third contains the closing prices.

`wadlts = willad(tsobj)` computes the Williams Accumulation/Distribution line for a set of stock price data contained in the financial time series object `tsobj`. The object must contain the high, low, and closing prices needed for this function. The function

assumes that the series are named `High`, `Low`, and `Close`. All three are required. `wadlts` is a financial time series object with the same dates as `tsobj` and a single data series named `WillAD`.

`wadlts = willad(tsobj, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

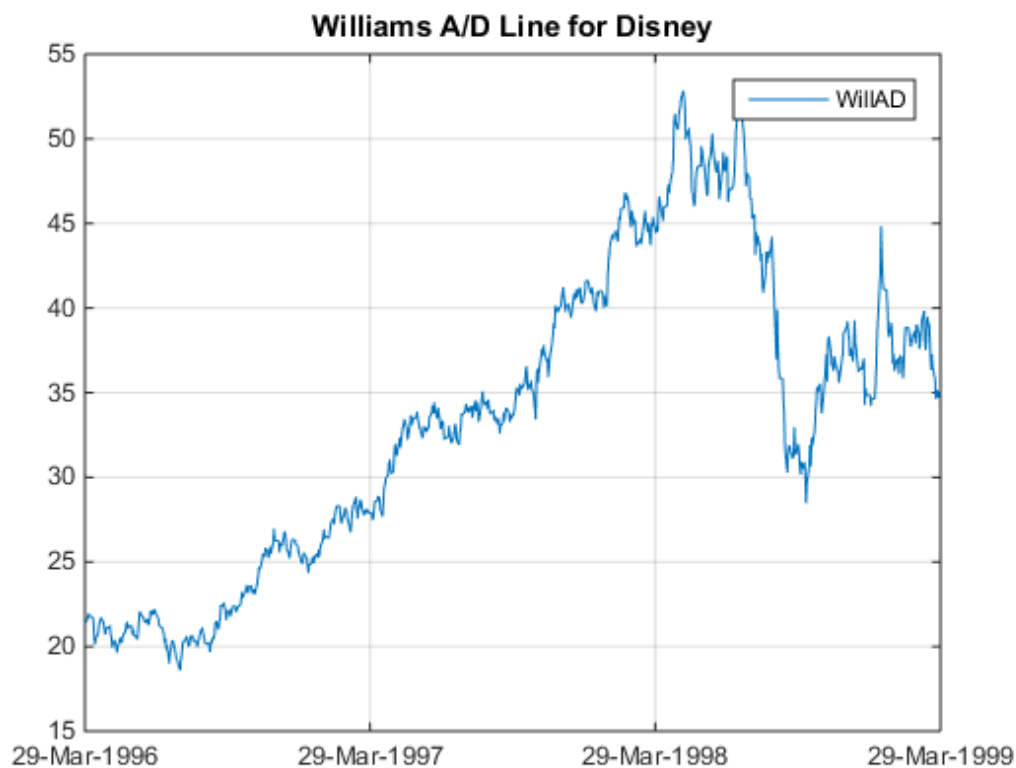
Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Williams A/D Line

This example shows how to compute the Williams A/D line for Disney stock and plot the results.

```
load disney.mat
dis_Willad = willad(dis);
plot(dis_Willad)
title('Williams A/D Line for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 314 - 315.

See Also

adline | adosc | willpctr

willpctr

Williams %R

Syntax

```
wpctr = willpctr(highp, lowp, closep, nperiods)
```

```
wpctr = willpctr([highp, lowp, closep], nperiods)
```

```
wpctrts = willpctr(tsobj)
```

```
wpctrts = willpctr(tsobj, nperiods)
```

```
wpctrts = willpctr(tsobj, nperiods, ParameterName, ParameterValue,  
... )
```

Arguments

highp	High price (vector)
lowp	Low price (vector)
closep	Closing price (vector)
nperiods	Number of periods (scalar). Default = 14.
tsobj	Financial time series object

Description

`wpctr = willpctr(highp, lowp, closep, nperiods)` calculates the Williams %R values for the given set of stock prices for a specified number of periods `nperiods`. The stock prices needed are the high (`highp`), low (`lowp`), and closing (`closep`) prices. `wpctr` is a vector that represents the Williams %R values from the stock data.

`wpctr = willpctr([highp, lowp, closep], nperiods)` accepts the price input as a three-column matrix representing the high, low, and closing prices, in that order.

`wpctrts = willpctr(tsobj)` calculates the Williams %R values for the financial time series object `tsobj`. The object must contain at least three data series named `High` (high prices), `Low` (low prices), and `Close` (closing prices). `wpctrts` is a financial time series object with the same dates as `tsobj` and a single data series named `WillPctR`.

`wpctrts = willpctr(tsobj, nperiods)` calculates the Williams %R values for the financial time series object `tsobj` for `nperiods` periods.

`wpctrts = willpctr(tsobj, nperiods, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

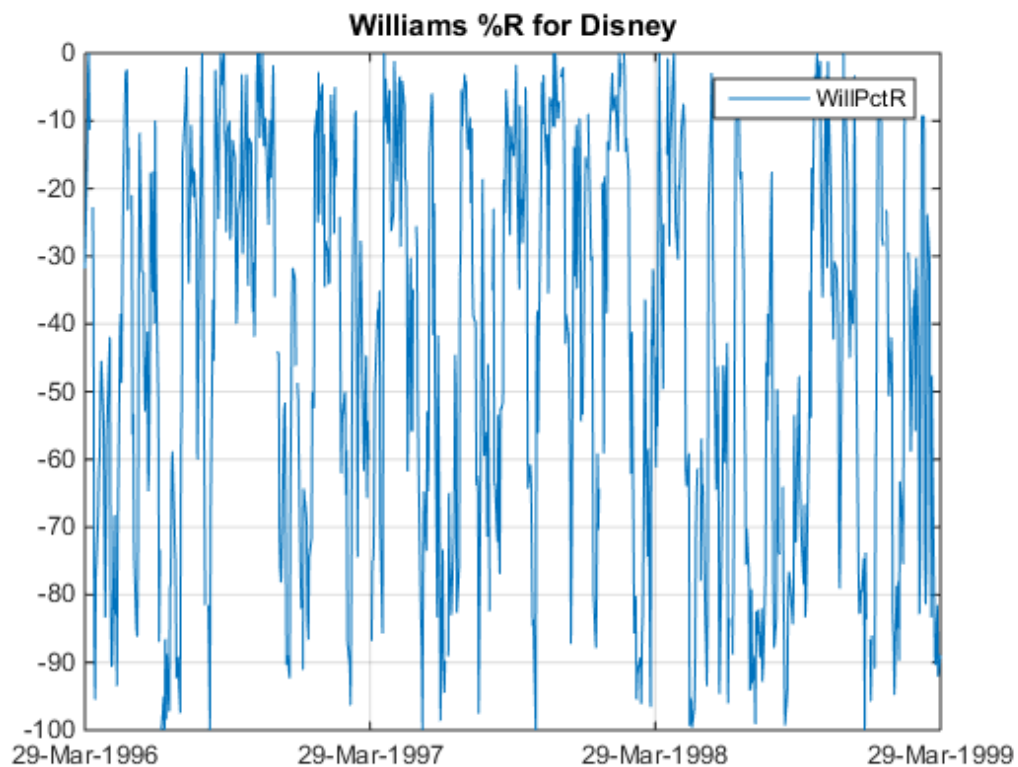
Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Williams %R Values

This example shows how to compute the Williams %R values for Disney stock and plot the results.

```
load disney.mat
dis_Wpctr = willpctr(dis);
plot(dis_Wpctr)
title('Williams %R for Disney')
```



References

Achelis, Steven B., *Technical Analysis from A to Z*, Second printing, McGraw-Hill, 1995, pp. 316 - 317.

See Also

stochosc | willad

wrkdydif

Number of working days between dates

Syntax

```
Days = wrkdydif(StartDate, EndDate, Holidays)
```

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.
Holidays	A vector containing values for the number of Holidays between the two dates.

Description

`Days = wrkdydif(StartDate, EndDate, Holidays)` returns the number of working days between dates `StartDate` and `EndDate` inclusive. `Holidays` is the number of holidays between the given dates, an integer. Enter dates as serial date numbers or date strings.

Examples

```
Days = wrkdydif('9/1/2000', '9/11/2000', 1)
```

or

```
Days = wrkdydif(730730, 730740, 1)
```

returns

```
Days =  
    6
```

See Also

busdate | datewrkdy | days360 | days365 | daysact | daysdif | holidays |
yearfrac

x2mdate

Excel serial date number to MATLAB serial date number

Syntax

`MATLABDate = x2mdate(ExcelDateNumber, Convention)`

Arguments

ExcelDateNumber	A vector or scalar of Excel serial date numbers.
Convention	<p>(Optional) Excel date system. A vector or scalar. When <code>Convention = 0</code> (default), the Excel 1900 date system is in effect. When <code>Convention = 1</code>, the Excel 1904 date system is used.</p> <p>In the Excel 1900 date system, the Excel serial date number 1 corresponds to January 1, 1900 A.D. In the Excel 1904 date system, date number 0 is January 1, 1904 A.D.</p> <p>Due to a software limitation in Excel software, the year 1900 is considered a leap year. As a result, all DATEVALUEs reported by Excel software between Jan. 1, 1900 and Feb. 28, 1900 (inclusive) differ from the values reported by 1. For example:</p> <ul style="list-style-type: none"> • In Excel software, Jan. 1, 1900 = 1 • In MATLAB, Jan. 1, 1900 = 2

Vector arguments must have consistent dimensions.

Description

`DateNumber = x2mdate(ExcelDateNumber, Convention)` converts Excel serial date numbers to MATLAB serial date numbers. MATLAB date numbers start with

1 = January 1, 0000 A.D., hence there is a difference of 693960 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with Spreadsheet Link EX software.

Examples

Given Excel date numbers in the 1904 system

```
ExDates = [35423 35788 36153];
```

convert them to MATLAB date numbers

```
MATLABDate = x2mdate(ExDates, 1)
```

```
MATLABDate =
```

```
730845      731210      731575
```

and then to date strings.

```
datestr(MATLABDate)
```

```
ans =
```

```
25-Dec-2000
```

```
25-Dec-2001
```

```
25-Dec-2002
```

See Also

[datenum](#) | [datestr](#) | [m2xdate](#)

xirr

Internal rate of return for nonperiodic cash flow

Syntax

```
Return = xirr(CashFlow, CashFlowDates)
Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations,
Basis)
```

Description

`Return = xirr(CashFlow, CashFlowDates)` returns the internal rate of return for a schedule of nonperiodic cash flows.

`Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations, Basis)` returns the internal rate of return for a schedule of nonperiodic cash flows with optional inputs.

Input Arguments

CashFlow

A vector or matrix of cash flows. If `CashFlow` is a matrix, each column represents a separate stream of cash flows whose internal rate of return is calculated. The first cash flow of each stream is the initial investment, entered as a negative number.

CashFlowDates

(Required) A vector or matrix of serial date numbers the same size as `CashFlow`, or a cell array of date strings the same size as `CashFlow`. Each column of `CashFlowDate` represents the dates of the corresponding column of `CashFlow`.

Guess

The initial estimate of the internal rate of return. `Guess` is a scalar applied to all streams, or a vector the same length as the number of streams.

Default: 0.1 (10%)

MaxIterations

The positive integer number of iterations used by Newton's method to solve the internal rate of return. **MaxIterations** is a scalar applied to all streams, or a vector the same length as the number of streams.

Default: 50

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Default: 0

Output Arguments

Return

Vector of the annualized internal rate of return of each cash flow stream. A NaN indicates that a solution is not found.

Examples

Find the internal rate of return for an investment of \$10,000 that returns the following nonperiodic cash flow. The original investment is the first cash flow and is a negative number.

Cash Flow	Dates
(\$10000)	January 12, 2007
\$2500	February 14, 2008
\$2000	March 3, 2008
\$3000	June 14, 2008
\$4000	December 1, 2008

Calculate the internal rate of return for this nonperiodic cash flow:

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
CashFlowDates = [ '01/12/2007'
                  '02/14/2008'
                  '03/03/2008'
                  '06/14/2008'
                  '12/01/2008' ];
Return = xirr(CashFlow, CashFlowDates)
```

This returns:

```
Return =
    0.1006 (or 10.0644% per annum)
```

References

Brealey and Myers, *Principles of Corporate Finance*, McGraw-Hill Higher Education, Chapter 5, 2003.

Sharpe, William F., and Gordon J. Alexander, *Investments*. Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990.

See Also

fvvar | irr | mirr | pvvar

year

Year of date

Syntax

```
Year = year(Date)
```

```
Year = year(Date, F)
```

Description

`Year = year(Date)` returns the year of a serial date number or a date string.

`Year = year(Date, F)` returns the year, given a serial date number or date string, in a specified date format.

Examples

```
Year = year(731798.776)
```

or

```
Year = year('05-Aug-2003')
```

returns

```
Year =
```

```
    2003
```

You can also use the `F` argument to designate a country-specific date format:

```
Year = year('2003/08/05', 'yyyy/mm/dd')
```

```
Year =
```

```
    2003.00
```

See Also

datevec | day | month | yeardays

yeardays

Number of days in year

Syntax

Days = yeardays(Year, Basis)

Arguments

Year	Enter as a four-digit integer.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (BMA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ICMA)• 9 = actual/360 (ICMA)• 10 = actual/365 (ICMA)• 11 = 30/360E (ICMA)• 12 = actual/actual (ISDA)• 13 = BUS/252 For more information, see basis .

Description

`Days = yeardays(Year, Basis)` returns the number of days in the given year, based upon the day-count basis.

Examples

`Days = yeardays(2000)`

`Days =`

`366`

`Days = yeardays(2000, 1)`

`Days =`

`360`

See Also

`days360` | `days365` | `daysact` | `year` | `yearfrac`

yearfrac

Fraction of year between dates

Syntax

YearFraction = yearfrac(StartDate, EndDate, Basis)

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.
Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>

All specified arguments must be number of instruments (NUMINST)-by-1 or 1-by-NUMINST conforming vectors or scalar arguments.

Description

`YearFraction = yearfrac(StartDate, EndDate, Basis)` returns a fraction based on the number of days between dates `StartDate` and `EndDate` using the given day-count basis. Note, the number of days in a year (365 or 366) is equal to the number of days in the calendar year after the `StartDate`. If `EndDate` is earlier than `StartDate`, `Fraction` is negative.

Definitions

The difference between `yearfrac` and `date2time` is that `date2time` counts full periods as a whole integer, even if the number of actual days in the periods are different. `yearfrac` does not count full periods. For example,

```
yearfrac('1/1/2000', '1/1/2001', 9)
```

```
ans =
```

```
1.0167
```

`yearfrac` for `Basis 9` (ACT/360 ISMA) will calculate $366/360 = 1.0167$. So, even if the dates have the same month and date, with a difference of 1 in the year, the returned value may not be exactly 1. On the other hand, `date2time` will calculate one full year period:

```
date2time('1/1/2000', '1/1/2001', 1, 9)
```

```
ans =
```

```
1
```

Examples

Compute `yearfrac` When the Calendar Year After the `StartDate` is Not a Leap Year

Given a `Basis` of 0 and a `Basis` of 1, compute `yearfrac`.

Define the `StartDate` and `EndDate` using a `Basis` of 0.

```
YearFraction = yearfrac('14 mar 01', '14 sep 01', 0)
```

```
YearFraction =
```

```
0.5041
```

Define the `StartDate` and `EndDate` using a `Basis` of 1.

```
YearFraction = yearfrac('14 mar 01', '14 sep 01', 1)
```

```
YearFraction =
```

```
0.5000
```

Compute `yearfrac` When the Calendar Year After the `StartDate` is a Leap Year

Given a `Basis` of 0, compute `yearfrac` when the calendar after `StartDate` is in a leap year.

Define the `StartDate` and `EndDate` using a `Basis` of 0.

```
yearFraction = yearfrac(' 14 mar 03', '14 sep 03', 0)
```

```
yearFraction =
```

```
0.5027
```

There are 184 days between March 14 and September 14, and the calendar year after the `StartDate` is a leap year, so `yearfrac` returns $184/366 = 0.5027$.

See Also

`days360` | `date2time` | `days365` | `daysact` | `daysdif` | `months` | `wrkdydif` | `year` | `yeardays`

ylddisc

Yield of discounted security

Syntax

Yield = ylddisc(Settle, Maturity, Face, Price, Basis)

Arguments

Settle	Settlement date. Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Maturity date. Enter as serial date number or date string.
Face	Redemption (par, face) value.
Price	Discounted price of the security.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252

For more information, see **basis**.

Description

`Yield = ylddisc(Settle, Maturity, Face, Price, Basis)` finds the yield of a discounted security.

Examples

Find the Yield of a Discounted Security

This example shows how to find the yield of the following discounted security.

```
Settle = '10/14/2000';  
Maturity = '03/17/2001';  
Face = 100;  
Price = 96.28;  
Basis = 2;
```

```
Yield = ylddisc(Settle, Maturity, Face, Price, Basis)
```

```
Yield =  
  
    0.0903
```

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 1.

See Also

`acrudisc` | `bndprice` | `bndyield` | `prdisc` | `yldmat` | `yldtbill`

yldmat

Yield with interest at maturity

Syntax

```
Yield = yldmat(Settle, Maturity, Issue, Face, Price,
CouponRate, Basis)
```

Arguments

Settle	Settlement date. Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Maturity date. Enter as serial date number or date string.
Issue	Issue date. Enter as serial date number or date string.
Face	Redemption (par, face) value.
Price	Price of the security.
CouponRate	Coupon rate. Enter as decimal fraction.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

For more information, see **basis**.

Description

`Yield = yldmat(Settle, Maturity, Issue, Face, Price, CouponRate, Basis)` returns the yield of a security paying interest at maturity.

Examples

Find the Yield of a Security Paying Interest at Maturity

This example shows how to find the yield of a security paying interest at maturity for the following.

```
Settle = '02/07/2000';  
Maturity = '04/13/2000';  
Issue = '10/11/1999';  
Face = 100;  
Price = 99.98;  
CouponRate = 0.0608;  
Basis = 1;
```

```
Yield = yldmat(Settle, Maturity, Issue, Face, Price, ...  
CouponRate, Basis)
```

```
Yield =
```

```
0.0607
```

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 3.

See Also

acrubond | bndprice | bndyield | prmat | ylddisc | yldtbill

yldtbill

Yield of Treasury bill

Syntax

```
Yield = yldtbill(Settle, Maturity, Face, Price)
```

Arguments

Settle	Settlement date. Enter as serial date number or date string. Settle must be earlier than Maturity .
Maturity	Maturity date. Enter as serial date number or date string.
Face	Redemption (par, face) value.
Price	Price of the Treasury bill.

Description

`Yield = yldtbill(Settle, Maturity, Face, Price)` returns the yield for a Treasury bill.

Examples

Find the Yield for a Treasury Bill

This example shows how to return the yield for a Treasury bill, given the settlement date of a Treasury bill is February 10, 2000, the maturity date is August 6, 2000, the par value is \$1000, and the price is \$981.36.

```
Yield = yldtbill('2/10/2000', '8/6/2000', 1000, 981.36)
```

```
Yield =
```

0.0384

References

Bodie, Kane, and Marcus, *Investments*, pages 41-43.

See Also

beytbill | bndyield | prtbill | yldmat

zbtprice

Zero curve bootstrapping from coupon bond data given price

Syntax

```
[ZeroRates, CurveDates] = zbtprice(Bonds, Prices, Settle,
OutputCompounding)
```

Arguments

Bonds	<p>Coupon bond information used to generate the zero curve. An n-by-2 to n-by-6 matrix where each row describes a bond. The first two columns are required; the rest are optional but must be added in order. All rows in Bonds must have the same number of columns.</p> <p>Columns are [Maturity CouponRate Face Period Basis EndMonthRule] where</p>	
	Maturity	Maturity date of the bond, as a serial date number. Use datenum to convert date strings to serial date numbers.
	CouponRate	Coupon rate of the bond, as a decimal fraction.
	Face	(Optional) Redemption or face value of the bond. Default = 100.
	Period	(Optional) Coupons per year of the bond, as an integer. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
	Basis	<p>(Optional) Day-count basis of the bond:</p> <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360

		<ul style="list-style-type: none"> • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
	EndMonthRule	(Optional) End-of-month flag. This flag applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore flag, meaning that a bond's coupon payment date is always the same day of the month. 1 = set flag (default), meaning that a bond's coupon payment date is always the last day of the month.
Prices	Column vector containing the clean price (price without accrued interest) of each bond in Bonds , respectively. The number of rows (n) must match the number of rows in Bonds .	
Settle	Settlement date, as a scalar serial date number. This represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.	
OutputCompounding	(Optional) Scalar that sets the compounding frequency per year for the output zero rates in ZeroRates . Allowed values are:	
	1	Annual compounding
	2	Semiannual compounding (default)

	3	Compounding three times per year
	4	Quarterly compounding
	6	Bimonthly compounding
	12	Monthly compounding
	-1	Continuous compounding

Description

[ZeroRates, CurveDates] = `zbtprice`(Bonds, Prices, Settle, OutputCompounding) uses the bootstrap method to return a zero curve given a portfolio of coupon bonds and their prices. A zero curve consists of the yields to maturity for a portfolio of theoretical zero-coupon bonds that are derived from the input **Bonds** portfolio. The bootstrap method that this function uses does *not* require alignment among the cash-flow dates of the bonds in the input portfolio. It uses theoretical par bond arbitrage and yield interpolation to derive all zero rates; specifically, the interest rates for cash flows are determined using linear interpolation. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

ZeroRates	<p>An m-by-1 vector of decimal fractions that are the implied zero rates for each point along the investment horizon represented by CurveDates; m is the number of bonds of unique maturity dates. In aggregate, the rates in ZeroRates constitute a zero curve.</p> <p>If more than one bond has the same maturity date, <code>zbtprice</code> returns the mean zero rate for that maturity. Any rates before the first maturity are assumed to be equal to the rate at the first maturity, that is, the curve is assumed to be flat before the first maturity.</p>
CurveDates	<p>An m-by-1 vector of unique maturity dates (as serial date numbers) that correspond to the zero rates in ZeroRates; m is the number of bonds of different maturity dates. These dates begin with the earliest maturity date and end with the latest maturity date Maturity in the Bonds matrix.</p>

Examples

Compute a Zero Curve Given a Portfolio of Coupon Bonds and Their Prices

Given data and prices for 12 coupon bonds, two with the same maturity date, and given the common settlement date.

```
Bonds = [datenum('6/1/1998') 0.0475 100 2 0 0;
          datenum('7/1/2000') 0.06 100 2 0 0;
          datenum('7/1/2000') 0.09375 100 6 1 0;
          datenum('6/30/2001') 0.05125 100 1 3 1;
          datenum('4/15/2002') 0.07125 100 4 1 0;
          datenum('1/15/2000') 0.065 100 2 0 0;
          datenum('9/1/1999') 0.08 100 3 3 0;
          datenum('4/30/2001') 0.05875 100 2 0 0;
          datenum('11/15/1999') 0.07125 100 2 0 0;
          datenum('6/30/2000') 0.07 100 2 3 1;
          datenum('7/1/2001') 0.0525 100 2 3 0;
          datenum('4/30/2002') 0.07 100 2 0 0];
```

```
Prices = [99.375;
          99.875;
          105.75 ;
          96.875;
          103.625;
          101.125;
          103.125;
          99.375;
          101.0 ;
          101.25 ;
          96.375;
          102.75 ];
```

```
Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve.

```
OutputCompounding = 2;
```

Execute the function `zbtprice` which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date.

```
[ZeroRates, CurveDates] = zbtprice(Bonds, Prices, Settle,...
OutputCompounding)
```

ZeroRates =

0.0616
0.0609
0.0658
0.0590
0.0647
0.0655
0.0606
0.0601
0.0642
0.0621
0.0627

CurveDates =

729907
730364
730439
730500
730667
730668
730971
731032
731033
731321
731336

References

Fabozzi, Frank J. “The Structure of Interest Rates.” Ch. 6 in Fabozzi, Frank J. and T. Dossa Fabozzi, eds. *The Handbook of Fixed Income Securities*. 4th ed. New York: Irwin Professional Publishing. 1995.

McEnally, Richard W. and James V. Jordan. “The Term Structure of Interest Rates.” Ch. 37 in Fabozzi and Fabozzi, *ibid.*

Das, Satyajit. “Calculating Zero Coupon Rates.” *Swap and Derivative Financing*. Appendix to Ch. 8, pp. 219-225. New York: Irwin Professional Publishing. 1994.

See Also
zbtyield

zbtyield

Zero curve bootstrapping from coupon bond data given yield

Syntax

```
[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle,
OutputCompounding)
```

Arguments

Bonds	Coupon bond information used to generate the zero curve. An n-by-2 to n-by-6 matrix where each row describes a bond. The first two columns are required; the rest are optional but must be added in order. All rows in Bonds must have the same number of columns. Columns are [Maturity CouponRate Face Period Basis EndMonthRule] where	
	Maturity	Maturity date of the bond, as a serial date number. Use <code>datenum</code> to convert date strings to serial date numbers.
	CouponRate	Coupon rate of the bond, as a decimal fraction.
	Face	(Optional) Redemption or face value of the bond. Default = 100.
	Period	(Optional) Coupons per year of the bond, as an integer. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
	Basis	(Optional) Day-count basis of the bond. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365

		<ul style="list-style-type: none"> • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
	EndMonthRule	(Optional) End-of-month flag. This flag applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore flag, meaning that a bond's coupon payment date is always the same day of the month. 1 = set flag (default), meaning that a bond's coupon payment date is always the last day of the month.
Yields	Column vector containing the yield to maturity of each bond in Bonds , respectively. The number of rows (n) must match the number of rows in Bonds . Yield to maturity must be compounded semiannually.	
Settle	Settlement date, as a scalar serial date number. This represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.	
OutputCompounding	(Optional) Scalar that sets the compounding frequency per year for the output zero rates in ZeroRates . Allowed values are:	
	1	Annual compounding
	2	Semiannual compounding (default)
	3	Compounding three times per year

	4	Quarterly compounding
	6	Bimonthly compounding
	12	Monthly compounding
	-1	Continuous compounding

Description

[ZeroRates, CurveDates] = `zbtyield`(Bonds, Yields, Settle, OutputCompounding) uses the bootstrap method to return a zero curve given a portfolio of coupon bonds and their yields. A zero curve consists of the yields to maturity for a portfolio of theoretical zero-coupon bonds that are derived from the input **Bonds** portfolio. The bootstrap method that this function uses does *not* require alignment among the cash-flow dates of the bonds in the input portfolio. It uses theoretical par bond arbitrage and yield interpolation to derive all zero rates; specifically, the interest rates for cash flows are determined using linear interpolation. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

ZeroRates	<p>An m-by-1 vector of decimal fractions that are the implied zero rates for each point along the investment horizon represented by CurveDates; m is the number of bonds of different maturity dates. In aggregate, the rates in ZeroRates constitute a zero curve.</p> <p>If more than one bond has the same maturity date, <code>zbtyield</code> returns the mean zero rate for that maturity. Any rates before the first maturity are assumed to be equal to the rate at the first maturity, that is, the curve is assumed to be flat before the first maturity.</p>
CurveDates	<p>An m-by-1 vector of unique maturity dates (as serial date numbers) that correspond to the zero rates in ZeroRates; m is the number of bonds of different maturity dates. These dates begin with the earliest maturity date and end with the latest maturity date Maturity in the Bonds matrix. Use <code>datestr</code> to convert serial date numbers to date strings.</p>

Examples

Compute a Zero Curve Given a Portfolio of Coupon Bonds and Their Yields

Given data and yields to maturity for 12 coupon bonds, two with the same maturity date; and given the common settlement date.

```
Bonds = [datenum('6/1/1998')    0.0475    100    2    0    0;
          datenum('7/1/2000')    0.06        100    2    0    0;
          datenum('7/1/2000')    0.09375    100    6    1    0;
          datenum('6/30/2001')   0.05125    100    1    3    1;
          datenum('4/15/2002')   0.07125    100    4    1    0;
          datenum('1/15/2000')   0.065        100    2    0    0;
          datenum('9/1/1999')    0.08        100    3    3    0;
          datenum('4/30/2001')   0.05875    100    2    0    0;
          datenum('11/15/1999')  0.07125    100    2    0    0;
          datenum('6/30/2000')   0.07        100    2    3    1;
          datenum('7/1/2001')    0.0525     100    2    3    0;
          datenum('4/30/2002')   0.07        100    2    0    0];
```

```
Yields = [0.0616
          0.0605
          0.0687
          0.0612
          0.0615
          0.0591
          0.0603
          0.0608
          0.0655
          0.0646
          0.0641
          0.0627];
```

```
Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve.

```
OutputCompounding = 2;
```

Execute the function `zbyield` which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date.

```
[ZeroRates, CurveDates] = zbyield(Bonds, Yields, Settle, ...
```

OutputCompounding)

ZeroRates =

0.0616
0.0603
0.0657
0.0590
0.0649
0.0650
0.0606
0.0611
0.0643
0.0614
0.0627

CurveDates =

729907
730364
730439
730500
730667
730668
730971
731032
731033
731321
731336

More About

- “Term Structure of Interest Rates” on page 2-38

References

Fabozzi, Frank J. “The Structure of Interest Rates.” Ch. 6 in Fabozzi, Frank J. and T. Dossa Fabozzi, eds. *The Handbook of Fixed Income Securities*. 4th ed. New York: Irwin Professional Publishing. 1995.

McEnally, Richard W. and James V. Jordan. "The Term Structure of Interest Rates." Ch. 37 in Fabozzi and Fabozzi, *ibid.*

Das, Satyajit. "Calculating Zero Coupon Rates." *Swap and Derivative Financing*. Appendix to Ch. 8, pp. 219-225. New York: Irwin Professional Publishing. 1994.

See Also

zbtprice

zero2disc

Discount curve given zero curve

Syntax

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates, Settle, Compounding, Basis)
```

Arguments

ZeroRates	Number of bonds (NUMBONDS)-by-1 vector of annualized zero rates, as decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by CurveDates.	
CurveDates	NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates.	
Settle	Serial date number that is the common settlement date for the zero rates; that is, the settlement date for the bonds from which the zero curve was bootstrapped.	
Compounding	(Optional) Scalar that indicates the compounding frequency per year used for annualizing the input zero rates in ZeroRates. Allowed values are:	
	1	Annual compounding
	2	Semiannual compounding (default)
	3	Compounding three times per year
	4	Quarterly compounding
	6	Bimonthly compounding
	12	Monthly compounding
	365	Daily compounding
	-1	Continuous compounding
Basis	(Optional) Day-count basis used for annualizing the input zero rates.	

	<ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
--	--

Description

[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates, Settle, Compounding, Basis) returns a discount curve given a zero curve and its maturity dates.

DiscRates	A NUMBONDS-by-1 vector of discount factors, as decimal fractions. In aggregate, the factors in constitute a discount curve for the investment horizon represented by CurveDates .
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the discount rates. This vector is the same as the input vector CurveDates .

Examples

Compute a Discount Curve Given a Zero Curve and Maturity Dates

Given a zero curve over a set of maturity dates and a settlement date.

```
ZeroRates = [0.0464  
             0.0509  
             0.0524  
             0.0525  
             0.0531  
             0.0525  
             0.0530  
             0.0531  
             0.0549  
             0.0536];
```

```
CurveDates = [datenum('06-Nov-2000')  
              datenum('11-Dec-2000')  
              datenum('15-Jan-2001')  
              datenum('05-Feb-2001')  
              datenum('04-Mar-2001')  
              datenum('02-Apr-2001')  
              datenum('30-Apr-2001')  
              datenum('25-Jun-2001')  
              datenum('04-Sep-2001')  
              datenum('12-Nov-2001')];
```

```
Settle = datenum('03-Nov-2000');
```

The zero curve is compounded daily on an actual/365 basis.

```
Compounding = 365;  
Basis = 3;
```

Execute the function `zero2disc` which returns the discount curve `DiscRates` at the maturity dates `CurveDates`.

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates, ...  
Settle, Compounding, Basis)
```

```
DiscRates =
```

```
0.9996
0.9947
0.9896
0.9866
0.9826
0.9787
0.9745
0.9665
0.9552
0.9466
```

```
CurveDates =
```

```
730796
730831
730866
730887
730914
730943
730971
731027
731098
731167
```

For readability, **ZeroRates** and **DiscRates** are shown here only to the basis point. However, MATLAB® software computed them at full precision. If you enter **ZeroRates** as shown, **DiscRates** may differ due to rounding.

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

disc2zero

zero2fwd

Forward curve given zero curve

Syntax

[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, Settle, Compounding, Basis)

Arguments

ZeroRates	Number of bonds (NUMBONDS)-by-1 vector of annualized zero rates, as decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by CurveDates. The first element pertains to forward rates from the settlement date to the first curve date.	
CurveDates	NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates.	
Settle	Serial date number that is the common settlement date for the zero rates.	
Compounding	(Optional) Scalar that sets the compounding frequency per year used to annualize the input zero rates and the output implied forward rates. Allowed values are:	
	1	Annual compounding
	2	Semiannual compounding (default)
	3	Compounding three times per year
	4	Quarterly compounding
	6	Bimonthly compounding
	12	Monthly compounding
	365	Daily compounding
	-1	Continuous compounding
Basis	(Optional) Day-count basis used to construct the input zero and output implied forward rate curves.	

	<ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
--	--

Description

[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, Settle, Compounding, Basis) returns an implied forward rate curve given a zero curve and its maturity dates.

ForwardRates	A NUMBONDS-by-1 vector of decimal fractions. In aggregate, the rates in ForwardRates constitute a forward curve over the dates in CurveDates .
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the forward rates in. This vector is the same as the input vector CurveDates .

Examples

Compute an Implied Forward Rate Curve Given a Zero Curve and Maturity Dates

Given a zero curve over a set of maturity dates, a settlement date, and a compounding rate, compute the forward rate curve.

```
ZeroRates = [0.0458  
            0.0502  
            0.0518  
            0.0519  
            0.0524  
            0.0519  
            0.0523  
            0.0525  
            0.0541  
            0.0529];
```

```
CurveDates = [datenum('06-Nov-2000')  
             datenum('11-Dec-2000')  
             datenum('15-Jan-2001')  
             datenum('05-Feb-2001')  
             datenum('04-Mar-2001')  
             datenum('02-Apr-2001')  
             datenum('30-Apr-2001')  
             datenum('25-Jun-2001')  
             datenum('04-Sep-2001')  
             datenum('12-Nov-2001')];
```

```
Settle = datenum('03-Nov-2000');  
Compounding = 1;
```

Execute the function `zero2fwd` to return the forward rate curve `ForwardRates` at the maturity dates `CurveDates`.

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, ...  
Settle, Compounding)
```

```
ForwardRates =  
  
    0.0458  
    0.0506  
    0.0535
```

```
0.0522  
0.0541  
0.0498  
0.0544  
0.0531  
0.0594  
0.0476
```

```
CurveDates =
```

```
730796  
730831  
730866  
730887  
730914  
730943  
730971  
731027  
731098  
731167
```

For readability, `ZeroRates` and `ForwardRates` are shown here only to the basis point. However, MATLAB® software computed them at full precision. If you enter `ZeroRates` as shown, `ForwardRates` may differ due to rounding.

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

`fwd2zero`

zero2pyld

Par yield curve given zero curve

Syntax

```
[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle,
Compounding, Basis, InputCompounding)
```

Arguments

ZeroRates	A number of bonds (NUMBONDS)-by-1 vector of annualized zero rates, as decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by CurveDates.	
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates.	
Settle	A serial date number that is the common settlement date for the zero rates.	
Compounding	(Optional) Scalar value representing the periodicity in which the output par rates are compounded when annualized. Allowed values are:	
	1	Annual compounding
	2	Semiannual compounding (default)
	3	Compounding three times per year
	4	Quarterly compounding
	6	Bimonthly compounding
	12	Monthly compounding
Basis	(Optional) Day-count basis used to annualize the implied zero rates.	
	<ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) 	

	<ul style="list-style-type: none"> • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252 <p>For more information, see basis.</p>
InputCompounding	(Optional) Scalar value representing the periodicity in which the input zero rates were compounded when annualized. The default is the value for Compounding.

Description

[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle, Compounding, Basis, InputCompounding) returns a par yield curve given a zero curve and its maturity dates.

ParRates	A NUMBONDS-by-1 vector of annualized par yields, as decimal fractions. (Par yields = coupon rates.) In aggregate, the yield rates in ParRates constitute a par yield curve for the investment horizon represented by CurveDates.
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the par yield rates. This vector is the same as the input vector CurveDates.

Examples

Given

- A zero curve over a set of maturity dates and
- A settlement date
- Annual compounding for the input zero curve and monthly compounding for the output par rates

compute a par yield curve.

```
ZeroRates = [0.0457
             0.0487
             0.0506
             0.0507
             0.0505
             0.0504
             0.0506
             0.0516
             0.0539
             0.0530];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
InputCompounding = 1;
Compounding = 12;

[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, ...
Settle, Compounding, [], InputCompounding)

ParRates =

    0.0479
    0.0511
    0.0530
    0.0531
    0.0526
    0.0524
    0.0525
```

```
0.0534  
0.0555  
0.0543
```

```
CurveDates =
```

```
730796  
730831  
730866  
730887  
730914  
730943  
730971  
731027  
731098  
731167
```

For readability, `ZeroRates` and `ParRates` are shown only to the basis point. However, MATLAB software computed them at full precision. If you enter `ZeroRates` as shown, `ParRates` may differ due to rounding.

More About

- “Term Structure of Interest Rates” on page 2-38

See Also

`pyld2zero`

score

Compute credit scores for given data

Syntax

```
Scores = score(sc)
```

```
Scores = score(sc, data)
```

```
[Scores,Points] = score(sc)
```

```
[Scores,Points] = score(sc,data)
```

Description

`Scores = score(sc)` computes the credit scores for the `creditscorecard` object's training data. This data can be a “training” or a “live” dataset. If the data input argument is not explicitly provided, the `score` function determines scores for the existing `creditscorecard` object's data.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are to be reported separately or spread across predictors. Missing data translates into NaN values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

`Scores = score(sc, data)` computes the credit scores for the given input data. This data can be a “training” or a “live” dataset.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are to be reported separately or spread across predictors. Missing data translates into NaN values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

`[Scores,Points] = score(sc)` computes the credit scores and points for the given data. If the data input argument is not explicitly provided, the `score` function determines scores for the existing `creditscorecard` object's data.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are to be reported separately or spread across predictors. Missing data translates into NaN values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

`[Scores,Points] = score(sc,data)` computes the credit scores and points for the given input data. This data can be a “training” or a “live” dataset.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are to be reported separately or spread across predictors. Missing data translates into NaN values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

Examples

Obtain Scores for Training Data

This example shows how to use `score` to obtain scores for the training data.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406

7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Score training data using the `score` function without an optional input for `data`. By default, it returns unscaled scores. For brevity, only the first ten scores are displayed.

```
Scores = score(sc);
disp(Scores(1:10))
```

```
1.4260
1.3630
0.9314
1.5300
1.2919
1.9908
-0.2524
1.1070
1.0299
0.9386
```

Scale scores and display both points and scores for each individual in the training data (for brevity, only the first ten rows are displayed). For other scaling methods, and other options for formatting points and scores, use the `formatpoints` function.

```
sc = formatpoints(sc, 'WorstAndBestScores', [300 850]);
[Scores, Points] = score(sc);
disp(Scores(1:10))
disp(Points(1:10,:))
```

```
661.8861
652.7861
590.3886
676.9272
642.5001
743.5370
419.2469
615.7666
604.6170
591.4234
```

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBalance
104.58	60.523	53.889	133.08	157.06	88.42	64.326
106.32	83.494	110.83	133.08	66.313	88.42	64.326
75.497	60.523	110.83	86.719	157.06	35.427	64.326
104.58	83.494	110.83	133.08	58.71	88.42	97.808
106.32	83.494	110.83	133.08	58.71	88.42	61.643
106.32	83.494	110.83	133.08	157.06	88.42	64.326
45.251	83.494	53.889	20.236	66.313	88.42	61.643
104.58	119.71	110.83	133.08	66.313	35.427	45.818
104.58	60.523	53.889	133.08	66.313	88.42	97.808
104.58	83.494	53.889	133.08	66.313	88.42	61.643

Score a New Dataset

This example shows how to use `score` to obtain scores for a new dataset (for example, a validation or a test dataset) using the optional `'data'` input in the `score` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

For the purpose of illustration, suppose that a few rows from the original data are our "new" data. Use the optional `data` input argument in the `score` function to obtain the scores for the `newdata`.

```
newdata = data(10:20,:);
Scores = score(sc,newdata)
```

Scores =

```
0.9386
0.8434
1.4298
0.6443
0.6360
1.3682
0.7356
0.7441
0.5934
1.6470
0.8160
```

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

data — Dataset to be scored

table

Dataset to be scored, specified as a table where each row corresponds to individual observations. The data must contain columns for each of the predictors in the `creditscorecard` object.

Output Arguments

Scores — Scores for each observation

vector

Scores for each observation, returned as a vector.

Points — Points per predictor for each observation

table

Points per predictor for each observation, returned as a table.

More About

Algorithms

The score of an individual i is given by the formula

$$\text{Score}(i) = \text{Shift} + \text{Slope} * (b_0 + b_1 * \text{WOE}_1(i) + b_2 * \text{WOE}_2(i) + \dots + b_p * \text{WOE}_p(i))$$

where b_j is the coefficient of the j -th variable in the model, and $\text{WOE}_j(i)$ is the Weight of Evidence (WOE) value for the i -th individual corresponding to the j -th model variable. Shift and Slope are scaling constants further discussed below. The scaling constant can be controlled with `formatpoints`.

If the data for individual i is in the i -th row of a given dataset, to compute a score, the $\text{data}(i,j)$ is binned using existing binning maps, and converted into a corresponding Weight of Evidence value $\text{WOE}_j(i)$. Using the model coefficients, the unscaled score is computed as

$$s = b_0 + b_1 * \text{WOE}_1(i) + \dots + b_p * \text{WOE}_p(i).$$

For simplicity, assume in the description above that the j -th variable in the model is the j -th column in the data input, although, in general, the order of variables in a given dataset does not have to match the order of variables in the model, and the dataset could have additional variables that are not used in the model.

The formatting options can be controlled using `formatpoints`.

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

autobinning | bindata | bininfo | creditscorecard | displaypoints |
fitmodel | formatpoints | modifybins | plotbins | setmodel

formatpoints

Format scorecard points and scaling

Syntax

```
sc = formatpoints(sc,Name,Value)
```

Description

`sc = formatpoints(sc,Name,Value)` modifies the scorecard points and scaling using optional name-value pair arguments. For example, use optional name-value pair arguments to change the scaling of the scores or the rounding of the points.

Examples

Scale Points Using Worst and Best Scores

This example shows how to use `formatpoints` to scale by providing the `Worst` and `Best` score values. By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding `CustIncome`, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0

2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732
'CustAge'	'[37,43)'	0.030121
'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851

'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf, 33000)'	-0.31036
'CustIncome'	'[33000, 38000)'	0.14951
'CustIncome'	'[38000, 42000)'	-0.019644
'CustIncome'	'[42000, 47000)'	0.25907
'CustIncome'	'[47000, Inf]'	0.47018
'TmWBank'	'[-Inf, 12)'	-0.18156
'TmWBank'	'[12, 23)'	-0.044231
'TmWBank'	'[23, 36)'	0.0083609
'TmWBank'	'[36, 55)'	0.14256
'TmWBank'	'[55, Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128
'AMBalance'	'[-Inf, 259.78)'	0.22621
'AMBalance'	'[259.78, 558.88)'	0.48049
'AMBalance'	'[558.88, 875.97)'	-0.023939
'AMBalance'	'[875.97, 1254.28)'	-0.0053831
'AMBalance'	'[1254.28, Inf]'	-0.1334

```
MinScore =
```

```
-1.0772
```

```
MaxScore =
```

```
2.7272
```

Scale by providing the 'Worst' and 'Best' score values. The range provided below is a common score range. Display the points information again to verify that they are now scaled and also display the scaled minimum and maximum scores.

```
sc = formatpoints(sc, 'WorstAndBestScores', [300 850]);
[PointsInfo, MinScore, MaxScore] = displaypoints(sc)
```

```
PointsInfo =
```

Predictors	Bin	Points
------------	-----	--------

'CustAge'	'[-Inf,37)'	45.251
'CustAge'	'[37,43)'	69.459
'CustAge'	'[43,48)'	75.497
'CustAge'	'[48,54)'	104.58
'CustAge'	'[54,Inf]'	106.32
'ResStatus'	'Home Owner'	83.494
'ResStatus'	'Tenant'	60.523
'ResStatus'	'Other'	119.71
'EmpStatus'	'Employed'	110.83
'EmpStatus'	'Unknown'	53.889
'CustIncome'	'[-Inf,33000)'	20.236
'CustIncome'	'[33000,38000)'	86.719
'CustIncome'	'[38000,42000)'	62.264
'CustIncome'	'[42000,47000)'	102.56
'CustIncome'	'[47000,Inf]'	133.08
'TmWBank'	'[-Inf,12)'	38.856
'TmWBank'	'[12,23)'	58.71
'TmWBank'	'[23,36)'	66.313
'TmWBank'	'[36,55)'	85.714
'TmWBank'	'[55,Inf]'	157.06
'OtherCC'	'No'	35.427
'OtherCC'	'Yes'	88.42
'AMBalance'	'[-Inf,259.78)'	97.808
'AMBalance'	'[259.78,558.88)'	134.57
'AMBalance'	'[558.88,875.97)'	61.643
'AMBalance'	'[875.97,1254.28)'	64.326
'AMBalance'	'[1254.28,Inf]'	45.818

MinScore =

300

MaxScore =

850

As expected, the values of `MinScore` and `MaxScore` correspond to the desired worst and best scores.

Scale Points Using Shift and Slope

This example shows how to use `formatpoints` to scale by providing the `Shift` and `Slope` values. By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points by the `formatpoints` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding `CustIncome`, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding `TmWBank`, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding `AMBBalance`, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding `EmpStatus`, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding `CustAge`, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding `OtherCC`, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding `ResStatus`, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501

ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732
'CustAge'	'[37,43)'	0.030121
'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851
'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf,33000)'	-0.31036
'CustIncome'	'[33000,38000)'	0.14951
'CustIncome'	'[38000,42000)'	-0.019644
'CustIncome'	'[42000,47000)'	0.25907
'CustIncome'	'[47000,Inf]'	0.47018
'TmWBank'	'[-Inf,12)'	-0.18156
'TmWBank'	'[12,23)'	-0.044231
'TmWBank'	'[23,36)'	0.0083609
'TmWBank'	'[36,55)'	0.14256
'TmWBank'	'[55,Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128

```
'AMBalance'      '[-Inf,259.78) '      0.22621
'AMBalance'      '[259.78,558.88) '    0.48049
'AMBalance'      '[558.88,875.97) '    -0.023939
'AMBalance'      '[875.97,1254.28) '   -0.0053831
'AMBalance'      '[1254.28,Inf] '      -0.1334
```

```
MinScore =
```

```
-1.0772
```

```
MaxScore =
```

```
2.7272
```

Scale by providing the 'Shift' and 'Slope' values. In this example, there is an arbitrary choice of shift and slope. Display the points information again to verify that they are now scaled and also display the scaled minimum and maximum scores.

```
sc = formatpoints(sc, 'ShiftAndSlope', [300 6]);
[PointsInfo, MinScore, MaxScore] = displaypoints(sc)
```

```
PointsInfo =
```

Predictors	Bin	Points
'CustAge'	'[-Inf,37) '	42.033
'CustAge'	'[37,43) '	43.038
'CustAge'	'[43,48) '	43.288
'CustAge'	'[48,54) '	44.496
'CustAge'	'[54,Inf] '	44.568
'ResStatus'	'Home Owner'	43.62
'ResStatus'	'Tenant'	42.667
'ResStatus'	'Other'	45.124
'EmpStatus'	'Employed'	44.755
'EmpStatus'	'Unknown'	42.392
'CustIncome'	'[-Inf,33000) '	40.995
'CustIncome'	'[33000,38000) '	43.754
'CustIncome'	'[38000,42000) '	42.739
'CustIncome'	'[42000,47000) '	44.412
'CustIncome'	'[47000,Inf] '	45.678

'TmWBank'	'[-Inf, 12)'	41.768
'TmWBank'	'[12, 23)'	42.592
'TmWBank'	'[23, 36)'	42.907
'TmWBank'	'[36, 55)'	43.712
'TmWBank'	'[55, Inf]'	46.674
'OtherCC'	'No'	41.626
'OtherCC'	'Yes'	43.825
'AMBalance'	'[-Inf, 259.78)'	44.214
'AMBalance'	'[259.78, 558.88)'	45.74
'AMBalance'	'[558.88, 875.97)'	42.714
'AMBalance'	'[875.97, 1254.28)'	42.825
'AMBalance'	'[1254.28, Inf]'	42.057

MinScore =

293.5369

MaxScore =

316.3630

Scale Points Using Points, Odds Levels, and PDO

This example shows how to use `formatpoints` to scale by providing the points, odds levels, and PDO (points to double the odds). By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points by the `formatpoints` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-05
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732
'CustAge'	'[37,43)'	0.030121

'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851
'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf,33000)'	-0.31036
'CustIncome'	'[33000,38000)'	0.14951
'CustIncome'	'[38000,42000)'	-0.019644
'CustIncome'	'[42000,47000)'	0.25907
'CustIncome'	'[47000,Inf]'	0.47018
'TmWBank'	'[-Inf,12)'	-0.18156
'TmWBank'	'[12,23)'	-0.044231
'TmWBank'	'[23,36)'	0.0083609
'TmWBank'	'[36,55)'	0.14256
'TmWBank'	'[55,Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128
'AMBalance'	'[-Inf,259.78)'	0.22621
'AMBalance'	'[259.78,558.88)'	0.48049
'AMBalance'	'[558.88,875.97)'	-0.023939
'AMBalance'	'[875.97,1254.28)'	-0.0053831
'AMBalance'	'[1254.28,Inf]'	-0.1334

MinScore =

-1.0772

MaxScore =

2.7272

Scale by providing the points, odds levels, and PDO (points to double the odds). Suppose that you want a score of 500 points to have odds of 2 (twice as likely to be good than to be bad) and that the odds double every 50 points (so that 550 points would have odds of 4).

```
sc = formatpoints(sc, 'PointsOddsAndPDO', [500 2 50]);
[PointsInfo, MinScore, MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf, 37)'	54.38
'CustAge'	'[37, 43)'	66.458
'CustAge'	'[43, 48)'	69.471
'CustAge'	'[48, 54)'	83.984
'CustAge'	'[54, Inf]'	84.851
'ResStatus'	'Home Owner'	73.461
'ResStatus'	'Tenant'	62
'ResStatus'	'Other'	91.533
'EmpStatus'	'Employed'	87.102
'EmpStatus'	'Unknown'	58.69
'CustIncome'	'[-Inf, 33000)'	41.898
'CustIncome'	'[33000, 38000)'	75.071
'CustIncome'	'[38000, 42000)'	62.869
'CustIncome'	'[42000, 47000)'	82.974
'CustIncome'	'[47000, Inf]'	98.202
'TmWBank'	'[-Inf, 12)'	51.189
'TmWBank'	'[12, 23)'	61.095
'TmWBank'	'[23, 36)'	64.889
'TmWBank'	'[36, 55)'	74.569
'TmWBank'	'[55, Inf]'	110.17
'OtherCC'	'No'	49.478
'OtherCC'	'Yes'	75.919
'AMBalance'	'[-Inf, 259.78)'	80.603
'AMBalance'	'[259.78, 558.88)'	98.946
'AMBalance'	'[558.88, 875.97)'	62.559
'AMBalance'	'[875.97, 1254.28)'	63.897
'AMBalance'	'[1254.28, Inf]'	54.663

MinScore =

372.2979

MaxScore =

646.7232

Report Base Points Separately

This example shows how to use `formatpoints` to separate the base points from the rest of the points assigned to each predictor variable. The `formatpoints` name-value pair argument `'BasePoints'` serves this purpose.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05

OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732
'CustAge'	'[37,43)'	0.030121
'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851
'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf,33000)'	-0.31036
'CustIncome'	'[33000,38000)'	0.14951
'CustIncome'	'[38000,42000)'	-0.019644
'CustIncome'	'[42000,47000)'	0.25907
'CustIncome'	'[47000,Inf]'	0.47018
'TmWBank'	'[-Inf,12)'	-0.18156
'TmWBank'	'[12,23)'	-0.044231
'TmWBank'	'[23,36)'	0.0083609
'TmWBank'	'[36,55)'	0.14256
'TmWBank'	'[55,Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128
'AMBalance'	'[-Inf,259.78)'	0.22621
'AMBalance'	'[259.78,558.88)'	0.48049
'AMBalance'	'[558.88,875.97)'	-0.023939
'AMBalance'	'[875.97,1254.28)'	-0.0053831

```
'AMBalance'      '[1254.28, Inf]'      -0.1334
```

```
MinScore =
```

```
-1.0772
```

```
MaxScore =
```

```
2.7272
```

By setting the name-value pair argument `BasePoints` to true, the points information table reports the base points separately in the first row. The minimum and maximum possible scores are not affected by this option.

```
sc = formatpoints(sc, 'BasePoints', true);
[PointsInfo, MinScore, MaxScore] = displaypoints(sc)
```

```
PointsInfo =
```

Predictors	Bin	Points
'BasePoints'	'BasePoints'	0.70329
'CustAge'	'[-Inf, 37)'	-0.23779
'CustAge'	'[37, 43)'	-0.070349
'CustAge'	'[43, 48)'	-0.028583
'CustAge'	'[48, 54)'	0.17261
'CustAge'	'[54, Inf]'	0.18463
'ResStatus'	'Home Owner'	0.026731
'ResStatus'	'Tenant'	-0.13216
'ResStatus'	'Other'	0.27726
'EmpStatus'	'Employed'	0.21583
'EmpStatus'	'Unknown'	-0.17804
'CustIncome'	'[-Inf, 33000)'	-0.41083
'CustIncome'	'[33000, 38000)'	0.049039
'CustIncome'	'[38000, 42000)'	-0.12011
'CustIncome'	'[42000, 47000)'	0.1586
'CustIncome'	'[47000, Inf]'	0.36971
'TmWBank'	'[-Inf, 12)'	-0.28203
'TmWBank'	'[12, 23)'	-0.1447
'TmWBank'	'[23, 36)'	-0.092109

```
'TmWBank'      '[36,55) '      0.042085
'TmWBank'      '[55,Inf] '     0.53561
'OtherCC'      'No'           -0.30574
'OtherCC'      'Yes'          0.060808
'AMBalance'    '[-Inf,259.78) ' 0.12574
'AMBalance'    '[259.78,558.88) ' 0.38002
'AMBalance'    '[558.88,875.97) ' -0.12441
'AMBalance'    '[875.97,1254.28) ' -0.10585
'AMBalance'    '[1254.28,Inf] ' -0.23387
```

```
MinScore =
```

```
-1.0772
```

```
MaxScore =
```

```
2.7272
```

Round Points

This example shows how to use `formatpoints` to round points. Rounding is usually applied after scaling, otherwise, if the points for a particular predictor are all in a small range, rounding could cause the rounded points for different bins to be the same. Also, rounding all the points may slightly change the minimum and maximum total points.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
```

2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732
'CustAge'	'[37,43)'	0.030121
'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851

```
'ResStatus'      'Home Owner'      0.1272
'ResStatus'      'Tenant'          -0.031688
'ResStatus'      'Other'           0.37773
'EmpStatus'      'Employed'        0.3163
'EmpStatus'      'Unknown'         -0.077573
'CustIncome'     '[-Inf,33000)'   -0.31036
'CustIncome'     '[33000,38000)'  0.14951
'CustIncome'     '[38000,42000)' -0.019644
'CustIncome'     '[42000,47000)'  0.25907
'CustIncome'     '[47000,Inf]'    0.47018
'TmWBank'        '[-Inf,12)'      -0.18156
'TmWBank'        '[12,23)'        -0.044231
'TmWBank'        '[23,36)'        0.0083609
'TmWBank'        '[36,55)'        0.14256
'TmWBank'        '[55,Inf]'       0.63608
'OtherCC'        'No'             -0.20527
'OtherCC'        'Yes'            0.16128
'AMBalance'     '[-Inf,259.78)'  0.22621
'AMBalance'     '[259.78,558.88)' 0.48049
'AMBalance'     '[558.88,875.97)' -0.023939
'AMBalance'     '[875.97,1254.28)' -0.0053831
'AMBalance'     '[1254.28,Inf]'  -0.1334
```

```
MinScore =
```

```
-1.0772
```

```
MaxScore =
```

```
2.7272
```

Scale points, and display the points information. By default, no rounding is applied.

```
sc = formatpoints(sc, 'WorstAndBestScores', [300 850]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo =
```

Predictors	Bin	Points
------------	-----	--------

'CustAge'	'[-Inf,37)'	45.251
'CustAge'	'[37,43)'	69.459
'CustAge'	'[43,48)'	75.497
'CustAge'	'[48,54)'	104.58
'CustAge'	'[54,Inf]'	106.32
'ResStatus'	'Home Owner'	83.494
'ResStatus'	'Tenant'	60.523
'ResStatus'	'Other'	119.71
'EmpStatus'	'Employed'	110.83
'EmpStatus'	'Unknown'	53.889
'CustIncome'	'[-Inf,33000)'	20.236
'CustIncome'	'[33000,38000)'	86.719
'CustIncome'	'[38000,42000)'	62.264
'CustIncome'	'[42000,47000)'	102.56
'CustIncome'	'[47000,Inf]'	133.08
'TmWBank'	'[-Inf,12)'	38.856
'TmWBank'	'[12,23)'	58.71
'TmWBank'	'[23,36)'	66.313
'TmWBank'	'[36,55)'	85.714
'TmWBank'	'[55,Inf]'	157.06
'OtherCC'	'No'	35.427
'OtherCC'	'Yes'	88.42
'AMBalance'	'[-Inf,259.78)'	97.808
'AMBalance'	'[259.78,558.88)'	134.57
'AMBalance'	'[558.88,875.97)'	61.643
'AMBalance'	'[875.97,1254.28)'	64.326
'AMBalance'	'[1254.28,Inf]'	45.818

Use the name-value pair argument `Round` to apply rounding for all points and then display the points information again.

```
sc = formatpoints(sc,'Round','AllPoints');
PointsInfo = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	45
'CustAge'	'[37,43)'	69
'CustAge'	'[43,48)'	75
'CustAge'	'[48,54)'	105

'CustAge'	'[54,Inf]'	106
'ResStatus'	'Home Owner'	83
'ResStatus'	'Tenant'	61
'ResStatus'	'Other'	120
'EmpStatus'	'Employed'	111
'EmpStatus'	'Unknown'	54
'CustIncome'	'[-Inf,33000)'	20
'CustIncome'	'[33000,38000)'	87
'CustIncome'	'[38000,42000)'	62
'CustIncome'	'[42000,47000)'	103
'CustIncome'	'[47000,Inf]'	133
'TmWBank'	'[-Inf,12)'	39
'TmWBank'	'[12,23)'	59
'TmWBank'	'[23,36)'	66
'TmWBank'	'[36,55)'	86
'TmWBank'	'[55,Inf]'	157
'OtherCC'	'No'	35
'OtherCC'	'Yes'	88
'AMBalance'	'[-Inf,259.78)'	98
'AMBalance'	'[259.78,558.88)'	135
'AMBalance'	'[558.88,875.97)'	62
'AMBalance'	'[875.97,1254.28)'	64
'AMBalance'	'[1254.28,Inf]'	46

Scores for Missing or Out-of-Range Data

This example shows how to use `formatpoints` to score missing or out-of-range data. When data is scored, some observations can be either missing (NaN, or `undefined`) or out of range. You will need to decide whether or not points are assigned to these cases. Use the name-value pair argument `Missing` to do so.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-05
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Suppose missing observations are added to the data that you want to score. Notice that by default, the points and score assigned to the missing value is NaN.

```
newdata = data(1:10, :);
newdata.CustAge(1) = NaN;
[Scores, Points] = score(sc, newdata)
```

Scores =

```
NaN
1.3630
0.9314
```

```

1.5300
1.2919
1.9908
-0.2524
1.1070
1.0299
0.9386

```

Points =

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBAl
NaN	-0.031688	-0.077573	0.47018	0.63608	0.16128	-0.009
0.2851	0.1272	0.3163	0.47018	0.0083609	0.16128	-0.009
0.071887	-0.031688	0.3163	0.14951	0.63608	-0.20527	-0.009
0.27308	0.1272	0.3163	0.47018	-0.044231	0.16128	0.2
0.2851	0.1272	0.3163	0.47018	-0.044231	0.16128	-0.02
0.2851	0.1272	0.3163	0.47018	0.63608	0.16128	-0.009
-0.13732	0.1272	-0.077573	-0.31036	0.0083609	0.16128	-0.02
0.27308	0.37773	0.3163	0.47018	0.0083609	-0.20527	-0
0.27308	-0.031688	-0.077573	0.47018	0.0083609	0.16128	0.2
0.27308	0.1272	-0.077573	0.47018	0.0083609	0.16128	-0.02

Use the name-value pair argument `Missing` to replace NaN with zero.

```

sc = formatpoints(sc, 'Missing', 'ZeroPoints');
[Scores,Points] = score(sc,newdata)

```

Scores =

```

1.1529
1.3630
0.9314
1.5300
1.2919
1.9908
-0.2524
1.1070
1.0299
0.9386

```

Points =

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBAL
0	-0.031688	-0.077573	0.47018	0.63608	0.16128	-0.003609
0.2851	0.1272	0.3163	0.47018	0.0083609	0.16128	-0.003609
0.071887	-0.031688	0.3163	0.14951	0.63608	-0.20527	-0.003609
0.27308	0.1272	0.3163	0.47018	-0.044231	0.16128	0.20527
0.2851	0.1272	0.3163	0.47018	-0.044231	0.16128	-0.003609
0.2851	0.1272	0.3163	0.47018	0.63608	0.16128	-0.003609
-0.13732	0.1272	-0.077573	-0.31036	0.0083609	0.16128	-0.003609
0.27308	0.37773	0.3163	0.47018	0.0083609	-0.20527	-0.003609
0.27308	-0.031688	-0.077573	0.47018	0.0083609	0.16128	0.20527
0.27308	0.1272	-0.077573	0.47018	0.0083609	0.16128	-0.003609

Use the name-value pair argument `Missing` to replace the missing value with the minimum points for the predictor that has the missing values, 'CustAge'.

```
sc = formatpoints(sc, 'Missing', 'MinPoints');
[Scores, Points] = score(sc, newdata)
```

Scores =

```
1.0156
1.3630
0.9314
1.5300
1.2919
1.9908
-0.2524
1.1070
1.0299
0.9386
```

Points =

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBAL
-0.13732	-0.031688	-0.077573	0.47018	0.63608	0.16128	-0.003609

```

0.2851      0.1272      0.3163      0.47018      0.0083609      0.16128      -0.009
0.071887   -0.031688     0.3163      0.14951      0.63608      -0.20527     -0.009
0.27308    0.1272      0.3163      0.47018      -0.044231     0.16128      0.2
0.2851      0.1272      0.3163      0.47018      -0.044231     0.16128      -0.02
0.2851      0.1272      0.3163      0.47018      0.63608      0.16128      -0.009
-0.13732   0.1272     -0.077573   -0.31036     0.0083609     0.16128      -0.02
0.27308    0.37773      0.3163      0.47018      0.0083609     -0.20527      -0
0.27308    -0.031688    -0.077573   0.47018      0.0083609     0.16128      0.2
0.27308    0.1272     -0.077573   0.47018      0.0083609     0.16128      -0.02

```

Use the name-value pair argument `Missing` to replace the missing value with the maximum points for the predictor that has the missing values, `'CustAge'`.

```

sc = formatpoints(sc, 'Missing', 'MaxPoints');
[Scores,Points] = score(sc,newdata)

```

Scores =

```

1.4380
1.3630
0.9314
1.5300
1.2919
1.9908
-0.2524
1.1070
1.0299
0.9386

```

Points =

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBa1a
0.2851	-0.031688	-0.077573	0.47018	0.63608	0.16128	-0.009
0.2851	0.1272	0.3163	0.47018	0.0083609	0.16128	-0.009
0.071887	-0.031688	0.3163	0.14951	0.63608	-0.20527	-0.009
0.27308	0.1272	0.3163	0.47018	-0.044231	0.16128	0.2
0.2851	0.1272	0.3163	0.47018	-0.044231	0.16128	-0.02
0.2851	0.1272	0.3163	0.47018	0.63608	0.16128	-0.009
-0.13732	0.1272	-0.077573	-0.31036	0.0083609	0.16128	-0.02
0.27308	0.37773	0.3163	0.47018	0.0083609	-0.20527	-0

```

0.27308 -0.031688 -0.077573 0.47018 0.0083609 0.16128 0.2
0.27308 0.1272 -0.077573 0.47018 0.0083609 0.16128 -0.0

```

Verify that the minimum and maximum points assigned to the missing data correspond to the minimum and maximum points for 'CustAge'. The points for 'CustAge' are reported in the first five rows of the points information table.

```

PointsInfo = displaypoints(sc);
PointsInfo(1:5,:)
min(PointsInfo.Points(1:5))
max(PointsInfo.Points(1:5))

```

```
ans =
```

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732
'CustAge'	'[37,43)'	0.030121
'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851

```
ans =
```

```
-0.1373
```

```
ans =
```

```
0.2851
```

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model
creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, `...`, `NameN`, `ValueN`.

Example: `sc = formatpoints(sc, 'BasePoints', true, 'Round', 'AllPoints', 'WorstAndBestScores', [100, 700])`

Note: `ShiftAndSlope`, `PointsOddsAndPDO`, and `WorstAndBestScores` are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (`BasePoints`, `Missing`, and `Round`) are not scaling methods and can be used together or with any one of the three scaling methods.

'BasePoints' — Indicator for separating base points

`false` (default) | logical scalar

Indicator for separating base points, specified as a logical scalar. If `true`, the scorecard explicitly separates base points. If `false`, the base points are spread across all variables in the `creditscorecard` object.

Data Types: `char`

'Missing' — Indicator for points assigned to missing or out-of-range information when scoring

`NoScore` (default) | string with values `NoScore`, `ZeroPoints`, `MinPoints`, and `MaxPoints`

Indicator for points assigned to missing or out-of-range information when scoring, specified as a string with a value for `NoScore`, `ZeroPoints`, `MinPoints`, or `MaxPoints`, where:

- `NoScore` — Missing and out-of-range data do not get points assigned and points are set to `NaN`. Also, the total score is set to `NaN`.

- **ZeroPoints** — Missing or out-of-range data get zero points. This accounts to ignoring that predictor and adding up all other predictors points.
- **MinPoints** — Missing or out-of-range data get the minimum possible points for that predictor. This penalizes the score if higher scores are better.
- **MaxPoints** — Missing or out-of-range data get the maximum possible points for that predictor. This penalizes the score if lower scores are better.

Data Types: char

'Round' — Indicator whether to round points or scores

'None' (default) | string with values 'AllPoints', 'FinalScore'

Indicator whether to round points or scores, specified as a string with values 'AllPoints', 'FinalScore' or 'None', where:

- **None** — No rounding is applied.
- **AllPoints** — Apply rounding to each predictor's points before adding up the total score.
- **FinalScore** — Round the final score only (rounding is applied after all points are added up).

Data Types: char

'ShiftAndSlope' — Indicator for shift and slope scaling parameters

[0, 1] (default) | numeric array with two elements [Shift, Slope]

Indicator for shift and slope scaling parameters for the credit scorecard, specified using numeric array with two elements [Shift, Slope]. Slope cannot be zero. The ShiftAndSlope values are used scale the scoring model.

Note: ShiftAndSlope, PointsOddsAndPDO, and WorstAndBestScores are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (BasePoints, Missing, and Round) are not scaling methods and can be used together or with any one of the three scaling methods.

To remove a previous scaling and revert to unscaled scores, set ShiftAndSlope to [0, 1].

Data Types: double

'PointsOddsAndPDO' — Indicator for target points for given odds and double odds level

numeric array with three elements [Points,Odds,PDO]

Indicator for target points (Points) for a given odds level (Odds) and the desired number of points to double the odds (PDO), specified using numeric array with three elements [Points,Odds,PDO]. Odds must be a positive number. The PointsOddsAndPDO values are used to find scaling parameters for the scoring model.

Note: The points to double the odds (PDO) may be positive or negative, depending on whether higher scores mean lower risk, or vice versa.

ShiftAndSlope, PointsOddsAndPDO, and WorstAndBestScores are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (BasePoints, Missing, and Round) are not scaling methods and can be used together or with any one of the three scaling methods.

To remove a previous scaling and revert to unscaled scores, set ShiftAndSlope to [0,1].

Data Types: double

'WorstAndBestScores' — Indicator for worst (highest risk) and best (lowest risk) scores in scorecard

numeric array with two elements [WorstScore,BestScore]

Indicator for worst (highest risk) and best (lowest risk) scores in the scorecard, specified as a numeric array with two elements [WorstScore,BestScore]. WorstScore and BestScore must be different values. These WorstAndBestScores values are used to find scaling parameters for the scoring model.

Note: WorstScore means the riskiest score, and its value could be lower or higher than the 'best' score. In other words, the 'minimum' score may be the 'worst' score or the 'best' score, depending on the desired scoring scale.

ShiftAndSlope, PointsOddsAndPDO, and WorstAndBestScores are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (BasePoints, Missing, and Round) are not scaling methods and can be used together or with any one of the three scaling methods.

To remove a previous scaling and revert to unscaled scores, set ShiftAndSlope to [0,1].

Data Types: double

Output Arguments

sc — Credit scorecard model
creditscorecard object

Credit scorecard model returned as an updated `creditscorecard` object. For more information on using the `creditscorecard` object, see [Using creditscorecard Objects](#).

More About

Algorithms

The score of an individual i is given by the formula

$$\text{Score}(i) = \text{Shift} + \text{Slope} * (b_0 + b_1 * \text{WOE}_1(i) + b_2 * \text{WOE}_2(i) + \dots + b_p * \text{WOE}_p(i))$$

where b_j is the coefficient of the j th variable in the model, and $\text{WOE}_j(i)$ is the Weight of Evidence (WOE) value for the i th individual corresponding to the j th model variable. Shift and Slope are scaling constants further discussed below. The scaling constant can be controlled with `formatpoints`.

If the data for individual i is in the i -th row of a given dataset, to compute a score, the $\text{data}(i,j)$ is binned using existing binning maps, and converted into a corresponding Weight of Evidence value $\text{WOE}_j(i)$. Using the model coefficients, the unscaled score is computed as

$$s = b_0 + b_1 * \text{WOE}_1(i) + \dots + b_p * \text{WOE}_p(i).$$

For simplicity, assume in the description above that the j -th variable in the model is the j -th column in the data input, although, in general, the order of variables in a given dataset does not have to match the order of variables in the model, and the dataset could have additional variables that are not used in the model.

The formatting options can be controlled using `formatpoints`. When the base points are reported separately (see the `formatpoints` parameter `BasePoints`), the base points are given by

$$\text{Base Points} = \text{Shift} + \text{Slope} * b_0,$$

and the points for the j -th predictor, i -th row are given by

```
Points_ji = Slope*(bj*WOEj(i)).
```

By default, the base points are not reported separately, in which case

```
Points_ji = (Shift + Slope*b0)/p + B*(bj*WOEj(i)),
```

where p is the number of predictors in the scorecard model.

By default, no rounding is applied to the points by the score function (`Round` is `None`). If `Round` is set to `AllPoints` using `formatpoints`, then the points for individual i for variable j are given by

```
points if rounding is 'AllPoints': round( Points_ji )
```

and, if base points are reported separately, the are also rounded. This yields integer-valued points per predictor, hence also integer-valued scores. If `Round` is set to `FinalScore` using `formatpoints`, then the points per predictor are not rounded, and only the final score is rounded

```
score if rounding is 'FinalScore': round(Score(i)).
```

Regarding the scaling parameters, the `Shift` parameter, and the `Slope` parameter can be set directly with the `ShiftAndSlope` parameter of `formatpoints`. Alternatively, you can use the `formatpoints` parameter for `WorstAndBestScores`. In this case, the parameters `Shift` and `Slope` are found internally by solving the system

```
Shift + Slope*smin = WorstScore,  
Shift + Slope*smax = BestScore,
```

where `WorstScore` and `BestScore` are the first and second elements in the `formatpoints` parameter for `WorstAndBestScores` and `smin` and `smax` are the minimum and maximum possible unscaled scores:

```
smin = b0 + min(b1*WOE1) + ... +min(bp*WOEp),  
smax = b0 + max(b1*WOE1) + ... +max(bp*WOEp).
```

A third alternative to scale scores is the `PointsOddsAndPDO` parameter in `formatpoints`. In this case, assume that the unscaled score s gives the log-odds for a row, and the `Shift` and `Slope` parameters are found by solving the following system

```
Points = Shift + Slope*log(Odds)  
Points + PDO = Shift + Slope*log(2*Odds)
```

where `Points`, `Odds`, and `PDO` ("points to double the odds") are the first, second, and third elements in the `PointsOddsAndPDO` parameter.

Whenever a given dataset has a missing or out-of-range value data (i,j), the points for predictor j , for individual i , are set to NaN by default, which results in a missing score for that row (a NaN score). Using the `formatpoints` parameter for `Missing`, you can modify this behavior and set the points to zero, or the minimum points, or the maximum points for that predictor.

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `modifybins` | `plotbins` | `score` | `setmodel`

displaypoints

Return points per predictor per bin

Syntax

```
PointsInfo = displaypoints(sc)  
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

Description

`PointsInfo = displaypoints(sc)` returns a table of points for all bins of all predictor variables used in the `creditscorecard` object after a linear logistic regression model is fit using `fitmodel` to the Weight of Evidence data. The `PointsInfo` table displays information on the predictor name, bin labels, and the corresponding points per bin.

`[PointsInfo,MinScore,MaxScore] = displaypoints(sc)` returns a table of points for all bins of all predictor variables used in the `creditscorecard` object after a linear logistic regression model is fit (`fitmodel`) to the Weight of Evidence data. The `PointsInfo` table displays information on the predictor name, bin labels, and the corresponding points per bin and `displaypoints`. In addition, the optional `MinScore` and `MaxScore` values are returned.

Examples

Display Unscaled Points

This example shows how to use `displaypoints` after a model is fitted to compute the unscaled points per bin, for a given predictor in the `creditscorecard` model.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the 'IDVar' argument in the `creditscorecard` function to indicate that 'CustID' contains ID information and should not be included as a predictor variable.

```
load CreditCardData
```

```
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Display unscaled points for predictors retained in the fitting model.

```
PointsInfo = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf, 37)'	-0.13732
'CustAge'	'[37, 43)'	0.030121
'CustAge'	'[43, 48)'	0.071887
'CustAge'	'[48, 54)'	0.27308
'CustAge'	'[54, Inf]'	0.2851
'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf, 33000)'	-0.31036
'CustIncome'	'[33000, 38000)'	0.14951
'CustIncome'	'[38000, 42000)'	-0.019644
'CustIncome'	'[42000, 47000)'	0.25907
'CustIncome'	'[47000, Inf]'	0.47018
'TmWBank'	'[-Inf, 12)'	-0.18156
'TmWBank'	'[12, 23)'	-0.044231
'TmWBank'	'[23, 36)'	0.0083609
'TmWBank'	'[36, 55)'	0.14256
'TmWBank'	'[55, Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128
'AMBalance'	'[-Inf, 259.78)'	0.22621
'AMBalance'	'[259.78, 558.88)'	0.48049
'AMBalance'	'[558.88, 875.97)'	-0.023939
'AMBalance'	'[875.97, 1254.28)'	-0.0053831
'AMBalance'	'[1254.28, Inf]'	-0.1334

Display Scaled Points

This example shows how to use `formatpoints` after a model is fitted to format scaled points, and then use `displaypoints` to display the scaled points per bin, for a given predictor in the `creditscorecard` model.

Points become scaled when a range is defined. Specifically, a linear transformation from the unscaled to the scaled points is necessary. This transformation is defined either by supplying a shift and slope or by specifying the worst and best scores possible. (For more information, see `formatpoints`.)

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom
Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Use the `formatpoints` function to scale providing the 'Worst' and 'Best' score values. The range provided below is a common score range.

```
sc = formatpoints(sc, 'WorstAndBestScores', [300 850]);
```

Display the points information again to verify that the points are now scaled and also display the scaled minimum and maximum scores.

```
sc = formatpoints(sc, 'WorstAndBestScores', [300 850]);
[PointsInfo, MinScore, MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf, 37)'	45.251
'CustAge'	'[37, 43)'	69.459
'CustAge'	'[43, 48)'	75.497
'CustAge'	'[48, 54)'	104.58
'CustAge'	'[54, Inf]'	106.32
'ResStatus'	'Home Owner'	83.494
'ResStatus'	'Tenant'	60.523
'ResStatus'	'Other'	119.71
'EmpStatus'	'Employed'	110.83
'EmpStatus'	'Unknown'	53.889
'CustIncome'	'[-Inf, 33000)'	20.236
'CustIncome'	'[33000, 38000)'	86.719
'CustIncome'	'[38000, 42000)'	62.264
'CustIncome'	'[42000, 47000)'	102.56
'CustIncome'	'[47000, Inf]'	133.08
'TmWBank'	'[-Inf, 12)'	38.856
'TmWBank'	'[12, 23)'	58.71
'TmWBank'	'[23, 36)'	66.313
'TmWBank'	'[36, 55)'	85.714
'TmWBank'	'[55, Inf]'	157.06
'OtherCC'	'No'	35.427
'OtherCC'	'Yes'	88.42
'AMBBalance'	'[-Inf, 259.78)'	97.808
'AMBBalance'	'[259.78, 558.88)'	134.57
'AMBBalance'	'[558.88, 875.97)'	61.643
'AMBBalance'	'[875.97, 1254.28)'	64.326

```

    'AMBalance'      '[1254.28, Inf]'      45.818

MinScore =

    300

MaxScore =

    850

```

Notice that, as expected, the values of `MinScore` and `MaxScore` correspond to the worst and best possible scores.

Separate the Base Points From the Total Points

This example shows how to use `displaypoints` after a model is fitted to separate the base points from the rest of the points assigned to each predictor variable. The name-value pair argument `'BasePoints'` in the `formatpoints` function is a boolean that serves this purpose. By default, the base points are spread across all variables in the scorecard.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```

load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID');

```

Perform automatic binning to bin for all predictors.

```

sc = autobinning(sc);

```

Fit a linear regression model using default parameters.

```

sc = fitmodel(sc);

```

1. Adding `CustIncome`, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding `TmWBank`, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding `AMBalance`, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.00040561508

4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBALANCE	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Use the `formatpoints` function to separate the base points by providing the `'BasePoints'` name-value pair argument.

```
sc = formatpoints(sc, 'BasePoints', true);
```

Display the base points, separated out from the other points, for predictors retained in the fitting model.

```
PointsInfo = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'BasePoints'	'BasePoints'	0.70329
'CustAge'	'[-Inf,37)'	-0.23779

'CustAge'	'[37,43)'	-0.070349
'CustAge'	'[43,48)'	-0.028583
'CustAge'	'[48,54)'	0.17261
'CustAge'	'[54,Inf]'	0.18463
'ResStatus'	'Home Owner'	0.026731
'ResStatus'	'Tenant'	-0.13216
'ResStatus'	'Other'	0.27726
'EmpStatus'	'Employed'	0.21583
'EmpStatus'	'Unknown'	-0.17804
'CustIncome'	'[-Inf,33000)'	-0.41083
'CustIncome'	'[33000,38000)'	0.049039
'CustIncome'	'[38000,42000)'	-0.12011
'CustIncome'	'[42000,47000)'	0.1586
'CustIncome'	'[47000,Inf]'	0.36971
'TmWBank'	'[-Inf,12)'	-0.28203
'TmWBank'	'[12,23)'	-0.1447
'TmWBank'	'[23,36)'	-0.092109
'TmWBank'	'[36,55)'	0.042085
'TmWBank'	'[55,Inf]'	0.53561
'OtherCC'	'No'	-0.30574
'OtherCC'	'Yes'	0.060808
'AMBalance'	'[-Inf,259.78)'	0.12574
'AMBalance'	'[259.78,558.88)'	0.38002
'AMBalance'	'[558.88,875.97)'	-0.12441
'AMBalance'	'[875.97,1254.28)'	-0.10585
'AMBalance'	'[1254.28,Inf]'	-0.23387

Display Points After Modifying Bin Labels

This example shows how to use `displaypoints` after a model is fitted and the `modifybins` function is used to provide user-defined bin labels for a numeric predictor.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Use the `displaypoints` function to display point information.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

PointsInfo =

Predictors	Bin	Points
'CustAge'	'[-Inf,37)'	-0.13732

'CustAge'	'[37,43)'	0.030121
'CustAge'	'[43,48)'	0.071887
'CustAge'	'[48,54)'	0.27308
'CustAge'	'[54,Inf]'	0.2851
'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf,33000)'	-0.31036
'CustIncome'	'[33000,38000)'	0.14951
'CustIncome'	'[38000,42000)'	-0.019644
'CustIncome'	'[42000,47000)'	0.25907
'CustIncome'	'[47000,Inf]'	0.47018
'TmWBank'	'[-Inf,12)'	-0.18156
'TmWBank'	'[12,23)'	-0.044231
'TmWBank'	'[23,36)'	0.0083609
'TmWBank'	'[36,55)'	0.14256
'TmWBank'	'[55,Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128
'AMBalance'	'[-Inf,259.78)'	0.22621
'AMBalance'	'[259.78,558.88)'	0.48049
'AMBalance'	'[558.88,875.97)'	-0.023939
'AMBalance'	'[875.97,1254.28)'	-0.0053831
'AMBalance'	'[1254.28,Inf]'	-0.1334

MinScore =

-1.0772

MaxScore =

2.7272

Use the `modifybins` function to specify user-defined bin labels for `'CustAge'` so that the bin ranges are described in natural language.

```
labels = {'Up to 36', '37 to 42', '43 to 47', '48 to 53', 'At least 54'};
sc = modifybins(sc, 'CustAge', 'BinLabels', labels);
```

Rerun `displaypoints` to verify the updated bin labels.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo =
```

Predictors	Bin	Points
'CustAge'	'Up to 36'	-0.13732
'CustAge'	'37 to 42'	0.030121
'CustAge'	'43 to 47'	0.071887
'CustAge'	'48 to 53'	0.27308
'CustAge'	'At least 54'	0.2851
'ResStatus'	'Home Owner'	0.1272
'ResStatus'	'Tenant'	-0.031688
'ResStatus'	'Other'	0.37773
'EmpStatus'	'Employed'	0.3163
'EmpStatus'	'Unknown'	-0.077573
'CustIncome'	'[-Inf,33000)'	-0.31036
'CustIncome'	'[33000,38000)'	0.14951
'CustIncome'	'[38000,42000)'	-0.019644
'CustIncome'	'[42000,47000)'	0.25907
'CustIncome'	'[47000,Inf]'	0.47018
'TmWBank'	'[-Inf,12)'	-0.18156
'TmWBank'	'[12,23)'	-0.044231
'TmWBank'	'[23,36)'	0.0083609
'TmWBank'	'[36,55)'	0.14256
'TmWBank'	'[55,Inf]'	0.63608
'OtherCC'	'No'	-0.20527
'OtherCC'	'Yes'	0.16128
'AMBalance'	'[-Inf,259.78)'	0.22621
'AMBalance'	'[259.78,558.88)'	0.48049
'AMBalance'	'[558.88,875.97)'	-0.023939
'AMBalance'	'[875.97,1254.28)'	-0.0053831
'AMBalance'	'[1254.28,Inf]'	-0.1334

```
MinScore =
```

```
-1.0772
```

```
MaxScore =
```

```
2.7272
```


- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

Output Arguments

PointsInfo — One row per bin, per predictor, with the corresponding points

table

One row per bin, per predictor, with the corresponding points, returned as a table. For example:

Predictors	Bin	Points
Predictor_1	Bin_11	Points_11

Predictor_2	Bin_21	Points_21

Predictor_ <i>j</i>	Bin_ <i>ji</i>	Points_ <i>ji</i>

When base points are reported separately (see `formatpoints`), the first row of the returned `PointsInfo` table contains the base points.

MinScore — Minimum possible total score

scalar

Minimum possible total score, returned as a scalar.

Note: Minimum score is the lowest possible total score in the mathematical sense, independently of whether a low score means high risk or low risk.

MaxScore — Maximum possible total score

scalar

Maximum possible total score, returned as a scalar.

Note: Maximum score is the highest possible total score in the mathematical sense, independently of whether a high score means high risk or low risk.

More About

Algorithms

The points for predictor j and bin i are, by default, given by

$$\text{Points_ji} = (\text{Shift} + \text{Slope} * \text{b0}) / p + \text{Slope} * (\text{bj} * \text{WOEj}(i))$$

where bj is the model coefficient of predictor j , p is the number of predictors in the model, and $\text{WOEj}(i)$ is the Weight of Evidence (WOE) value for the i -th bin corresponding to the j -th model predictor. **Shift** and **Slope** are scaling constants.

When the base points are reported separately (see the `formatpoints` name-value pair argument `BasePoints`), the base points are given by

$$\text{Base Points} = \text{Shift} + \text{Slope} * \text{b0},$$

and the points for the j -th predictor, i -th row are given by

$$\text{Points_ji} = \text{Slope} * (\text{bj} * \text{WOEj}(i)).$$

By default, the base points are not reported separately.

The minimum and maximum scores are:

$$\begin{aligned} \text{MinScore} &= \text{Shift} + \text{Slope} * \text{b0} + \min(\text{Slope} * \text{b1} * \text{WOE1}) + \dots + \min(\text{Slope} * \text{bp} * \text{WOEp}), \\ \text{MaxScore} &= \text{Shift} + \text{Slope} * \text{b0} + \max(\text{Slope} * \text{b1} * \text{WOE1}) + \dots + \max(\text{Slope} * \text{bp} * \text{WOEp}). \end{aligned}$$

Use `formatpoints` to control the way points are scaled, rounded, and whether the base points are reported separately. See `formatpoints` for more information on format parameters and for details and formulas on these formatting options.

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using creditscorecard Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `fitmodel` |
`formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

fitmodel

Fit logistic regression model to Weight of Evidence (WOE) data

Syntax

```
sc = fitmodel(sc)

[sc,mdl] = fitmodel(sc)
[sc,mdl] = fitmodel( ____,Name,Value)
```

Description

`sc = fitmodel(sc)` fits a logistic regression model to the Weight of Evidence (WOE) data and stores the model predictor names and corresponding coefficients in the `creditscorecard` object.

`fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic or manual binning process. The response variable is mapped so that "Good" is 1, and "Bad" is 0. This implies that higher (unscaled) scores correspond to better (less risky) individuals (smaller probability of default).

Alternatively, you can use `setmodel` to provide names of the predictors that you want in the logistic regression model, along with their corresponding coefficients.

`[sc,mdl] = fitmodel(sc)` fits a logistic regression model to the Weight of Evidence (WOE) data and stores the model predictor names and corresponding coefficients in the `creditscorecard` object. `fitmodel` returns an updated `creditscorecard` object and a `GeneralizedLinearModel` object containing the fitted model.

`fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic or manual binning process. The response variable is mapped so that "Good" is 1, and "Bad" is 0. This implies that higher (unscaled) scores correspond to better (less risky) individuals (smaller probability of default).

Alternatively, you can use `setmodel` to provide names of the predictors that you want in the logistic regression model, along with their corresponding coefficients.

`[sc,mdl] = fitmodel(____,Name,Value)` fits a logistic regression model to the Weight of Evidence (WOE) data using optional name-value pair arguments and stores the model predictor names and corresponding coefficients in the `creditscorecard` object. Using name-value pair arguments, you can select which Generalized Linear Model to fit the data. `fitmodel` returns an updated `creditscorecard` object and a `GeneralizedLinearModel` object containing the fitted model.

`fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic or manual binning process. The response variable is mapped so that "Good" is 1, and "Bad" is 0. This implies that higher (unscaled) scores correspond to better (less risky) individuals (smaller probability of default).

Alternatively, you can use `setmodel` to provide names of the predictors that you want in the logistic regression model, along with their corresponding coefficients.

Examples

Fit a Stepwise Logistic Model

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: 'CustID'
  GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc =
```

```
creditscorecard with properties:
```

```
      IDVar: 'CustID'
      GoodLabel: 0
      ResponseVar: 'status'
      VarNames: {1x11 cell}
      PredictorVars: {1x9 cell}
```

Use `fitmodel` to fit a logistic regression model using Weight of Evidence (WOE) data. `fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic binning process. `fitmodel` then fits a logistic regression model using a stepwise method (by default).

```
sc = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-05
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

Generalized Linear regression model:

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom
Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Fit a Logistic Model with All Predictors

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
    creditscorecard with properties:
        IDVar: 'CustID'
        GoodLabel: 0
        ResponseVar: 'status'
        VarNames: {1x11 cell}
        PredictorVars: {1x9 cell}
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc =
    creditscorecard with properties:
        IDVar: 'CustID'
        GoodLabel: 0
        ResponseVar: 'status'
        VarNames: {1x11 cell}
        PredictorVars: {1x9 cell}
```

Use `fitmodel` to fit a logistic regression model using Weight of Evidence (WOE) data. `fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic binning process. Set the `VariableSelection` name-value pair argument to `FullModel` to specify that all predictors must be included in the fitted logistic regression model.

```
sc = fitmodel(sc, 'VariableSelection', 'FullModel');
```

```
Generalized Linear regression model:  
status ~ [Linear formula with 10 terms in 9 predictors]  
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70262	0.063862	11.002	3.734e-28
CustAge	0.57683	0.27064	2.1313	0.033062
TmAtAddress	1.0653	0.55233	1.9287	0.053762
ResStatus	1.4189	0.65162	2.1775	0.029441
EmpStatus	0.89916	0.29217	3.0776	0.002087
CustIncome	0.77506	0.21942	3.5323	0.0004119
TmWBank	1.0826	0.26583	4.0727	4.648e-05
OtherCC	1.1354	0.52827	2.1493	0.031612
AMBalance	0.99315	0.32642	3.0425	0.0023459
UtilRate	0.16723	0.55745	0.29999	0.76419

1200 observations, 1190 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 85.6, p-value = 1.25e-14

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `[sc,mdl] = fitmodel(sc,'VariableSelection','FullModel')`

'PredictorVars' — Predictor variables for fitting creditscorecard object

all predictors in the `creditscorecard` object (default) | cell array of strings

Predictor variables for fitting the `creditscorecard` object, specified when using a cell array of strings. When provided, the `creditscorecard` object property `PredictorVars` is updated. When not provided, the predictors used to create the `creditscorecard` object (by using the `creditscorecard` function) are used.

'VariableSelection' — Variable selection method to fit logistic regression model

'Stepwise' (default) | string with values 'Stepwise', 'FullModel'

The variable selection method to fit the logistic regression model, specified as a string with values 'Stepwise' or 'FullModel':

- **Stepwise** — Uses a stepwise selection method which calls the Statistics Toolbox function `stepwiseglm`. Only variables in `PredictorVars` can potentially become part of the model and uses the `StartingModel` name-value pair argument to select the starting model.
- **FullModel** — Fits a model with all predictor variables in the `PredictorVars` name-value pair argument and calls `fitglm`.

Note: Only variables in the `PredictorVars` property of the `creditscorecard` object can potentially become part of the logistic regression model and only linear terms are included in this model with no interactions or any other higher-order terms.

The response variable is mapped so that “Good” is 1 and “Bad” is 0.

Data Types: char

'StartingModel' — Initial model for Stepwise variable selection

'Constant' (default) | string with values 'Constant', 'Linear'

Initial model for the `Stepwise` variable selection method, specified using a string with values 'Constant' or 'Linear'. This option determines the initial model (constant or linear) that the Statistics Toolbox function `stepwiseglm` starts with.

- **Constant** — Starts the stepwise method with an empty (constant only) model.

- **Linear** — Starts the stepwise method from a full (all predictors in) model.

Note: `StartingModel` is used only for the `Stepwise` option of `VariableSelection` and has no effect for the `FullModel` option of `VariableSelection`.

Data Types: char

'Display' — Indicator to display model information at command line

'On' (default) | string with values 'On', 'Off'

Indicator to display model information at command line, specified using a string with value 'On' or 'Off'.

Data Types: char

Output Arguments

sc — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object. The `creditscorecard` object contains information about the model predictors and coefficients used to fit the WOE data. For more information on using the `creditscorecard` object, see [Using creditscorecard Objects](#).

mdl — Fitted logistic model

`GeneralizedLinearModel` object

Fitted logistic model, returned as an object of type `GeneralizedLinearModel` containing the fitted model. For more information on a `GeneralizedLinearModel` object, see [GeneralizedLinearModel](#).

More About

Algorithms

For the logistic regression model used in the `creditscorecard` object, the probability of being “Bad” is given by

$\text{ProbBad} = \exp(-s) / (1 + \exp(-s))$.

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using creditscorecard Objects
- “What Are Generalized Linear Models?”

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitglm` | `formatpoints` | `GeneralizedLinearModel` | `modifybins` | `plotbins` | `score` | `setmodel` | `stepwiseglm`

setmodel

Set model predictors and coefficients

Syntax

```
sc = setmodel(sc,ModelPredictors,ModelCoefficients)
```

Description

`sc = setmodel(sc,ModelPredictors,ModelCoefficients)` sets the predictors and coefficients of a linear logistic regression model fitted outside the `creditscorecard` object and returns an updated `creditscorecard` object. The predictors and coefficients are used for the computation of scorecard points. Use `setmodel` in lieu of `fitmodel`, which fits a linear logistic regression model, because `setmodel` offers increased flexibility. For example, when a model fitted with `fitmodel` needs to be modified, you can use `setmodel`. For more information, see “Workflows for Using `setmodel`” on page 18-1621.

Note: When using `setmodel`, the following assumptions apply:

- The model coefficients correspond to a linear logistic regression model (where only linear terms are included in the model and there are no interactions or any other higher-order terms).
 - The model was previously fitted using Weight of Evidence (WOE) data with the response mapped so that ‘Good’ is 1 and ‘Bad’ is 0.
-

Examples

Modify a GLM Model Fitted with `fitmodel`

This example shows how to use `setmodel` to make modifications to a logistic regression model initially fitted using the `fitmodel` function, and then set the new logistic regression model predictors and coefficients back into the `creditscorecard` object.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: 'CustID'
  GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}
```

Perform automatic binning.

```
sc = autobinning(sc);
```

The standard workflow is to use the `fitmodel` function to fit a logistic regression model using a stepwise method. However, `fitmodel` only supports limited options regarding the stepwise procedure. You can use the optional `mdl` output argument from `fitmodel` to get a copy of the fitted `GeneralizedLinearModel` object, to later modify.

```
[sc,mdl] = fitmodel(sc);
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901

```
Generalized Linear regression model:
```

```
status ~ [Linear formula with 8 terms in 7 predictors]
Distribution = Binomial
```

```
Estimated Coefficients:
```

Estimate	SE	tStat	pValue
-----	-----	-----	-----

(Intercept)	0.70329	0.06376	11.03	2.7292e-28
CustAge	0.57752	0.2701	2.1382	0.032501
ResStatus	1.3829	0.65008	2.1273	0.033395
EmpStatus	0.89259	0.2915	3.062	0.0021984
CustIncome	0.77069	0.21883	3.5218	0.00042856
TmWBank	1.104	0.26503	4.1655	3.1073e-05
OtherCC	1.1395	0.52667	2.1636	0.030496
AMBalance	1.0213	0.30537	3.3444	0.00082447

1200 observations, 1192 error degrees of freedom

Dispersion: 1

Chi²-statistic vs. constant model: 81.7, p-value = 6.09e-15

Suppose you want to include, or "force," the predictor 'UtilRate' in the logistic regression model, even though the stepwise method did not include it in the fitted model. You can add 'UtilRate' to the logistic regression model using the `GeneralizedLinearModel` object `mdl` directly.

```
mdl = mdl.addTerms('UtilRate')
```

```
mdl =
```

Generalized Linear regression model:

```
status ~ [Linear formula with 9 terms in 8 predictors]
```

```
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70321	0.063761	11.029	2.7753e-28
CustAge	0.57752	0.27008	2.1383	0.032494
ResStatus	1.3887	0.65047	2.135	0.032765
EmpStatus	0.8903	0.29162	3.053	0.002266
CustIncome	0.76861	0.21896	3.5103	0.00044764
TmWBank	1.0995	0.26539	4.1428	3.4307e-05
OtherCC	1.1345	0.52696	2.1528	0.031331
AMBalance	0.9864	0.32635	3.0225	0.0025069
UtilRate	0.16817	0.5563	0.30231	0.76242

1200 observations, 1191 error degrees of freedom

```
Dispersion: 1  
Chi^2-statistic vs. constant model: 81.8, p-value = 2.09e-14
```

Use `setmodel` to update the model predictors and model coefficients in the `creditscorecard` object. The `ModelPredictors` input argument does not explicitly include a string for the intercept. However, the `ModelCoefficients` input argument does have the intercept information as its first element.

```
ModelPredictors = mdl.PredictorNames  
ModelCoefficients = mdl.Coefficients.Estimate  
  
sc = setmodel(sc,ModelPredictors,ModelCoefficients);
```

```
ModelPredictors =  
  
    'CustAge'  
    'ResStatus'  
    'EmpStatus'  
    'CustIncome'  
    'TmWBank'  
    'OtherCC'  
    'AMBalance'  
    'UtilRate'
```

```
ModelCoefficients =  
  
    0.7032  
    0.5775  
    1.3887  
    0.8903  
    0.7686  
    1.0995  
    1.1345  
    0.9864  
    0.1682
```

Verify that `'UtilRate'` is part of the scorecard predictors by displaying the scorecard points.

```
pi = displaypoints(sc)
```

pi =

Predictors	Bin	Points
'CustAge'	'[-Inf, 37)'	-0.14989
'CustAge'	'[37, 43)'	0.017552
'CustAge'	'[43, 48)'	0.059318
'CustAge'	'[48, 54)'	0.26051
'CustAge'	'[54, Inf]'	0.27253
'ResStatus'	'Home Owner'	0.11474
'ResStatus'	'Tenant'	-0.044812
'ResStatus'	'Other'	0.36632
'EmpStatus'	'Employed'	0.30317
'EmpStatus'	'Unknown'	-0.089684
'CustIncome'	'[-Inf, 33000)'	-0.32182
'CustIncome'	'[33000, 38000)'	0.13681
'CustIncome'	'[38000, 42000)'	-0.031889
'CustIncome'	'[42000, 47000)'	0.24608
'CustIncome'	'[47000, Inf]'	0.45662
'TmWBank'	'[-Inf, 12)'	-0.19297
'TmWBank'	'[12, 23)'	-0.056207
'TmWBank'	'[23, 36)'	-0.0038303
'TmWBank'	'[36, 55)'	0.12981
'TmWBank'	'[55, Inf]'	0.62132
'OtherCC'	'No'	-0.2165
'OtherCC'	'Yes'	0.14844
'AMBalance'	'[-Inf, 259.78)'	0.20934
'AMBalance'	'[259.78, 558.88)'	0.45494
'AMBalance'	'[558.88, 875.97)'	-0.032256
'AMBalance'	'[875.97, 1254.28)'	-0.014334
'AMBalance'	'[1254.28, Inf]'	-0.13798
'UtilRate'	'[-Inf, 0.04)'	0.1034
'UtilRate'	'[0.04, 0.09)'	0.095483
'UtilRate'	'[0.09, 0.19)'	0.089917
'UtilRate'	'[0.19, 0.36)'	0.10349
'UtilRate'	'[0.36, Inf]'	0.04939

Fit a Logistic Regression Model Outside of the `creditscorecard` Object

This example shows how to use `setmodel` to fit a logistic regression model directly, without using the `fitmodel` function, and then set the new model predictors and coefficients back into the `creditscorecard` object. This approach gives more flexibility

regarding options to control the stepwise procedure. This example fits a logistic regression model with a nondefault value for the 'PEnter' parameter, the criterion to admit a new predictor in the logistic regression model during the stepwise procedure.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the 'IDVar' argument to indicate that 'CustID' contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: 'CustID'
  GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}
```

Perform automatic binning.

```
sc = autobinning(sc);
```

The logistic regression model needs to be fit with Weight of Evidence (WOE) data. The WOE transformation is a special case of binning, since the data first needs to be binned, and then the binned information is mapped to the corresponding WOE values. This transformation is done using the `bindata` function. `bindata` has an argument that prepares the data for the model fitting step. By setting the `bindata` name-value pair argument for 'OutputType' to `WOEModelInput`:

- All predictors are converted to WOE values.
- The output contains only predictors and response (no 'IDVar' or any unused variables).
- Predictors with infinite or undefined (NaN) WOE values are discarded.
- The response values are mapped so that "Good" is 1 and "Bad" is 0 (this implies that higher unscaled scores correspond to better, less risky customers).

```
bd = bindata(sc, 'OutputType', 'WOEModelInput');
```

Use a sample of four variables (including the response) in the original data.

```
data(1:10,{ 'CustAge' 'ResStatus' 'CustIncome' 'status'})
```

```
ans =
```

<u>CustAge</u>	<u>ResStatus</u>	<u>CustIncome</u>	<u>status</u>
53	Tenant	50000	0
61	Home Owner	52000	0
47	Tenant	37000	0
50	Home Owner	53000	0
68	Home Owner	53000	0
65	Home Owner	48000	0
34	Home Owner	32000	1
50	Other	51000	0
50	Tenant	52000	1
49	Home Owner	53000	1

Use the sample of the same rows and four variables as in the previous step, but use the binned data (`bd`) where the `bindata` name-value pair argument for `'OutputType'` is set to `'WOEModelInput'`.

```
bd(1:10,{ 'CustAge' 'ResStatus' 'CustIncome' 'status'})
```

```
ans =
```

<u>CustAge</u>	<u>ResStatus</u>	<u>CustIncome</u>	<u>status</u>
0.29888	-0.095564	0.47972	1
0.31969	0.019329	0.47972	1
-0.049492	-0.095564	0.06363	1
0.29888	0.019329	0.47972	1
0.31969	0.019329	0.47972	1
0.31969	0.019329	0.47972	1
-0.41175	0.019329	-0.53306	0
0.29888	0.20049	0.47972	1
0.29888	-0.095564	0.47972	0
0.29888	0.019329	0.47972	0

Fit a logistic linear regression model using a stepwise method with the Statistics Toolbox™ function `stepwiseglm`, but use a nondefault value for the `'PEnter'` and `'PRemove'` optional arguments. The predictor `'TmAtAddress'` would normally be excluded from the logistic linear regression model using default options for the stepwise procedure.

```
mdl = stepwiseglm(bd, 'constant', 'Distribution', 'binomial', ...
    'Upper', 'linear', 'PEnter', 0.1, 'PRemove', 0.15)
```

1. Adding CustIncome, Deviance = 1493.2079, Chi2Stat = 30.233456, PValue = 3.8304463e-0
2. Adding TmWBank, Deviance = 1476.036, Chi2Stat = 17.17184, PValue = 3.414608e-05
3. Adding AMBalance, Deviance = 1463.5299, Chi2Stat = 12.506148, PValue = 0.0004056150
4. Adding EmpStatus, Deviance = 1454.6882, Chi2Stat = 8.841685, PValue = 0.0029442719
5. Adding CustAge, Deviance = 1450.492, Chi2Stat = 4.1961542, PValue = 0.040515764
6. Adding OtherCC, Deviance = 1446.3066, Chi2Stat = 4.1854492, PValue = 0.040772406
7. Adding ResStatus, Deviance = 1441.7029, Chi2Stat = 4.6036533, PValue = 0.031903901
8. Adding TmAtAddress, Deviance = 1437.9586, Chi2Stat = 3.7442978, PValue = 0.05298798

```
mdl =
```

Generalized Linear regression model:

```
status ~ [Linear formula with 9 terms in 8 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.70268	0.063861	11.003	3.6836e-28
CustAge	0.57692	0.27066	2.1315	0.033046
TmAtAddress	1.0653	0.55221	1.9291	0.053717
ResStatus	1.4128	0.65117	2.1696	0.030039
EmpStatus	0.90148	0.29205	3.0868	0.0020234
CustIncome	0.77709	0.21929	3.5437	0.00039455
TmWBank	1.0872	0.26546	4.0956	4.2101e-05
OtherCC	1.1402	0.52803	2.1593	0.030826
AMBalance	1.0278	0.30541	3.3654	0.00076432

1200 observations, 1191 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 85.5, p-value = 3.83e-15

Use `setmodel` to update the model predictors and model coefficients in the `creditscorecard` object. The `ModelPredictors` input argument does not explicitly include a string for the intercept. However, the `ModelCoefficients` input argument does have the intercept information as its first element.

```
ModelPredictors = mdl.PredictorNames
ModelCoefficients = mdl.Coefficients.Estimate

sc = setmodel(sc,ModelPredictors,ModelCoefficients);
```

```
ModelPredictors =
```

```
    'CustAge'
    'TmAtAddress'
    'ResStatus'
    'EmpStatus'
    'CustIncome'
    'TmWBank'
    'OtherCC'
    'AMBalance'
```

```
ModelCoefficients =
```

```
    0.7027
    0.5769
    1.0653
    1.4128
    0.9015
    0.7771
    1.0872
    1.1402
    1.0278
```

Verify that the desired model predictors are part of the scorecard predictors by displaying the scorecard points.

```
pi = displaypoints(sc)
```

```
pi =
```

Predictors	Bin	Points
'CustAge'	'[- Inf, 37)'	-0.14971
'CustAge'	'[37, 43)'	0.017559
'CustAge'	'[43, 48)'	0.059282
'CustAge'	'[48, 54)'	0.26027
'CustAge'	'[54, Inf]'	0.27227
'TmAtAddress'	'[- Inf, 16)'	0.037086
'TmAtAddress'	'[16, 30)'	0.27005
'TmAtAddress'	'[30, 44)'	0.029932
'TmAtAddress'	'[44, 65)'	0.1947
'TmAtAddress'	'[65, Inf]'	-0.07336
'ResStatus'	'Home Owner'	0.11514
'ResStatus'	'Tenant'	-0.047176
'ResStatus'	'Other'	0.37108
'EmpStatus'	'Employed'	0.30581
'EmpStatus'	'Unknown'	-0.091982
'CustIncome'	'[- Inf, 33000)'	-0.3264
'CustIncome'	'[33000, 38000)'	0.13728
'CustIncome'	'[38000, 42000)'	-0.033277
'CustIncome'	'[42000, 47000)'	0.24775
'CustIncome'	'[47000, Inf]'	0.46062
'TmWBank'	'[- Inf, 12)'	-0.18991
'TmWBank'	'[12, 23)'	-0.054669
'TmWBank'	'[23, 36)'	-0.0028758
'TmWBank'	'[36, 55)'	0.12928
'TmWBank'	'[55, Inf]'	0.61532
'OtherCC'	'No'	-0.21809
'OtherCC'	'Yes'	0.14868
'AMBalance'	'[- Inf, 259.78)'	0.21438
'AMBalance'	'[259.78, 558.88)'	0.47028
'AMBalance'	'[558.88, 875.97)'	-0.037368
'AMBalance'	'[875.97, 1254.28)'	-0.018694
'AMBalance'	'[1254.28, Inf]'	-0.14753

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

ModelPredictors — Predictor names included in fitted model

cell array of strings with predictor values

```
{'PredictorName1', 'PredictorName2', ...}
```

Predictor names included in the fitted model, specified as a cell array of strings as `{'PredictorName1', 'PredictorName2', ...}`. The predictor names must match predictor variable names in the `creditscorecard` object.

Note: Do not include a string for the constant term in `ModelPredictors`, `setmodel` internally handles the `'(Intercept)'` term based on the number of model coefficients (see `ModelCoefficients`).

Data Types: `cell`

ModelCoefficients — Model coefficients corresponding to model predictors

numeric array with values `[coeff1, coeff2, ...]`

Model coefficients corresponding to the model predictors, specified as a numeric array of model coefficients, `[coeff1, coeff2, ...]`. If N is the number of predictor names provided in `ModelPredictors`, the size of `ModelCoefficients` can be N or $N+1$. If `ModelCoefficients` has $N+1$ elements, then the first coefficient is used as the `'(Intercept)'` of the fitted model. Otherwise, the `'(Intercept)'` is set to 0.

Data Types: `double`

Output Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, returned as an updated `creditscorecard` object. The `creditscorecard` object contains information about the model predictors and coefficients of the fitted model. For more information on using the `creditscorecard` object, see [Using creditscorecard Objects](#).

More About

Workflows for Using `setmodel`

When using `setmodel`, there are two possible workflows to set the final model predictors and model coefficients into a `creditscorecard` object.

The first workflow is:

- Use `fitmodel` to get the optional output argument `mdl`. This is a `GeneralizedLinearModel` object and you can add and remove terms, or modify the parameters of the stepwise procedure. Only linear terms can be in the model (no interactions or any other higher-order terms).
- Once the `GeneralizedLinearModel` object is satisfactory, set the final model predictors and model coefficients into the `creditscorecard` object using the `setmodel` input arguments for `ModelPredictors` and `ModelCoefficients`.

An alternate workflow is:

- Obtain the Weight of Evidence (WOE) data using `bindata`. Use the `'WOEModelInput'` option for the `'OutputType'` name-value pair argument in `bindata` to ensure that:
 - The predictors data is transformed to WOE.
 - Only predictors whose bins have finite WOE values are included.
 - The response variable is placed in the last column.
 - The response variable is mapped (“Good” is 1 and “Bad” is 0).
- Use the data from the previous step to fit a linear logistic regression model (only linear terms in the model, no interactions or any other higher-order terms). See, for example, `stepwiseglm`.
- Once the `GeneralizedLinearModel` object is satisfactory, set the final model predictors and model coefficients into the `creditscorecard` object using the `setmodel` input arguments for `ModelPredictors` and `ModelCoefficients`.

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitglm` | `fitmodel` | `formatpoints` | `GeneralizedLinearModel` | `modifybins` | `plotbins` | `score` | `stepwiseglm`

bindata

Binned predictor variables

Syntax

```
bdata = bindata(sc)
bdata = bindata(sc,data)
bdata = bindata(sc,Name,Value)
```

Description

`bdata = bindata(sc)` binned predictor variables returned as a table. This is a table of the same size (see exception in the following Note) as the data input, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

`bdata = bindata(sc,data)` returns a table of binned predictor variables. `bindata` returns a table of the same size as the `creditscorecard` data, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

`bdata = bindata(sc,Name,Value)` binned predictor variables returned as a table using optional name-value pair arguments. This is a table of the same size (see exception in the following Note) as the data input, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

Note: When the `bindata` name-value pair argument `'OutputType'` is set to `'WOEModelInput'`, the `bdata` output only contains the columns corresponding to predictors whose bins do not have `Inf` or `NaN` Weight of Evidence (WOE) values, and `bdata` includes the mapped response as the last column.

Missing data (if any) are included in the `bdata` output as missing data as well, and do not influence the rules to discard predictors when `'OutputType'` is set to `'WOEModelInput'`.

Examples

Bin `creditscorecard` Data as Bin Numbers, Categories, or WOE Values

This example shows how to use the `bindata` function to simply bin or discretize data.

Suppose bin ranges of

- '0 to 30'
- '31 to 50'
- '51 and up'

are determined for the age variable (via manual or automatic binning). If a data point with age 41 is given, binning this data point means placing it in the bin for 41 years old, which is the second bin, or the '31 to 50' bin. Binning is then the mapping from the original data, into discrete groups or bins. In this example, you can say that a 41-year old is mapped into bin number 2, or that it is binned into the '31 to 50' category. If you know the Weight of Evidence (WOE) value for each of the three bins, you could also replace the data point 41 with the WOE value corresponding to the second bin. `bindata` supports the three binning formats just mentioned:

- Bin number (where the '`OutputType`' name-value pair argument is set to '`BinNumber`'); this is the default option, and in this case, 41 is mapped to bin 2.
- Categorical (where the '`OutputType`' name-value pair argument is set to '`Categorical`'); in this case, 41 is mapped to the '31 to 50' bin.
- WOE value (where the '`OutputType`' name-value pair argument is set to '`WOE`'); in this case, 41 is mapped to the WOE value of bin number 2.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the '`IDVar`' argument to indicate that '`CustID`' contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

```

      IDVar: 'CustID'
    GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}

```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for 'CustAge'.

```
bininfo(sc, 'CustAge')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,37)'	134	100	1.34	-0.41175	0.035005
'[37,43)'	154	86	1.7907	-0.12181	0.0030263
'[43,48)'	154	80	1.925	-0.049492	0.00048159
'[48,54)'	180	66	2.7273	0.29888	0.017309
'[54,Inf]'	181	65	2.7846	0.31969	0.019718
'Totals'	803	397	2.0227	NaN	0.075539

These are the first 10 age values in the original data, used to create the `creditscorecard` object.

```
data(1:10, 'CustAge')
```

```
ans =
```

```

CustAge
-----
53
61
47
50

```

```
68
65
34
50
50
49
```

Bin scorecard data into bin numbers (default behavior).

```
bdata = bindata(sc);
```

According to the bin information, the first age should be mapped into the fourth bin, the second age into the fifth bin, etc. These are the first 10 binned ages, in bin-number format.

```
bdata(1:10, 'CustAge')
```

```
ans =
```

```
    CustAge
    _____
    4
    5
    3
    4
    5
    5
    1
    4
    4
    4
```

Bin the scorecard data and show their bin labels. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'Categorical'`.

```
bdata = bindata(sc, 'OutputType', 'Categorical');
```

These are the first 10 binned ages, in categorical format.

```
bdata(1:10, 'CustAge')
```

```
ans =
    CustAge
    -----
    [48,54)
    [54,Inf]
    [43,48)
    [48,54)
    [54,Inf]
    [54,Inf]
    [-Inf,37)
    [48,54)
    [48,54)
    [48,54)
```

Convert the scorecard data to WOE values. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'WOE'`.

```
bdata = bindata(sc, 'OutputType', 'WOE');
```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values that are internally displayed using the `bininfo` function.

```
bdata(1:10, 'CustAge')
```

```
ans =
    CustAge
    -----
    0.29888
    0.31969
   -0.049492
    0.29888
    0.31969
    0.31969
   -0.41175
    0.29888
    0.29888
    0.29888
```

Bin Additional "Test" Data

This example shows how to use the `bindata` function's optional input for the data to bin. If not provided, `bindata` bins the `creditscorecard` training data. However, if a different dataset needs to be binned, for example, some "test" data, this can be passed into `bindata` as an optional input.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: 'CustID'
  GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc, 'CustAge')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,37)'	134	100	1.34	-0.41175	0.035005
'[37,43)'	154	86	1.7907	-0.12181	0.0030263

'[43,48)'	154	80	1.925	-0.049492	0.00048159
'[48,54)'	180	66	2.7273	0.29888	0.017309
'[54,Inf]'	181	65	2.7846	0.31969	0.019718
'Totals'	803	397	2.0227	NaN	0.075539

For the purpose of illustration, take a few rows from the original data as "test" data and display the first 10 age values in the test data.

```
tdata = data(101:110, :);
tdata(1:10, 'CustAge')
```

```
ans =
```

```
    CustAge
```

```
-----
34
59
64
61
28
65
55
37
49
51
```

Convert the test data to WOE values. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'WOE'`, passing the test data (`tdata`) as an optional input.

```
bdata = bindata(sc,tdata, 'OutputType', 'WOE')
```

```
bdata =
```

CustID	CustAge	TmAtAddress	ResStatus	EmpStatus	CustIncome	TmWB
-----	-----	-----	-----	-----	-----	-----
101	-0.41175	0.17105	-0.095564	0.2418	-0.15585	0.4
102	0.31969	-0.15132	0.019329	-0.19947	0.20579	-0.1
103	0.31969	0.10032	0.019329	0.2418	0.47972	-0.08

104	0.31969	-0.15132	-0.095564	0.2418	0.47972	-0.083
105	-0.41175	0.10032	0.019329	0.2418	0.06363	0.48
106	0.31969	-0.054355	0.019329	-0.19947	0.20579	-0.13
107	0.31969	-0.047639	-0.095564	0.2418	0.47972	0.48
108	-0.12181	0.17105	0.019329	-0.19947	-0.53306	-0.13
109	0.29888	0.17105	-0.095564	0.2418	-0.15585	-0.13
110	0.29888	0.17105	0.019329	0.2418	0.20579	-0.13

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values displayed internally by `bininfo`.

```
bdata(1:10, 'CustAge')
```

```
ans =
```

```
CustAge
```

```
-----
-0.41175
0.31969
0.31969
0.31969
-0.41175
0.31969
0.31969
-0.12181
0.29888
0.29888
```

Apply a Weight of Evidence (WOE) Transformation to Data

`bindata` supports the following types of WOE transformation:

- When the `'OutputType'` name-value argument is set to `'WOE'`, `bindata` simply applies the WOE transformation to all predictors and keeps the rest of the variables in the original data in place and unchanged.
- When the `'OutputType'` name-value pair argument is set to `'WOEModelInput'`, `bindata` returns a table that can be used directly as an input for fitting a logistic regression model for the scorecard. In this case, `bindata`:
- Applies WOE transformation to all predictors.

- Returns predictor variables, but no IDVar or unused variables are included in the output.
- Includes the mapped response variable as the last column.
- The `fitmodel` function calls `bindata` internally using the 'WOEModelInput' option to fit the logistic regression model for the `creditscorecard` model.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the 'IDVar' argument to indicate that 'CustID' contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID')
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: 'CustID'
  GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
  PredictorVars: {1x9 cell}
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for 'CustAge'.

```
bininfo(sc, 'CustAge')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
' [-Inf,37) '	134	100	1.34	-0.41175	0.035005
' [37,43) '	154	86	1.7907	-0.12181	0.0030263
' [43,48) '	154	80	1.925	-0.049492	0.00048159
' [48,54) '	180	66	2.7273	0.29888	0.017309
' [54,Inf] '	181	65	2.7846	0.31969	0.019718

```
'Totals'      803      397      2.0227      NaN      0.075539
```

These are the first 10 age values in the original data, used to create the `creditscorecard` object.

```
data(1:10, 'CustAge')
```

```
ans =
```

```
  CustAge
```

```
-----  
53  
61  
47  
50  
68  
65  
34  
50  
50  
49
```

Convert the test data to WOE values. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'WOE'`.

```
bdata = bindata(sc, 'OutputType', 'WOE');
```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values displayed internally by `bininfo`.

```
bdata(1:10, 'CustAge')
```

```
ans =
```

```
  CustAge
```

```
-----  
0.29888  
0.31969  
-0.049492
```

```

0.29888
0.31969
0.31969
-0.41175
0.29888
0.29888
0.29888

```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values displayed internally by `bininfo`.

```
bdata(1:10, 'CustAge')
```

```
ans =
```

```

CustAge
-----
0.29888
0.31969
-0.049492
0.29888
0.31969
0.31969
-0.41175
0.29888
0.29888
0.29888

```

The size of the original data and the size of `bdata` output are the same because `bindata` leaves unused variables (such as `'IDVar'`) unchanged and in place.

```
whos data bdata
```

Name	Size	Bytes	Class	Attributes
bdata	1200x11	109650	table	
data	1200x11	89234	table	

The response values are the same in the original data and in the binned data because, by default, `bindata` does not modify response values.

```
disp([data.status(1:10) bdata.status(1:10)])  
  
0    0  
0    0  
0    0  
0    0  
0    0  
0    0  
1    1  
0    0  
1    1  
1    1
```

When fitting a logistic regression model with WOE data, set the 'OutputType' name-value pair argument to 'WOEModelInput'.

```
bdata = bindata(sc, 'OutputType', 'WOEModelInput');
```

The binned predictor data is the same as when the 'OutputType' name-value pair argument is set to 'WOE'.

```
bdata(1:10, 'CustAge')
```

```
ans =  
  
CustAge  
-----  
0.29888  
0.31969  
-0.049492  
0.29888  
0.31969  
0.31969  
-0.41175  
0.29888  
0.29888  
0.29888
```

However, the size of the original data and the size of `bdata` output are different. This is because `bindata` removes unused variables (such as 'IDVar').

```
whos data bdata
```

Name	Size	Bytes	Class	Attributes
bdata	1200x10	99814	table	
data	1200x11	89234	table	

The response values are also modified in this case and are mapped so that "Good" is 1 and "Bad" is 0.

```
disp([data.status(1:10) bdata.status(1:10)])
```

```

0     1
0     1
0     1
0     1
0     1
0     1
1     0
0     1
1     0
1     0

```

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

data — Data to bin given the rules set in `creditscorecard` object

table

Data to bin given the rules set in the `creditscorecard` object, specified using a table. By default, data is set to the `creditscorecard` object's raw data.

Before creating a `creditscorecard` object, perform a data preparation task to have an appropriately structured data as input to a `creditscorecard` object.

Data Types: table

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

```
Example: bdata =  
bindata(sc, 'OutputType', 'WOE', 'ResponseFormat', 'Mapped')
```

'OutputType' — Output format

'BinNumber' (default) | string with values 'BinNumber', 'Categorical', 'WOE'

Output format, specified as a string with the following values:

- `BinNumber` — Returns the bin numbers corresponding to each observation.
- `Categorical` — Returns the bin label corresponding to each observation.
- `WOE` — Returns the Weight of Evidence (WOE) corresponding to each observation.
- `WOEModelInput` — Use this option when fitting a model. This option:
 - Returns the Weight of Evidence (WOE) corresponding to each observation.
 - Returns predictor variables, but no `IDVar` or unused variables are included in the output.
 - Discards any predictors whose bins have `Inf` or `NaN` WOE values.
 - Includes the mapped response variable as the last column.

Data Types: char

'ResponseFormat' — Response values format

'RawData' (default) | string with values 'RawData', 'Mapped'

Response values format, specified using a string with the following values:

- `RawData` — The response variable is copied unchanged into the `bdata` output.
- `Mapped` — The response values are modified (if necessary) so that "Good" is mapped to 1, and "Bad" is mapped to 0.

Data Types: char

Output Arguments

bdata — Binned predictor variables

table

Binned predictor variables, returned as a table. This is a table of the same size (see exception in the following Note) as the data input, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

Note: When the `bindata` name-value pair argument `'OutputType'` is set to `'WOEModelInput'`, the `bdata` output only contains the columns corresponding to predictors whose bins do not have `Inf` or `NaN` Weight of Evidence (WOE) values, and `bdata` includes the mapped response as the last column.

Missing data (if any) are included in the `bdata` output as missing data as well, and do not influence the rules to discard predictors when `'OutputType'` is set to `'WOEModelInput'`.

More About

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

plotbins

Plot histogram counts for predictor variables

Syntax

```
plotbins(sc,PredictorName)
hFigure = plotbins(sc,PredictorName)
hFigure = plotbins( ____,Name,Value)
```

Description

`plotbins(sc,PredictorName)` plots histogram counts for given predictor variables. When a predictor's bins are modified using `modifybins` or `autobinning`, rerun `plotbins` to update the figure to reflect the change.

`hFigure = plotbins(sc,PredictorName)` returns a handle to the figure. `plotbins` plots histogram counts for given predictor variables. When a predictor's bins are modified using `modifybins` or `autobinning`, rerun `plotbins` to update the figure to reflect the change.

`hFigure = plotbins(____,Name,Value)` returns a handle to the figure. `plotbins` plots histogram counts for given predictor variables using optional name-value pair arguments. When a predictor's bins are modified using `modifybins` or `autobinning`, rerun `plotbins` to update the figure to reflect the change.

Examples

Plot a Histogram for Bin Information

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the `PredictorName` input argument for `CustIncome` using the defaults for the algorithm `EqualFrequency`.


```
sc = autobinning(sc, 'CustIncome')
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: ''
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x10 cell}
```

Use `bininfo` to display the autobinned data.

```
[bi, cp] = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,33000)'	127	107	1.1869	-0.53306	0.059364
'[33000,38000)'	194	90	2.1556	0.06363	0.00094769
'[38000,42000)'	135	78	1.7308	-0.15585	0.004419
'[42000,47000)'	164	66	2.4848	0.20579	0.0078175
'[47000,Inf]'	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.11421

```
cp =
```

```
33000
38000
42000
47000
```

Manually remove the second cut point (the boundary between the second and third bins) to merge bins two and three. Use the `modifybins` function to update the scorecard and then display updated bin information.

```
cp(2) = [];
```

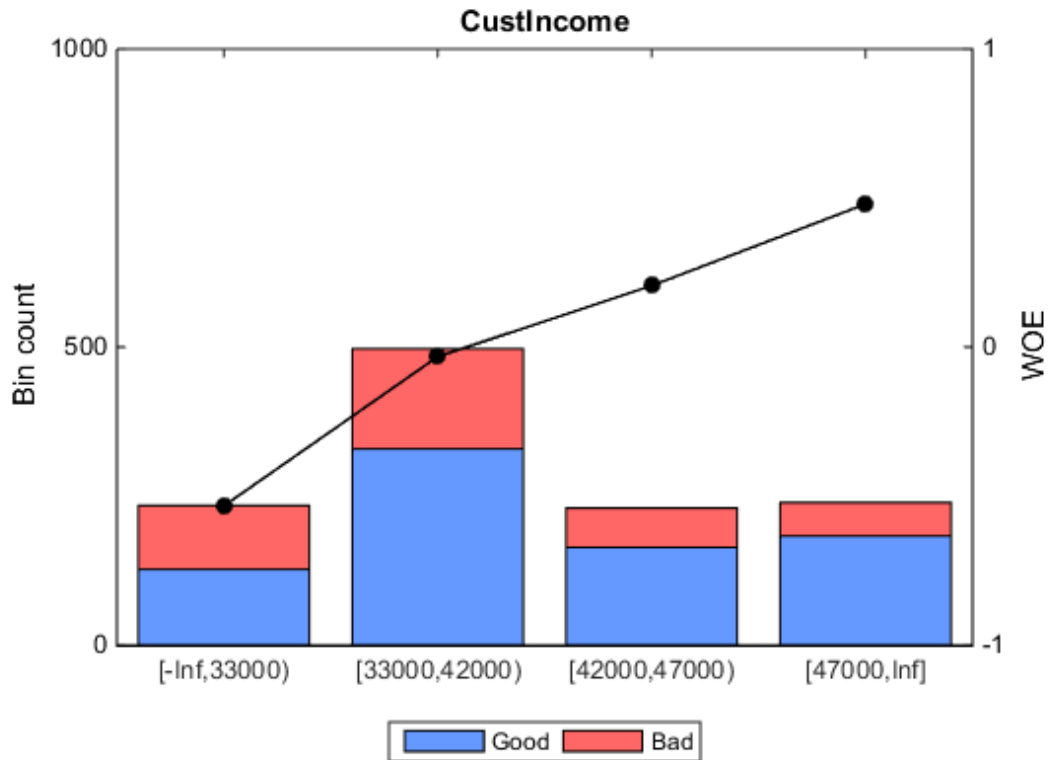
```
sc = modifybins(sc, 'CustIncome', 'CutPoints', cp);  
bi = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,33000)'	127	107	1.1869	-0.53306	0.059364
'[33000,42000)'	329	168	1.9583	-0.032325	0.0004351
'[42000,47000)'	164	66	2.4848	0.20579	0.0078175
'[47000,Inf]'	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.10927

Plot the histogram count for updated bin information for the PredictorName called CustIncome.

```
plotbins(sc, 'CustIncome');
```



Plot a Histogram for Bin Information Using Name-Value Pair Arguments

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the `PredictorName` input argument for `CustIncome` using the defaults for the algorithm `EqualFrequency`.

```
sc = autobinning(sc, 'CustIncome')
```

```

sc =
    creditscorecard with properties:
        IDVar: ''
        GoodLabel: 0
        ResponseVar: 'status'
        VarNames: {1x11 cell}
        PredictorVars: {1x10 cell}

```

Use `bininfo` to display the autobinned data.

```
[bi, cp] = bininfo(sc, 'CustIncome')
```

```

bi =

```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,33000)'	127	107	1.1869	-0.53306	0.059364
'[33000,38000)'	194	90	2.1556	0.06363	0.00094769
'[38000,42000)'	135	78	1.7308	-0.15585	0.004419
'[42000,47000)'	164	66	2.4848	0.20579	0.0078175
'[47000,Inf]'	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.11421

```

cp =

```

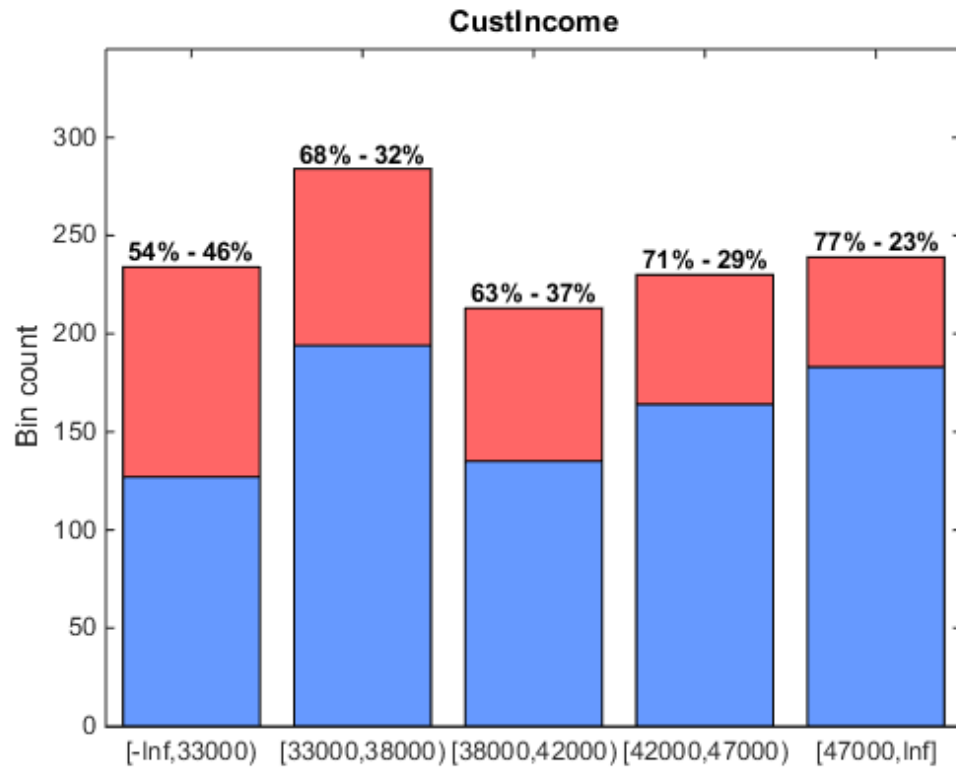
```

    33000
    38000
    42000
    47000

```

Plot the bin information for `CustIncome` without the Weight of Evidence (WOE) line and without a legend by setting the `'WOE'` and `'Legend'` name-value arguments to `'Off'`. Also, set the `'BinText'` name-value pair argument to `'PercentRows'` to show as text over the plot bars for the proportion of "Good" and "Bad" within each bin, that is, the probability of "Good" and "Bad" within each bin.

```
plotbins(sc, 'CustIncome', 'WOE', 'Off', 'Legend', 'Off', 'BinText', 'PercentRows');
```



- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

PredictorName — Name of one or more predictors to plot

string with predictor name | cell array of strings with predictor names

Name of one or more predictors to plot, specified using a string or cell array of strings containing the one or more names of the predictors.

Data Types: char | cell

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, . . . ,NameN,ValueN.

Example: `plotbins(sc,PredictorName,'BinText','Count','WOE','On')`

'BinText' — Information to display on top of plotted bin counts

'None' (default) | string with values 'None', 'Count', 'PercentRows', 'PercentCols', 'PercentTotal'

Information to display on top of plotted bin counts, specified using a string with values:

- **None** — No text is displayed on top of the bins.
- **Count** — For each bin, displays the count for “Good” and “Bad.”
- **PercentRows** — For each bin, displays the count for “Good” and “Bad” as a percentage of the number of observations in the bin.
- **PercentCols** — For each bin, displays the count for “Good” and “Bad” as a percentage of the total “Good” and total “Bad” in the entire sample.
- **PercentTotal** — For each bin, displays the count for “Good” and “Bad” as a percentage of the total number of observations in the entire sample.

Data Types: char

'WOE' — Indicator for Weight of Evidence (WOE)

'On' (default) | string with values 'On', 'Off'

Indicator for Weight of Evidence (WOE) line, specified using a string with values On or Off. When set to On, the WOE line is plotted on top of the histogram.

Data Types: char

'Legend' — Indicator for legend on plot

'On' (default) | string with values 'On', 'Off'

Indicator for legend on the plot, specified using a string with values `On` or `Off`.

Data Types: `char`

Output Arguments

hFigure — Figure handle of predictor variable

figure object

Figure handle of predictor variable, returned as a figure object or array of figure objects if more than one `PredictorName` is specified as an input.

More About

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `score` | `setmodel`

modifybins

Modify predictor's bins

Syntax

```
sc = modifybins(sc,PredictorName,Name,Value)
```

Description

`sc = modifybins(sc,PredictorName,Name,Value)` manually modifies predictor bins for numeric predictors or categorical predictors using optional name-value pair arguments. For numeric predictors, minimum value, maximum value, and cut points can be specified. For categorical predictors, category groupings can be specified. Bin labels can be specified for both types of predictors.

Examples

Modify Predictor Bins for Numeric Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The predictor `CustIncome` is numeric. By default, each value of a predictor is placed in a separate bin.

```
bi = bininfo(sc,'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'18000'	2	3	0.66667	-1.1099	0.0056227
'19000'	1	2	0.5	-1.3976	0.0053002
'20000'	4	2	2	-0.011271	6.3641e-07

'21000'	6	3	2	-0.011271	9.5462e-07
'22000'	4	2	2	-0.011271	6.3641e-07
'23000'	4	4	1	-0.70442	0.0035885
'24000'	5	5	1	-0.70442	0.0044856
'25000'	4	9	0.44444	-1.5153	0.026805
'26000'	4	11	0.36364	-1.716	0.038999
'27000'	6	6	1	-0.70442	0.0053827
'28000'	13	11	1.1818	-0.53736	0.0061896
'29000'	11	10	1.1	-0.60911	0.0069988
'30000'	18	16	1.125	-0.58664	0.010493
'31000'	24	8	3	0.39419	0.0038382
'32000'	21	15	1.4	-0.36795	0.0042797
'33000'	35	19	1.8421	-0.093509	0.00039951
'34000'	33	17	1.9412	-0.041124	7.095e-05
'35000'	39	20	1.95	-0.036589	6.6225e-05
'36000'	30	12	2.5	0.21187	0.0015113
'37000'	57	22	2.5909	0.24759	0.0038545
'38000'	28	22	1.2727	-0.46326	0.0095182
'39000'	39	22	1.7727	-0.1319	0.00090321
'40000'	42	17	2.4706	0.20004	0.0018969
'41000'	26	17	1.5294	-0.27954	0.0029191
'42000'	39	16	2.4375	0.18655	0.001542
'43000'	39	14	2.7857	0.32009	0.0042582
'44000'	27	12	2.25	0.10651	0.00036184
'45000'	35	12	2.9167	0.36602	0.00489
'46000'	24	12	2	-0.011271	3.8185e-06
'47000'	22	6	3.6667	0.59486	0.0073073
'48000'	35	9	3.8889	0.65371	0.013673
'49000'	15	8	1.875	-0.07581	0.00011153
'50000'	26	3	8.6667	1.4551	0.036118
'51000'	14	7	2	-0.011271	2.2274e-06
'52000'	17	5	3.4	0.51936	0.0044541
'53000'	15	6	2.5	0.21187	0.00075566
'54000'	10	3	3.3333	0.49955	0.0024461
'55000'	6	0	Inf	Inf	Inf
'56000'	9	2	4.5	0.79966	0.004934
'57000'	2	2	1	-0.70442	0.0017942
'58000'	1	2	0.5	-1.3976	0.0053002
'59000'	3	2	1.5	-0.29895	0.00038918
'60000'	4	0	Inf	Inf	Inf
'61000'	3	0	Inf	Inf	Inf
'62000'	1	1	1	-0.70442	0.00089712
'Totals'	803	397	2.0227	NaN	Inf

Use `modifybins` to set a minimum value of 0, and cut points every 10000, from 20000 to 60000. Display updated bin information, including cut points.

```
sc = modifybins(sc, 'CustIncome', 'MinValue', 0, 'CutPoints', 20000:10000:60000);
[bi,cp] = bininfo(sc, 'CustIncome')
```

bi =

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,20000)'	3	5	0.6	-1.2152	0.010765
'[20000,30000)'	61	63	0.96825	-0.73668	0.060942
'[30000,40000)'	324	173	1.8728	-0.076967	0.0024846
'[40000,50000)'	304	123	2.4715	0.20042	0.013781
'[50000,60000)'	103	32	3.2188	0.46457	0.022144
'[60000,Inf]'	8	1	8	1.375	0.010235
'Totals'	803	397	2.0227	NaN	0.12035

cp =

```
20000
30000
40000
50000
60000
```

The first and last bins contain very few points. To merge the first bin into the second one, remove the first cut point. Similarly, to merge the last bin into the second-to-last one, remove the last cut point. Then use `modifybins` to update the scorecard, and display updated bin information.

```
cp(1)=[];
cp(end)=[];
sc = modifybins(sc, 'CustIncome', 'CutPoints', cp);
bi = bininfo(sc, 'CustIncome')
```

bi =

Bin	Good	Bad	Odds	WOE	InfoValue
-----	------	-----	------	-----	-----------

'[0,30000]'	64	68	0.94118	-0.76504	0.070065
'[30000,40000]'	324	173	1.8728	-0.076967	0.0024846
'[40000,50000]'	304	123	2.4715	0.20042	0.013781
'[50000,Inf]'	111	33	3.3636	0.5086	0.028028
'Totals'	803	397	2.0227	NaN	0.11436

Modify Predictor Bins for Categorical Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The binning map or rules for categorical data are summarized in a "category grouping" table, returned as an optional output. By default, each category is placed in a separate bin. Here is the information for the predictor `ResStatus`.

```
[bi,cg] = bininfo(sc, 'ResStatus')
```

bi =

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682
'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

cg =

Category	BinNumber
'Home Owner'	1
'Tenant'	2
'Other'	3

To group categories 'Tenant' and 'Other', modify the category grouping table `cg`, so the bin number for 'Other' is the same as the bin number for 'Tenant'. Then use `modifybins` to update the scorecard.

```
cg.BinNumber(3) = 2;  
sc = modifybins(sc, 'ResStatus', 'CatGrouping', cg);
```

Display the updated bin information. Note that the bin labels has been updated and that the bin membership information is contained in the category grouping `cg`.

```
[bi, cg] = bininfo(sc, 'ResStatus')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Group1'	365	177	2.0621	0.019329	0.0001682
'Group2'	438	220	1.9909	-0.015827	0.00013772
'Totals'	803	397	2.0227	NaN	0.00030592

```
cg =
```

Category	BinNumber
'Home Owner'	1
'Tenant'	2
'Other'	2

Merge Bins for Numerical and Categorical Predictors

Create a `creditscorecard` object (using a dataset from Refaat 2011).

```
load CreditCardData  
sc = creditscorecard(data, 'IDVar', 'CustID', 'GoodLabel', 0);
```

For the numerical predictor `CustAge`, use the `modifybins` function to set the following cut points:

```
cp = [25 37 49 65];
```

```
sc = modifybins(sc, 'CustAge', 'CutPoints', cp, 'MinValue', 0, 'MaxValue', 75);
bininfo(sc, 'CustAge')
```

ans =

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,25)'	9	8	1.125	-0.58664	0.0052464
'[25,37)'	125	92	1.3587	-0.39789	0.030268
'[37,49)'	340	183	1.8579	-0.084959	0.0031898
'[49,65)'	298	108	2.7593	0.31054	0.030765
'[65,75]'	31	6	5.1667	0.93781	0.022031
'Totals'	803	397	2.0227	NaN	0.0915

Use the modifybins function to merge the 2nd and 3rd bins.

```
sc = modifybins(sc, 'CustAge', 'CutPoints', cp([1 3 4]));
bininfo(sc, 'CustAge')
```

ans =

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,25)'	9	8	1.125	-0.58664	0.0052464
'[25,49)'	465	275	1.6909	-0.17915	0.020355
'[49,65)'	298	108	2.7593	0.31054	0.030765
'[65,75]'	31	6	5.1667	0.93781	0.022031
'Totals'	803	397	2.0227	NaN	0.078397

Display bin information for the categorical predictor ResStatus.

```
[bi,cg] = bininfo(sc, 'ResStatus');
disp(bi)
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682
'Tenant'	307	167	1.8383	-0.095564	0.0036638

'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

Use the `modifybins` function to merge categories 2 and 3.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc, 'ResStatus', 'CatGrouping', cg);
bininfo(sc, 'ResStatus')
```

ans =

Bin	Good	Bad	Odds	WOE	InfoValue
'Group1'	365	177	2.0621	0.019329	0.0001682
'Group2'	438	220	1.9909	-0.015827	0.00013772
'Totals'	803	397	2.0227	NaN	0.00030592

Split Bins for Numerical and Categorical Predictors

Create a `creditscorecard` object (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID', 'GoodLabel', 0)
```

sc =

creditscorecard with properties:

```
    IDVar: 'CustID'
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x9 cell}
```

For the numerical predictor `TmAtAddress`, use the `modifybins` function to set the following cut points:

```
cp = [30 80 120];
sc = modifybins(sc, 'TmAtAddress', 'CutPoints', cp, 'MinValue', 0, 'MaxValue', 210);
```

```
bininfo(sc, 'TmAtAddress')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,30]'	330	154	2.1429	0.057722	0.0013305
'[30,80]'	379	201	1.8856	-0.070187	0.0024086
'[80,120]'	78	36	2.1667	0.068771	0.00044396
'[120,210]'	16	6	2.6667	0.27641	0.0013301
'Totals'	803	397	2.0227	NaN	0.0055131

Use the `modifybins` function to split the 2nd bin.

```
sc = modifybins(sc, 'TmAtAddress', 'CutPoints', [cp(1) 50 cp(2:end)]);
bininfo(sc, 'TmAtAddress')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,30]'	330	154	2.1429	0.057722	0.0013305
'[30,50]'	211	104	2.0288	0.0030488	2.4387e-06
'[50,80]'	168	97	1.732	-0.15517	0.005449
'[80,120]'	78	36	2.1667	0.068771	0.00044396
'[120,210]'	16	6	2.6667	0.27641	0.0013301
'Totals'	803	397	2.0227	NaN	0.0085559

Display bin information for the categorical predictor `ResStatus`.

```
[bi, cg] = bininfo(sc, 'ResStatus')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682

'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

```
cg =
```

Category	BinNumber
'Home Owner'	1
'Tenant'	2
'Other'	3

Use the `modifybins` function to merge categories 2 and 3.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc, 'ResStatus', 'CatGrouping', cg);
bininfo(sc, 'ResStatus')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Group1'	365	177	2.0621	0.019329	0.0001682
'Group2'	438	220	1.9909	-0.015827	0.00013772
'Totals'	803	397	2.0227	NaN	0.00030592

Use the `modifybins` function to split bin 2 and put `Other` under bin 3.

```
cg.BinNumber(3) = 3;
sc = modifybins(sc, 'ResStatus', 'CatGrouping', cg);
[bi, cg] = bininfo(sc, 'ResStatus')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682

'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

cg =

Category	BinNumber
'Home Owner'	1
'Tenant'	2
'Other'	3

Modify Bin Labels

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Use `modifybins` to reset the minimum value and create three bins for the predictor `CustIncome` and display updated bin information.

```
sc = modifybins(sc, 'CustIncome', 'MinValue', 0, 'CutPoints', [30000 50000]);
bi = bininfo(sc, 'CustIncome')
```

bi =

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,30000)'	64	68	0.94118	-0.76504	0.070065
'[30000,50000)'	628	296	2.1216	0.047762	0.0017421
'[50000,Inf]'	111	33	3.3636	0.5086	0.028028
'Totals'	803	397	2.0227	NaN	0.099836

Modify the bin labels and display updated bin information.

```
NewLabels = {'Up to 30k', '30k to 50k', '50k and more'};
```

```
sc = modifybins(sc, 'CustIncome', 'BinLabels', NewLabels);
bi = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Up to 30k'	64	68	0.94118	-0.76504	0.070065
'30k to 50k'	628	296	2.1216	0.047762	0.0017421
'50k and more'	111	33	3.3636	0.5086	0.028028
'Totals'	803	397	2.0227	NaN	0.099836

Bin labels should be the last bin-modification step. As in this example, user-defined bin labels often contain information about the cut points, minimum, or maximum values for numeric data, or information about category groupings for categorical data. To prevent situations where user-defined labels and cut points are inconsistent (and labels are misleading), the `creditscorecard` object overrides user-defined labels every time the bins are modified using `modifybins`.

To illustrate `modifybins` overriding user-defined labels every time the bins are modified, reset the first cut point to 31000 and display updated bin information. Note that the bin labels are reset to their default format and accurately reflect the change in the cut points.

```
sc = modifybins(sc, 'CustIncome', 'CutPoints', [31000 50000]);
bi = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[0,31000)'	82	84	0.97619	-0.72852	0.079751
'[31000,50000)'	610	280	2.1786	0.074251	0.0040364
'[50000,Inf]'	111	33	3.3636	0.5086	0.028028
'Totals'	803	397	2.0227	NaN	0.11182

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

PredictorName — Name of predictor

string

Name of predictor, specified as a string containing the name of the predictor. `PredictorName` is case sensitive.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, `...`, `NameN`, `ValueN`.

Example: `sc = modifybins(sc,PredictorName,'MinValue',10,'CutPoints',[23, 44, 66, 88])`

'MinValue' — Minimum acceptable value (numeric predictors only)

`-Inf` (default) | numeric

Minimum acceptable value, specified as a numeric value (for numeric predictors only). Values below this number are considered out of range.

Data Types: `double`

'MaxValue' — Maximum acceptable value (numeric predictors only)

`Inf` (default) | numeric

Maximum acceptable value, specified as a numeric value (for numeric predictors only). Values above this number are considered out of range.

Data Types: `double`

'CutPoints' — Split points between bins

each observed value of the predictor is placed in a separate bin (default) | nondecreasing numeric array

Split points between bins, specified using a nondecreasing numeric array. If there are `NumBins` bins, there are $n = \text{NumBins} - 1$ cut points so that C_1, C_2, \dots, C_n describe the bin boundaries with the following convention:

- The first bin includes any values $\geq \text{MinValue}$, but $< C_1$.
- The second bin includes any values $\geq C_1$, but $< C_2$.
- The last bin includes any values $\geq C_n$, and $\leq \text{MaxValue}$.

Note: Cut points do not include `MinValue` or `MaxValue`.

By default, cut points are defined so that each observed value of the predictor is placed in a separate bin. If the sorted observed values are V_1, \dots, V_M , the default cut points are V_2, \dots, V_M , which define M bins.

Data Types: `double`

'CatGrouping' — Table with two columns named `Category` and `BinNumber`

each category is placed in a separate bin (default) | table with two columns named `Category` and `BinNumber`

Table with two columns named `Category` and `BinNumber` specified using a table, where the first column contains an exhaustive list of categories for the predictor, and the second column contains the bin number to which each category belongs.

By default, each category is placed in a separate bin. If the observed categories are `'Cat1' ... 'CatM'`, the default category grouping is as follows.

Category	BinNumber
<code>'Cat1'</code>	1
<code>'Cat2'</code>	2
...	...
<code>'CatM'</code>	M

Data Types: double

'BinLabels' — Bin labels for each bin

automatically generated bin labels depending on the predictor's type (default) | cell array of strings

Bin labels for each bin, specified using a cell array of strings with bin label names. Bin labels are used to tag the bins in different object functions such as `bininfo`, `plotbins`, and `displaypoints`. A `creditscorecard` object automatically sets default bins whenever bins are modified. The default format for bin labels depends on the predictor's type.

The format for BinLabels is:

- Numeric data — Before any manual or automatic modification of the predictor bins, there is a bin for each observed predictor value by default. In that case, the bin labels simply show the predictor values. Once the predictor bins have been modified, there are nondefault values for `MinValue` or `MaxValue`, or nondefault cut points `C1`, `C2`, ..., `Cn`. In that case, the bin labels are:
 - Bin 1 label: ' [MinValue, C1) '
 - Bin 2 label: ' [C1, C2) '
 - Last bin label: ' [Cn,MaxValue] '

For example, if there are three bins, `MinValue` is 0 and `MaxValue` is 40, and cut point 1 is 20 and cut point 2 is 30, then the corresponding three bin labels are:

```
' [0,20) '
' [20,30) '
' [30,40] '
```

- Categorical data — For categorical data, before any modification of the predictor bins, there is one bin per category. In that case, the bin labels simply show the predictor categories. Once the bins have been modified, the labels are set to 'Group1', 'Group2', etc., for bin 1, bin 2, etc., respectively. For example, suppose we have the following category grouping

Category	BinNumber
'Cat1'	1
'Cat2'	2
'Cat3'	2

Bin 1 contains 'Cat1' only and its bin label is set to 'Group1'. Bin 2 contains 'Cat2' and 'Cat3' and its bin label is set to 'Group2'.

Tip Using BinLabels should be the last step (if needed) in modifying bins. BinLabels definitions are overridden each time that the bins are modified using the `modifybins` or `autobinning` functions.

Data Types: `cell`

Output Arguments

sc — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object. For more information on using the `creditscorecard` object, see Using `creditscorecard` Objects.

More About

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `plotbins` | `score` | `setmodel`

bininfo

Return predictor's bin information

Syntax

```
bi = bininfo(sc,PredictorName)
```

```
bi = bininfo( ____,Name,Value)
```

```
[bi,bm] = bininfo(sc,PredictorName,Name,Value)
```

Description

`bi = bininfo(sc,PredictorName)` returns information at bin level, such as frequencies of “Good,” “Bad,” and bin statistics for the predictor specified in `PredictorName`.

`bi = bininfo(____,Name,Value)` returns information at bin level, such as frequencies of “Good,” “Bad,” and bin statistics for the predictor specified in `PredictorName` using optional name-value pair arguments.

`[bi,bm] = bininfo(sc,PredictorName,Name,Value)` returns information at bin level, such as frequencies of “Good,” “Bad,” and bin statistics for the predictor specified in `PredictorName` using optional name-value pair arguments. `bininfo` also optionally returns the binning map or bin rules in the form of a vector of cut points for numeric predictors, or a table of category groupings for categorical predictors.

Examples

Display Bin Information Using Default Options

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Display bin information for the categorical predictor `ResStatus`.

```
bi = bininfo(sc, 'ResStatus')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682
'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

Display Bin Information Using Name-Value Arguments

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Display customized bin information for the categorical predictor `ResStatus`, keeping only the `WOE` column.

```
bi = bininfo(sc, 'ResStatus', 'Statistics', 'WOE')
```

```
bi =
```

Bin	Good	Bad	WOE
'Home Owner'	365	177	0.019329
'Tenant'	307	167	-0.095564
'Other'	131	53	0.20049
'Totals'	803	397	NaN

Display customized bin information for the categorical predictor `ResStatus`, keeping only the `Odds` and `WOE` columns, without the `Totals` row.

```
bi = bininfo(sc, 'ResStatus', 'Statistics', {'Odds', 'WOE'}, 'Totals', 'Off')
```



```
bi =
```

Bin	Good	Bad	Odds	WOE
'Home Owner'	365	177	2.0621	0.019329
'Tenant'	307	167	1.8383	-0.095564
'Other'	131	53	2.4717	0.20049

Display Bin Information and Binning Map for Categorical Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The binning map or rules for categorical data are summarized in a "category grouping" table, returned as an optional output. By default, each category is placed in a separate bin. Here is the information for the predictor `ResStatus`.

```
[bi,cg] = bininfo(sc, 'ResStatus')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Home Owner'	365	177	2.0621	0.019329	0.0001682
'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

```
cg =
```

Category	BinNumber
'Home Owner'	1
'Tenant'	2
'Other'	3

To group categories `Tenant` and `Other`, modify the category grouping table `cg` so that the bin number for `Other` is the same as the bin number for `Tenant`. Then use the `modifybins` function to update the scorecard.

```
cg.BinNumber(3) = 2;  
sc = modifybins(sc, 'ResStatus', 'CatGrouping', cg);
```

Display the updated bin information. The bin labels have been updated and that the bin membership information is contained in the category grouping `cg`.

```
[bi, cg] = bininfo(sc, 'ResStatus')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'Group1'	365	177	2.0621	0.019329	0.0001682
'Group2'	438	220	1.9909	-0.015827	0.00013772
'Totals'	803	397	2.0227	NaN	0.00030592

```
cg =
```

Category	BinNumber
'Home Owner'	1
'Tenant'	2
'Other'	2

Display Bin Information and Binning Map for Numeric Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData  
sc = creditscorecard(data);
```

The predictor `CustIncome` is numeric. By default, each value of the predictor is placed in a separate bin.

```
bi = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'18000'	2	3	0.66667	-1.1099	0.0056227
'19000'	1	2	0.5	-1.3976	0.0053002
'20000'	4	2	2	-0.011271	6.3641e-07
'21000'	6	3	2	-0.011271	9.5462e-07
'22000'	4	2	2	-0.011271	6.3641e-07
'23000'	4	4	1	-0.70442	0.0035885
'24000'	5	5	1	-0.70442	0.0044856
'25000'	4	9	0.44444	-1.5153	0.026805
'26000'	4	11	0.36364	-1.716	0.038999
'27000'	6	6	1	-0.70442	0.0053827
'28000'	13	11	1.1818	-0.53736	0.0061896
'29000'	11	10	1.1	-0.60911	0.0069988
'30000'	18	16	1.125	-0.58664	0.010493
'31000'	24	8	3	0.39419	0.0038382
'32000'	21	15	1.4	-0.36795	0.0042797
'33000'	35	19	1.8421	-0.093509	0.00039951
'34000'	33	17	1.9412	-0.041124	7.095e-05
'35000'	39	20	1.95	-0.036589	6.6225e-05
'36000'	30	12	2.5	0.21187	0.0015113
'37000'	57	22	2.5909	0.24759	0.0038545
'38000'	28	22	1.2727	-0.46326	0.0095182
'39000'	39	22	1.7727	-0.1319	0.00090321
'40000'	42	17	2.4706	0.20004	0.0018969
'41000'	26	17	1.5294	-0.27954	0.0029191
'42000'	39	16	2.4375	0.18655	0.001542
'43000'	39	14	2.7857	0.32009	0.0042582
'44000'	27	12	2.25	0.10651	0.00036184
'45000'	35	12	2.9167	0.36602	0.00489
'46000'	24	12	2	-0.011271	3.8185e-06
'47000'	22	6	3.6667	0.59486	0.0073073
'48000'	35	9	3.8889	0.65371	0.013673
'49000'	15	8	1.875	-0.07581	0.00011153
'50000'	26	3	8.6667	1.4551	0.036118
'51000'	14	7	2	-0.011271	2.2274e-06
'52000'	17	5	3.4	0.51936	0.0044541
'53000'	15	6	2.5	0.21187	0.00075566
'54000'	10	3	3.3333	0.49955	0.0024461

'55000'	6	0	Inf	Inf	Inf
'56000'	9	2	4.5	0.79966	0.004934
'57000'	2	2	1	-0.70442	0.0017942
'58000'	1	2	0.5	-1.3976	0.0053002
'59000'	3	2	1.5	-0.29895	0.00038918
'60000'	4	0	Inf	Inf	Inf
'61000'	3	0	Inf	Inf	Inf
'62000'	1	1	1	-0.70442	0.00089712
'Totals'	803	397	2.0227	NaN	Inf

Reduce the number of bins using the `autobinning` function (the `modifybins` function can also be used).

```
sc = autobinning(sc, 'CustIncome');
```

Display the updated bin information. The binning map or rules for numeric data are summarized as "cut points," returned as an optional output (`cp`).

```
[bi,cp] = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,33000)'	127	107	1.1869	-0.53306	0.059364
'[33000,38000)'	194	90	2.1556	0.06363	0.00094769
'[38000,42000)'	135	78	1.7308	-0.15585	0.004419
'[42000,47000)'	164	66	2.4848	0.20579	0.0078175
'[47000,Inf]'	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.11421

```
cp =
```

```
33000
38000
42000
47000
```

Manually remove the second cut point (the boundary between the second and third bins) to merge bins two and three. Use the `modifybins` function to update the scorecard.

```
cp(2) = [];
sc = modifybins(sc, 'CustIncome', 'CutPoints', cp);
```

Display the updated bin information.

```
[bi,cp] = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
' [-Inf,33000) '	127	107	1.1869	-0.53306	0.059364
' [33000,42000) '	329	168	1.9583	-0.032325	0.0004351
' [42000,47000) '	164	66	2.4848	0.20579	0.0078175
' [47000,Inf] '	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.10927

```
cp =
```

```
33000
42000
47000
```

Note, it is recommended to avoid having bins with frequencies of zero because they lead to infinite or undefined (NaN) statistics. Use the `modifybins` or `autobinning` functions to modify bins.

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

PredictorName — Predictor name

string

Predictor name, specified using a string containing the name of the predictor. PredictorName is case sensitive.

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, . . . ,NameN,ValueN.

```
Example: bi = bininfo(sc,  
PredictorName,'Statistics','WOE','Totals','On')
```

'Statistics' — List of statistics to include for bin information

{ 'Odds', 'WOE', 'InfoValue' } (default) | strings with values 'Odds', 'WOE', 'InfoValue' | cell array of strings with values 'Odds', 'WOE', 'InfoValue'

List of statistics to include in the bin information, specified as a string or a cell array of strings. For more information, see “Statistics for a Credit Scorecard”. Possible values are:

- 'Odds' — Odds information is the ratio of “Good” to “Bad.”
- 'WOE' — Weight of Evidence. WOE is only defined at a bin level and there is no definition for a “Total WOE.” The output for bin information table (bi) sets the Total WOE to NaN.
- 'InfoValue' — Information value. The computation of InfoValue does ignore bins that are empty (neither “Good” nor “Bad” observations), because it uses a 'nansum' to compute the Total Information Value. However, bins with only “Good” or only “Bad” observations do lead to an infinite Information Value. Consider modifying the bins in those cases by using `modifybins` or `autobinning`.

Note: It is recommended to avoid having bins with frequencies of zero because they lead to infinite or undefined (NaN) statistics. Use `modifybins` or `autobinning` to modify bins.

Data Types: char | cell

'Totals' — Indicator to include row of totals at bottom information table

'On' (default) | string with values 'On', 'Off'

Indicator to include a row of totals at the bottom of the information table, specified as a string with values On or Off.

Data Types: char

Output Arguments

bi — Bin information

table

Bin information, returned as a table. The bin information table contains one row per bin and a row of totals. The columns contain bin descriptions, frequencies of “Good” and “Bad,” and bin statistics. It is recommended to avoid having bins with frequencies of zero because they lead to infinite or undefined (NaN) statistics. Use `modifybins` or `autobinning` to modify bins.

bm — Binning map or rules

vector of cut points for numeric predictors | table of category groupings for categorical predictors

Binning map or rules, returned as a vector of cut points for numeric predictors, or a table of category groupings for categorical predictors. For more information, see `modifybins`.

More About

Statistics for a Credit Scorecard

Weight of Evidence (WOE) is a measure of the difference of the distribution of “Goods” and “Bads” within a bin.

Suppose the predictor's data takes on M possible values b_1, \dots, b_M . For binned data, M is a small number. The response takes on two values, “Good” and “Bad.” The frequency table of the data is given by:

	Good	Bad	Total
b1:	n11	n12	n1
b2:	n21	n22	n2
bM:	nM1	nM2	nM
Total:	nGood	nBad	nTotal

The Weight of Evidence (WOE) is defined for each data value b_i as

$$\text{WOE}(i) = \log((n_{i1}/n_{\text{Good}})/(n_{i2}/n_{\text{Bad}})).$$

If you define

$$p_{\text{Good}}(i) = n_{i1}/n_{\text{Good}}, \quad p_{\text{Bad}}(i) = n_{i2}/n_{\text{Bad}}$$

then $p_{\text{Good}}(i)$ is the proportion of “Good” observations that take on the value b_i , and similarly for $p_{\text{Bad}}(i)$. In other words, $p_{\text{Good}}(i)$ gives the distribution of good observations over the M observed values of the predictor, and similarly for $p_{\text{Bad}}(i)$. With this, an equivalent formula for the WOE is

$$\text{WOE}(i) = \log(p_{\text{Good}}(i)/p_{\text{Bad}}(i)).$$

Using the same frequency table, the odds for row i are defined as

$$\text{Odds}(i) = n_{i1} / n_{i2},$$

and the odds for the sample are defined as

$$\text{OddsTotal} = n_{\text{Good}} / n_{\text{Bad}}.$$

For each row i , you can also compute its contribution to the total Information Value, given by

$$\text{InfoValue}(i) = (p_{\text{Good}}(i) - p_{\text{Bad}}(i)) * \text{WOE}(i),$$

and the total Information Value is simply the sum of all the $\text{InfoValue}(i)$ terms. (A `nansum` is returned to discard contributions from rows with no observations at all.)

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

autobinning | bindata | creditscorecard | displaypoints | fitmodel |
formatpoints | modifybins | plotbins | score | setmodel

autobinning

Perform automatic binning of given predictors

Syntax

```
sc = autobinning(sc,PredictorNames)
sc = autobinning( ___,Name,Value)
```

Description

`sc = autobinning(sc,PredictorNames)` performs automatic binning of the predictors given in `PredictorNames`.

Automatic binning finds binning maps or rules to bin numeric data and to group categories of categorical data. The binning rules are stored in the `creditscorecard` object. To apply the binning rules to the `creditscorecard` object data, or to a new dataset, use `bindata`.

`sc = autobinning(___,Name,Value)` performs automatic binning of the predictors given in `PredictorNames` using optional name-value pair arguments. See the name-value argument `Algorithm` for a description of the supported binning algorithms.

Automatic binning finds binning maps or rules to bin numeric data and to group categories of categorical data. The binning rules are stored in the `creditscorecard` object. To apply the binning rules to the `creditscorecard` object data, or to a new dataset, use `bindata`.

Examples

Perform Automatic Binning Using the Defaults

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the PredictorNames called CustIncome using the default options. By default, autobinning uses the EqualFrequency algorithm with five bins.

```
sc = autobinning(sc, 'CustIncome');
```

Use bininfo to display the binned data.

```
bi = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,33000)'	127	107	1.1869	-0.53306	0.059364
'[33000,38000)'	194	90	2.1556	0.06363	0.00094769
'[38000,42000)'	135	78	1.7308	-0.15585	0.004419
'[42000,47000)'	164	66	2.4848	0.20579	0.0078175
'[47000,Inf]'	183	56	3.2679	0.47972	0.041657
'Totals'	803	397	2.0227	NaN	0.11421

Perform Automatic Binning Using Name-Value Pair Arguments

Create a creditcard object using the CreditCardData.mat file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditcard(data);
```

Perform automatic binning for the PredictorNames called CustIncome using the EqualFrequency algorithm with four bins. This example explicitly sets both the Algorithm and the AlgorithmOptions name-value arguments. However, because EqualFrequency is the default choice for the Algorithm name-value parameter, the use of the Algorithm parameter is optional in this case.

```
AlgoOptions = {'NumBins',4};
sc = autobinning(sc, 'CustIncome', 'Algorithm', 'EqualFrequency', ...
'AlgorithmOptions', AlgoOptions);
```

Use bininfo to display the binned data. Here, the cut points, which delimit the bins, are also displayed.

```
[bi,cp] = bininfo(sc, 'CustIncome')
```

```
bi =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'[-Inf,34000)'	162	126	1.2857	-0.4531	0.052396
'[34000,40000)'	226	115	1.9652	-0.028816	0.00023709
'[40000,45000)'	173	76	2.2763	0.11814	0.0028361
'[45000,Inf]'	242	80	3.025	0.40249	0.040192
'Totals'	803	397	2.0227	NaN	0.095661

```
cp =
```

```
34000
40000
45000
```

Perform Automatic Binning for Multiple Predictors

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the `PredictorNames` called `CustIncome` and `CustAge` using the default options. By default, `autobinning` uses the `EqualFrequency` algorithm with five bins.

```
sc = autobinning(sc, {'CustAge', 'CustIncome'});
```

Use `bininfo` to display the binned data.

```
bi1 = bininfo(sc, 'CustIncome')
bi2 = bininfo(sc, 'CustAge')
```

```
bi1 =
```

Bin	Good	Bad	Odds	WOE	InfoValue
-----	------	-----	------	-----	-----------

' [-Inf,33000) '	127	107	1.1869	-0.53306	0.059364
' [33000,38000) '	194	90	2.1556	0.06363	0.00094769
' [38000,42000) '	135	78	1.7308	-0.15585	0.004419
' [42000,47000) '	164	66	2.4848	0.20579	0.0078175
' [47000,Inf] '	183	56	3.2679	0.47972	0.041657
' Totals '	803	397	2.0227	NaN	0.11421

bi2 =

Bin	Good	Bad	Odds	WOE	InfoValue
' [-Inf,37) '	134	100	1.34	-0.41175	0.035005
' [37,43) '	154	86	1.7907	-0.12181	0.0030263
' [43,48) '	154	80	1.925	-0.049492	0.00048159
' [48,54) '	180	66	2.7273	0.29888	0.017309
' [54,Inf] '	181	65	2.7846	0.31969	0.019718
' Totals '	803	397	2.0227	NaN	0.075539

Perform Automatic Binning for a Categorical Predictor

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the `PredictorNames` that is a categorical predictor called `ResStatus` using the default options. By default, `autobinning` uses the `EqualFrequency` algorithm with five bins.

```
sc = autobinning(sc, 'ResStatus');
```

Use `bininfo` to display the binned data.

```
bi = bininfo(sc, 'ResStatus')
```

bi =

Bin	Good	Bad	Odds	WOE	InfoValue
-----	------	-----	------	-----	-----------

'Home Owner'	365	177	2.0621	0.019329	0.0001682
'Tenant'	307	167	1.8383	-0.095564	0.0036638
'Other'	131	53	2.4717	0.20049	0.0059418
'Totals'	803	397	2.0227	NaN	0.0097738

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

sc — Credit scorecard model

creditscorecard object

Credit scorecard model, specified as a `creditscorecard` object. The `creditscorecard` function must be used to create a `creditscorecard` object.

PredictorNames — Predictor or predictors names to automatically bin

string | cell array of strings

Predictor or predictors names to automatically bin, specified as a string or a cell array of strings containing the name of the predictor or predictors. `PredictorNames` are case-sensitive and when no `PredictorNames` are defined, all predictors in the `PredictorVars` property of the `creditscorecard` object are binned.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `sc = autobinning(sc, 'Algorithm', 'EqualFrequency')`

'Algorithm' — Algorithm selection

`'EqualFrequency'` (default) | string with values `'EqualFrequency'`, `'EqualWidth'`

Algorithm selection, specified using a string indicating which algorithm to use. The same algorithm is used for all predictors in PredictorNames). Possible values are:

- **EqualFrequency** — Unsupervised algorithm that divides the data into a predetermined number of bins that contain approximately the same number of observations. This algorithm is also known as “equal height” or “equal depth.” For more information, see “Equal Frequency”.
- **EqualWidth** — Unsupervised algorithm that divides the range of values in the domain of the predictor variable into a predetermined number of bins of “equal width.” For numeric data, the width is measured as the distance between bin edges. For categorical data, width is measured as the number of categories within a bin. For more information, see “Equal Width”.

Data Types: char

'AlgorithmOptions' — Algorithm options for selected Algorithm

{'NumBins',5} for EqualFrequency (default) | cell array with {'NumBins',*n*} for Algorithm options for EqualFrequency and EqualWidth

Algorithm options for the selected Algorithm, specified using a cell array. Possible values are:

- For EqualFrequency algorithm:
 - {'NumBins',*n*} — Specifies the desired number (*n*) of bins. The default is {'NumBins',5} and the number of bins must be a positive number.
- For EqualWidth algorithm:
 - {'NumBins',*n*} — Specifies the desired number (*n*) of bins. The default is {'NumBins',5} and the number of bins must be a positive number.

Example: `sc =
autobinning(sc,'CustAge','Algorithm','EqualFrequency','AlgorithmOptions',
{'NumBins',6})`

Data Types: cell

Output Arguments

sc — Credit scorecard model
creditscorecard object

Credit scorecard model, returned as an updated `creditscorecard` object containing the automatically determined binning maps or rules (cut points or category groupings) for one or more predictors. For more information on using the `creditscorecard` object, see Using `creditscorecard` Objects.

Note: If you have previously used the `modifybins` function to manually modify bins, these changes will be lost when running `autobinning` because all of the data is automatically binned based on internal autobinning rules.

More About

Equal Frequency

Unsupervised algorithm that divides the data into a predetermined number of bins that contain approximately the same number of observations.

`EqualFrequency` is defined as:

Let $v[1], v[2], \dots, v[N]$ be the sorted list of different values or categories observed in the data. Let $f[i]$ be the frequency of $v[i]$. Let $F[k] = f[1] + \dots + f[k]$ be the cumulative sum of frequencies up to the k th sorted value. Then $F[N]$ is the same as the total number of observations.

Define $\text{AvgFreq} = F[N] / \text{NumBins}$, which is the ideal average frequency per bin after binning. The n th cut point index is the index k such that the distance $\text{abs}(F[k] - n * \text{AvgFreq})$ is minimized.

This rule attempts to match the cumulative frequency up to the n th bin. If a single value contains too many observations, equal frequency bins are not possible, and the above rule yields less than `NumBins` total bins. In that case, the algorithm determines `NumBins` bins by breaking up bins, in the order in which the bins were constructed.

Equal Width

Unsupervised algorithm that divides the range of values in the domain of the predictor variable into a predetermined number of bins of “equal width.” For numeric data, the width is measured as the distance between bin edges. For categorical data, width is measured as the number of categories within a bin.

The `EqualWidth` option is defined as:

For numeric data, if `MinValue` and `MaxValue` are the minimum and maximum data values, then

`Width = (MaxValue - MinValue) / NumBins`

and the `CutPoints` are set to `MinValue + Width`, `MinValue + 2*Width`, ... `MaxValue - Width`. If a `MinValue` or `MaxValue` have not been specified using the `modifybins` function, the `EqualWidth` option sets `MinValue` and `MaxValue` to the minimum and maximum values observed in the data.

For categorical data, if there are `NumCats` number of original categories, then

`Width = NumCats / NumBins`,

and set cut point indices to the rounded values of `Width`, `2*Width`, ..., `NumCats - Width`, plus 1.

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”
- Using `creditscorecard` Objects

References

Refaat, M., *Data Preparation for Data Mining Using SAS*, Morgan Kaufmann, 2006.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

creditscorecard

Create `creditscorecard` object

Build a credit scorecard model by creating a `creditscorecard` object and specifying input data in a table format. For more information on using a `creditscorecard` object, see [Using creditscorecard Objects](#).

After creating a `creditscorecard` object, you can use the associated object functions to bin the data and perform logistic regression analysis to develop a credit scorecard model to guide credit decisions. This workflow shows how to develop a credit scorecard model.

- 1 Create a `creditscorecard` object.
- 2 Bin the data.
- 3 Fit a logistic regression model.
- 4 Review and format the credit scorecard points.
- 5 Score the data.

For details on this workflow, see “[Credit Scorecard Modeling Workflow](#)”.

Syntax

```
sc = creditscorecard(data)
sc = creditscorecard( ____, Name, Value)
```

Description

`sc = creditscorecard(data)` creates a `creditscorecard` object by specifying data.

`sc = creditscorecard(____, Name, Value)` creates a `creditscorecard` object by specifying data and using optional name-value pair arguments.

Examples

Create a creditscorecard Object

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data)
```

```
sc =

creditscorecard with properties:

    IDVar: ''
   GoodLabel: 0
  ResponseVar: 'status'
   VarNames: {1x11 cell}
 PredictorVars: {1x10 cell}
```

In this example, the default values for the properties `ResponseVar`, `PredictorVars` and `GoodLabel` are assigned when this object is created. By default, the property `ResponseVar` is set to the variable name that is in the last column of the input data ('`status`' in this example). The property `PredictorVars` contains the names of all the variables that are in `VarNames`, but excludes `IDVar` and `ResponseVar`. Also, by default in the previous example, `GoodLabel` is set to 0, since it is the value in the response variable (`ResponseVar`) with the highest count.

Display the `creditscorecard` object properties using dot notation.

```
sc.PredictorVars
sc.VarNames
```

```
ans =

Columns 1 through 5

    'CustID'    'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'

Columns 6 through 10
```

```

        'CustIncome'    'TmWBank'    'OtherCC'    'AMBalance'    'UtilRate'

ans =

Columns 1 through 5

    'CustID'    'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'

Columns 6 through 10

    'CustIncome'    'TmWBank'    'OtherCC'    'AMBalance'    'UtilRate'

Column 11

    'status'

```

Since `IDVar` and `PredictorVars` have public access, you can change their values at the command line.

```

sc.IDVar = 'CustID'
sc.PredictorVars = {'CustIncome', 'ResStatus', 'AMBalance'}
disp(sc)

```

```

sc =

creditscorecard with properties:

    IDVar: 'CustID'
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x9 cell}

```

```

sc =

creditscorecard with properties:

    IDVar: 'CustID'
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}

```

```

    PredictorVars: {'CustIncome' 'ResStatus' 'AMBalance'}

creditscorecard with properties:

    IDVar: 'CustID'
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {'CustIncome' 'ResStatus' 'AMBalance'}

```

Create a creditscorecard Object and Set GoodLabel and ResponseVar

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Then use name-value pair arguments for the `creditscorecard` function to define `GoodLabel` and `ResponseVar`.

```

load CreditCardData
sc = creditscorecard(data, 'IDVar', 'CustID', 'GoodLabel', 0, 'ResponseVar', 'status')

```

```

sc =

creditscorecard with properties:

    IDVar: 'CustID'
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x9 cell}

```

`GoodLabel` and `ResponseVar` can only be set (enforced) when creating a `creditscorecard` object using the `creditscorecard` function.

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Input Arguments

data — Data for `creditscorecard` object
table

Data for the `creditscorecard` object, specified as a table, where each column of data can be any one of the following data types:

- Numeric
- Logical
- Cell array of strings
- Character array
- Categorical

In addition, the table must contain a binary response variable. Before creating a `creditscorecard` object, you must perform a data preparation task to have appropriately structured data as input to a `creditscorecard` object.

Data Types: `table`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `sc = creditscorecard(data, 'GoodLabel', 0, 'IDVar', 'CustID', 'ResponseVar', 'status', 'Pr {'CustID', 'CustIncome'})`

'GoodLabel' — Indicator for which of two possible values in response variable correspond to “Good” observations

set to the response value with the highest count (default) | string | numeric scalar | logical

Indicator for which of the two possible values in the response variable correspond to “Good” observations, specified as a numeric scalar, logical, or string. When specifying `GoodLabel`, follow these guidelines.

If Response Is...	GoodLabel Must be...
numeric	numeric
logical	logical or numeric
cell array of strings	string

If Response Is...	GoodLabel Must be...
character array	string
categorical	string

If not specified, GoodLabel is set to the response value with the highest count. GoodLabel can only be set when creating the `creditscorecard` object. This parameter cannot be set using dot notation.

Data Types: char | double

'IDVar' — Variable name used as ID or tag for observations

empty string '' (default) | string

Variable name used as ID or tag for the observations, specified as a string. The IDVar data could be an ordinal number (for example, 1,2,3...), a Social Security number, and so on. This is provided as a convenience to remove this column from the predictor variables. IDVar is case sensitive. You can set this optional parameter using the `creditscorecard` function or by using dot notation at the command line, as follows.

Example: `sc.IDVar = 'CustID'`

Data Types: char

'ResponseVar' — Response variable name for "Good" or "Bad" indicator

last column of the data input (default) | string

Response variable name for the "Good" or "Bad" indicator, specified as a string. The response variable data must be binary. If not specified, ResponseVar is set to the last column of the data input. ResponseVar can only be set when creating the `creditscorecard` object using the `creditscorecard` function. ResponseVar is case sensitive.

Data Types: char

'PredictorVars' — Predictor variable names

set difference between VarNames and {IDVar,ResponseVar} (default) | cell array of strings containing names

Predictor variable names, specified using a cell array of strings containing names. By default, when you create a credit scorecard using `creditscorecard`, all variables are predictors except for IDVar and ResponseVar. This property can be modified by using dot notation or by using a name-value pair argument for the `fitmodel` function.

PredictorVars is case sensitive and the predictor variable name cannot be the same as the IDVar or ResponseVar.

Data Types: `cell`

Output Arguments

sc — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as a `creditscorecard` object that contains the binning maps or rules (cut points or category groupings) for one or more predictors. For more information on using the `creditscorecard` object, see [Using creditscorecard Objects](#).

More About

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”

References

Anderson, R., *The Credit Scoring Toolkit*, Oxford University Press, 2007.

Refaat, M., *Credit Risk Scorecards: Development and Implementation Using SAS*, lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `plotbins` | `score` | `setmodel`

Using creditcorecard Objects

Build credit scorecard model

Build a credit scorecard model by creating a `creditcorecard` object and specify input data in a table format.

After creating a `creditcorecard` object, you can use the associated object functions to bin the data and perform logistic regression analysis to develop a credit scorecard model to guide credit decisions. This workflow shows how to develop a credit scorecard model.

- 1 Create a `creditcorecard` object.
- 2 Bin the data.
- 3 Fit a logistic regression model.
- 4 Review and format the credit scorecard points.
- 5 Score the data.

For more detailed information on this workflow, see “Credit Scorecard Modeling Workflow”.

Examples

Create a creditcorecard Object

Create a `creditcorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditcorecard(data)
```

```
sc =
```

```
creditcorecard with properties:
```

```
    IDVar: ''
   GoodLabel: 0
  ResponseVar: 'status'
    VarNames: {1x11 cell}
```

```
PredictorVars: {1x10 cell}
```

Display `creditscorecard` Object Properties

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data)
```

```
sc =
```

```
creditscorecard with properties:
```

```
    IDVar: ''
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x10 cell}
```

To display the `creditscorecard` object properties, use dot notation.

```
sc.PredictorVars
sc.VarNames
```

```
ans =
```

```
Columns 1 through 5
```

```
'CustID'    'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'
```

```
Columns 6 through 10
```

```
'CustIncome'    'TmWBank'    'OtherCC'    'AMBalance'    'UtilRate'
```

```
ans =
```

```
Columns 1 through 5
```

```
'CustID'    'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'
```

```
Columns 6 through 10
```

```
    'CustIncome'    'TmWBank'    'OtherCC'    'AMBalance'    'UtilRate'
```

```
Column 11
```

```
    'status'
```

Change a Property Value for a creditcorecard Object

Create a `creditcorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditcorecard(data)
```

```
sc =
```

```
creditcorecard with properties:
```

```
    IDVar: ''
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x10 cell}
```

Since the `IDVar` property has public access, you can change its value at the command line.

```
sc.IDVar = 'CustID'
```

```
sc =
```

```
creditcorecard with properties:
```

```
    IDVar: 'CustID'
    GoodLabel: 0
    ResponseVar: 'status'
    VarNames: {1x11 cell}
    PredictorVars: {1x9 cell}
```

- “Case Study for a Credit Scorecard Analysis”
- “Troubleshooting Credit Scorecard Results”

Properties

IDVar — Name of the variable used as ID or tag for the observations

empty string ‘ ’ (default) | string

Name of the variable used as ID or tag for the observations, specified as a string. This property can be set as an optional parameter using the `creditscorecard` function or by using dot notation at the command line. IDVar is case sensitive.

Example: `sc.IDVar = 'CustID'`

Data Types: char

VarNames — All variable names from the data input

VarNames come directly from data input to `creditscorecard` object (default)

VarNames is a cell array of strings containing the names of all variables in the data. The VarNames come directly from the data input to the `creditscorecard` object. VarNames is case sensitive.

Example:

Data Types: cell

ResponseVar — Name of the response variable, “Good” or “Bad” indicator

last column of the data input (default) | string

Name of the response variable, “Good” or “Bad” indicator, specified as a string. The response variable data must be binary. If not specified, ResponseVar is set to the last column of the data input. This property can only be set with an optional parameter when using the `creditscorecard` function. ResponseVar is case sensitive.

Data Types: char

GoodLabel — Indicator for which of the two possible values in the response variable correspond to “Good” observations

set to the response value with the highest count (default) | string | numeric scalar | logical

Indicator for which of the two possible values in the response variable correspond to “Good” observations. When specifying GoodLabel, follow these guidelines:

If Response is...	GoodLabel must be:
numeric	numeric
logical	logical or numeric
cell array of strings	string
character array	string
categorical	string

If not specified, GoodLabel is set to the response value with the highest count. This property can only be set with an optional parameter when using the `creditcorecard` function. This property cannot be set using dot notation.

Data Types: `char` | `double`

PredictorVars — Predictor variable names

set difference between `VarNames` and `{IDVar,ResponseVar}` (default) | cell array of strings containing names

Predictor variable names, specified using a cell array of strings containing names. By default, when you create a credit scorecard using `creditcorecard`, all variables are predictors except for `IDVar` and `ResponseVar`. This property can be modified using a name-value pair argument for the `fitmodel` function or by using dot notation or with the `fitmodel` function. `PredictorVars` is case sensitive and the predictor variable name cannot be the same as the `IDVar` or `ResponseVar`.

Example: `sc.PredictorVars = {'CustID','CustIncome'}`

Data Types: `cell`

creditcorecard Property	Set/Modify Property From Command Line Using creditcorecard Function	Modify Property Using Dot Notation	Property Not User-Defined and Value is Defined Internally
IDVar	Yes	Yes	N/A
VarNames	No	No	Yes

creditscorecard Property	Set/Modify Property From Command Line Using creditscorecard Function	Modify Property Using Dot Notation	Property Not User-Defined and Value is Defined Internally
ResponseVar	Yes	No	If not specified, set to last column of <code>data</code> input
GoodLabel	Yes	No	If not specified, set to response value with highest count
PredictorVars	Yes (also modifiable using <code>fitmodel</code> function)	Yes	N/A

Object Functions

creditscorecard autobinningbininfomodifybins bindataplotbins
fitmodelsetmodeldisplaypoints formatpointsscore

Create Object

To create a `creditscorecard` object, use the `creditscorecard` function.

See Also

creditscorecard

More About

- “Credit Scorecard Modeling Workflow”
- “About Credit Scorecards”

Bibliography

- “Bond Pricing and Yields” on page A-2
- “Term Structure of Interest Rates” on page A-3
- “Derivatives Pricing and Yields” on page A-4
- “Portfolio Analysis” on page A-5
- “Investment Performance Metrics” on page A-6
- “Financial Statistics” on page A-7
- “Standard References” on page A-8
- “Credit Risk Analysis” on page A-9
- “Portfolio Optimization” on page A-10
- “Stochastic Differential Equations” on page A-11

Note: For the well-known algorithms and formulas used in Financial Toolbox software (such as how to compute a loan payment given principal, interest rate, and length of the loan), no references are given here. The references here pertain to less common formulas.

Bond Pricing and Yields

The pricing and yield formulas for fixed-income securities come from:

[1] Golub, B.W. and L.M. Tilman, *Risk Management: Approaches for Fixed Income Markets* Wiley, 2000.

[2] Martellini, L., P. Priaulet, and S. Priaulet *Fixed Income Securities* Wiley, 2003.

[3] Mayle, Jan, *Standard Securities Calculation Methods* New York: Securities Industry Association, Inc. Vol. 1, 3rd ed., 1993, ISBN 1-882936-01-9. Vol. 2, 1994, ISBN 1-882936-02-7.

[4] Tuckman, B. *Fixed Income Securities: Tools for Today's Markets* Wiley, 2002.

In many cases these formulas compute the price of a security given yield, dates, rates, and other data. These formulas are nonlinear, however; so when solving for an independent variable within a formula, Financial Toolbox software uses Newton's method. See any elementary numerical methods textbook for the mathematics underlying Newton's method.

Term Structure of Interest Rates

The formulas and methodology for term structure functions come from:

[5] Fabozzi, Frank J., “The Structure of Interest Rates.” Ch. 6 in Fabozzi, Frank J. and T. Dossa Fabozzi, eds. *The Handbook of Fixed Income Securities*. 4th ed. New York: Irwin Professional Publishing, 1995, ISBN 0-7863-0001-9.

[6] McEnally, Richard W. and James V. Jordan, “The Term Structure of Interest Rates.” Ch. 37 in Fabozzi and Fabozzi, *ibid*.

[7] Das, Satyajit, “Calculating Zero Coupon Rates.” *Swap and Derivative Financing*. Appendix to Ch. 8, pp. 219-225, New York: Irwin Professional Publishing., 1994, ISBN 1-55738-542-4.

Derivatives Pricing and Yields

The pricing and yield formulas for derivative securities come from:

[8] Chriss, Neil A., “Black-Scholes and Beyond: Option Pricing Models,” Chicago: Irwin Professional Publishing, 1997, ISBN 0-7863-1025-1.

[9] Cox, J., S. Ross, and M. Rubenstein, “Option Pricing: A Simplified Approach”, *Journal of Financial Economics* 7, Sept. 1979, pp. 229 - 263.

[10] Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003, ISBN 0-13-009056-5.

Portfolio Analysis

The Markowitz model is used for portfolio analysis computations. For a discussion of this model see Chapter 7 of:

[11] Bodie, Zvi, Alex Kane, and Alan J. Marcus, *Investments*, Burr Ridge, IL: Irwin. 2nd. ed., 1993, ISBN 0-256-08342-8.

Investment Performance Metrics

The risk and ratio formulas for investment performance metrics come from:

[12] Daniel Bernoulli, "Exposition of a New Theory on the Measurement of Risk," *Econometrica*, Vol. 22, No 1, January 1954, pp. 23-36 (English translation of "Specimen Theoriae Novae de Mensura Sortis," *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, Tomus V, 1738, pp. 175-192).

[13] Martin Eling and Frank Schuhmacher, *Does the Choice of Performance Measure Influence the Evaluation of Hedge Funds?*, Working Paper, November 2005.

[14] John Lintner, "The Valuation of Risk Assets and the Selection of Risky Investments in Stocks Portfolios and Capital Budgets," *Review of Economics and Statistics*, Vol. 47, No. 1, February 1965, pp. 13-37.

[15] Malik Magdon-Ismail, Amir F. Atiya, Amrit Pratap, and Yaser S. Abu-Mostafa, "On the Maximum Drawdown of a Brownian Motion," *Journal of Applied Probability*, Volume 41, Number 1, March 2004, pp. 147-161.

[16] Malik Magdon-Ismail and Amir Atiya, "Maximum Drawdown," <http://www.risk.net/risk-magazine>, October 2004.

[17] Harry Markowitz, "Portfolio Selection," *Journal of Finance*, Vol. 7, No. 1, March 1952, pp. 77-91.

[18] Harry Markowitz, *Portfolio Selection: Efficient Diversification of Investments*, John Wiley & Sons, 1959.

[19] Jan Mossin, "Equilibrium in a Capital Asset Market," *Econometrica*, Vol. 34, No. 4, October 1966, pp. 768-783.

[20] Christian S. Pedersen and Ted Rudholm-Alfvén, "Selecting a Risk-Adjusted Shareholder Performance Measure," *Journal of Asset Management*, Vol. 4, No. 3, 2003, pp. 152-172.

[21] William F. Sharpe, "Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk," *Journal of Finance*, Vol. 19, No. 3, September 1964, pp. 425-442.

[22] Katerina Simons, "Risk-Adjusted Performance of Mutual Funds," *New England Economic Review*, September/October 1998, pp. 34-48.

Financial Statistics

The discussion of computing statistical values for portfolios containing missing data elements derives from the following references:

[23] Little, Roderick J.A. and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[24] Meng, Xiao-Li, and Donald B. Rubin, "Maximum Likelihood Estimation via the ECM Algorithm," *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

[25] Sexton, Joe and Anders Rygh Swensen, "ECM Algorithms That Converge at the Rate of EM," *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

[26] Dempster, A.P., N.M. Laird, and Donald B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, 1977, pp. 1-37.

Standard References

Standard references include:

[27] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures*, Vol. 2, Spring 1995. This addendum explains and clarifies the end-of-month rule.

[28] Brealey, Richard A. and Stewart C. Myers, *Principles of Corporate Finance*, New York: McGraw-Hill. 4th ed., 1991, ISBN 0-07-007405-4.

[29] Daigler, Robert T., *Advanced Options Trading*. Chicago: Probus Publishing Co., 1994, ISBN 1-55738-552-1.

[30] *A Dictionary of Finance*. Oxford: Oxford University Press., 1993, ISBN 0-19-285279-5.

[31] Fabozzi, Frank J. and T. Dossa Fabozzi, eds. *The Handbook of Fixed-Income Securities*. Burr Ridge, IL: Irwin. 4th ed., 1995, ISBN 0-7863-0001-9.

[32] Fitch, Thomas P., *Dictionary of Banking Terms*. Hauppauge, NY: Barron's. 2nd ed., 1993, ISBN 0-8120-1530-4.

[33] Hill, Richard O., Jr., *Elementary Linear Algebra*. Orlando, FL: Academic Press. 1986, ISBN 0-12-348460-X.

[34] Luenberger, David G., *Investment Science*, Oxford University Press, 1998. ISBN 0195108094.

[35] Marshall, John F. and Vipul K. Bansal, *Financial Engineering: A Complete Guide to Financial Innovation*. New York: New York Institute of Finance. 1992, ISBN 0-13-312588-2.

[36] Sharpe, William F., *Macro-Investment Analysis*. An “electronic work-in-progress” published on the World Wide Web, 1995, at <http://www.stanford.edu/~wfsharpe/mia/mia.htm>.

[37] Sharpe, William F. and Gordon J. Alexander, *Investments*. Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990, ISBN 0-13-504382-4.

[38] Stigum, Marcia, with Franklin Robinson, *Money Market and Bond Calculations*. Richard D. Irwin., 1996, ISBN 1-55623-476-7.

Credit Risk Analysis

The credit rating and estimation transition probabilities come from:

[39] Altman, E., "Financial Ratios, Discriminant Analysis and the Prediction of Corporate Bankruptcy," *Journal of Finance*, Vol. 23, No. 4, (Sep., 1968), pp. 589-609.

[40] Basel Committee on Banking Supervision, *International Convergence of Capital Measurement and Capital Standards: A Revised Framework, Bank for International Settlements (BIS)*, comprehensive version, June 2006.

[41] Hanson, S. and T. Schuermann, "Confidence Intervals for Probabilities of Default," *Journal of Banking & Finance*, Elsevier, vol. 30(8), August 2006, pp. 2281-2301.

[42] Jafry, Y. and T. Schuermann, "Measurement, Estimation and Comparison of Credit Migration Matrices," *Journal of Banking & Finance*, Elsevier, vol. 28(11), November 2004, pp. 2603-2639.

[43] Löffler, G. and P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.

[44] Schuermann, T., "Credit Migration Matrices," in E. Melnick and B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment*, Wiley, 2008.

Portfolio Optimization

The Markowitz model is used for portfolio optimization computations.

[45] Kelley, J. E., "The Cutting-Plane Method for Solving Convex Programs," *Journal of the Society for Industrial and Applied Mathematics*, Vol. 8, No. 4, December 1960, pp. 703-712.

[46] Markowitz, H., "Portfolio Selection," *Journal of Finance*, Vol. 7, No. 1, March 1952, pp. 77-91.

[47] Markowitz, H. M., *Portfolio Selection: Efficient Diversification of Investments*, John Wiley & Sons, Inc., 1959.

[48] Rockafellar, R. T. and S. Uryasev, "Optimization of Conditional Value-at-Risk," *Journal of Risk*, Vol. 2, No. 3, Spring 2000, pp. 21-41.

[49] Rockafellar, R. T. and S. Uryasev, "Conditional Value-at-Risk for General Loss Distributions," *Journal of Banking and Finance*, Vol. 26, 2002, pp. 1443-1471.

[50] Konno, H. and H. Yamazaki, "Mean-Absolute Deviation Portfolio Optimization Model and Its Application to Tokyo Stock Market," *Management Science*, Vol. 37, No. 5, May 1991, pp. 519-531.

Stochastic Differential Equations

The SDE formulas come from:

[51] 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.

[52] Ait-Sahalia, Y., “Transition Densities for Interest Rate and Other Nonlinear Diffusions,” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

[53] Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

[54] Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

[55] Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

[56] Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

active return	Amount of return achieved in excess of the return produced by an appropriate benchmark (for example, an index portfolio).
active risk	Standard deviation of the active return. Also known as the tracking error .
American option	An option that can be exercised any time until its expiration date. Contrast with European option.
amortization	Reduction in value of an asset over some period for accounting purposes. Generally used with intangible assets. Depreciation is the term used with fixed or tangible assets.
annuity	A series of payments over a period of time. The payments are usually in equal amounts and usually at regular intervals such as quarterly, semiannually, or annually.
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.
arbitrage	The purchase of securities on one market for immediate resale on another market to profit from a price or currency discrepancy.
basis point	One hundredth of one percentage point, or 0.0001.
basis	Day count basis determines how interest accrues over time for various instruments and the amount transferred on interest payment dates. The calculation of accrued interest for dates between payments also uses day count basis. Day count basis is a fraction of $\text{Number of interest accrual days} / \text{Days in the relevant coupon period}$. Supported day count conventions and basis values are:

Basis Value	Day Count Convention
0	actual/actual (default) — Number of days in both a period and a year is the actual number of days.
1	30/360 SIA — Year fraction is calculated based on a 360 day year with 30-day months, after applying the following rules: If the first date and the second date are the last day of February, the second date is changed to the 30th. If the first date falls on the 31st or is the last day of February, it is changed to the 30th. If after the preceding test, the first day is the 30th and the second day is the 31st, then the second day is changed to the 30th.
2	actual/360 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360.
3	actual/365 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year).
4	30/360 PSA — Number of days in every month is set to 30 (including February). If the start date of the period is either the 31st of a month or the last day of February, the start date is set to the 30th, while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
5	30/360 ISDA — Number of days in every month is set to 30, except for February where it is the actual number of days. If the start date of the period is the 31st of a month, the start date is set to the 30th while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.

Basis Value	Day Count Convention
6	30E /360 — Number of days in every month is set to 30 except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360.
7	actual/365 Japanese — Number of days in a period is equal to the actual number of days, except for leap days (29th February) which are ignored. The number of days in a year is 365 (even in a leap year).
8	actual/actual ICMA — Number of days in both a period and a year is the actual number of days and the compounding frequency is annual.
9	actual/360 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360 and the compounding frequency is annual.
10	actual/365 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year) and the compounding frequency is annual.
11	30/360 ICMA — Number of days in every month is set to 30, except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360 and the compounding frequency is annual.
12	actual/365 ISDA — The day count fraction is calculated using the following formula: (Actual number of days in period that fall in a leap year / 366) + (Actual number of days in period that fall in a normal year / 365).

Basis Value	Day Count Convention
13	bus/252 — The number of days in a period is equal to the actual number of business days. The number of business days in a year is 252.

beta

The price volatility of a financial instrument relative to the price volatility of a market or index as a whole. Beta is commonly used with respect to equities. A high-beta instrument is riskier than a low-beta instrument.

binomial model

A method of pricing options or other equity derivatives in which the probability over time of each possible price follows a binomial distribution. The basic assumption is that prices can move to only two values (one higher and one lower) over any short time period.

Black-Scholes model

The first complete mathematical model for pricing options, developed by Fischer Black and Myron Scholes. It examines market price, strike price, volatility, time to expiration, and interest rates. It is limited to only certain kinds of options.

Bollinger band chart

A financial chart that plots actual asset data along with three other bands of data: the upper band is two standard deviations above a user-specified moving average; the lower band is two standard deviations below that moving average; and the middle band is the moving average itself.

bootstrapping, bootstrap method

An arithmetic method for backing an implied zero curve out of the par yield curve.

Brownian motion

A zero-mean continuous-time stochastic process with independent increments (also known as a *Wiener process*).

building a binomial tree

For a binomial option model: plotting the two possible short-term price-changes values, and then the subsequent two values each, and then the subsequent two values each, and so on over time, is known as “building a binomial tree.” See also **binomial model**.

call	<p>a. An option to buy a certain quantity of a stock or commodity for a specified price within a specified time. See also put. b. A demand to submit bonds to the issuer for redemption before the maturity date. c. A demand for payment of a debt. d. A demand for payment due on stock bought on margin.</p>
callable bond	<p>A bond that allows the issuer to buy back the bond at a predetermined price at specified future dates. The bond contains an embedded call option; that is, the holder has sold a call option to the issuer. See also puttable bond.</p>
candlestick chart	<p>A financial chart usually used to plot the high, low, open, and close price of a security over time. The body of the “candle” is the region between the open and close price of the security. Thin vertical lines extend up to the high and down to the low, respectively. If the open price is greater than the close price, the body is empty. If the close price is greater than the open price, the body is filled. See also high-low-close chart.</p>
cap	<p>Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain level.</p>
cash flow	<p>Cash received and paid over time.</p>
clean price	<p>The price of a bond excluding any interest that has accrued since issue or the most recent coupon payment.</p>
collar	<p>Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain upper level nor fall below a lower level. It is designed to protect an investor against wide fluctuations in interest rates.</p>
convexity	<p>A measure of the rate of change in duration; measured in time. The greater the rate of change, the more the duration changes as yield changes.</p>
correlation	<p>The simultaneous change in value of two random numeric variables.</p>

correlation coefficient	A statistic in which the covariance is scaled to a value between minus one (perfect negative correlation) and plus one (perfect positive correlation).
coupon	Detachable certificate attached to a bond that shows the amount of interest payable at regular intervals, usually semiannually. Originally coupons were actually attached to the bonds and had to be cut off or “clipped” to redeem them and receive the interest payment.
coupon dates	The dates when the coupons are paid. Typically a bond pays coupons annually or semiannually.
coupon rate	The nominal interest rate that the issuer promises to pay the buyer of a bond.
covariance	A measure of the degree to which returns on two assets move in tandem. A positive covariance means that asset returns move together; a negative covariance means they vary inversely.
credit rating	A credit rating evaluates a potential borrower’s ability to repay debt.
day count convention	A convention used to determine the number of days between two coupon dates, which is important in calculating accrued interest and present value when the next coupon payment is less than a full coupon period away. See also basis
delta	The rate of change of the price of a derivative security relative to the price of the underlying asset; that is, the first derivative of the curve that relates the price of the derivative to the price of the underlying security.
depreciation	Reduction in value of fixed or tangible assets over some period for accounting purposes. See also amortization .
derivative	A financial instrument that is based on some underlying asset. For example, an option is a derivative instrument based on the right to buy or sell an underlying instrument.

diffusion	The function that characterizes the random (stochastic) portion of a stochastic differential equation. <i>See also</i> stochastic differential equation .
dirty price	The price of a bond including the accrued interest.
discount curve	The curve of discount rates versus maturity dates for bonds.
discretization error	Errors that may arise due to discrete-time sampling of continuous stochastic processes.
drawdown	The peak to trough decline during a specific record period of an investment or fund.
drift	The function that characterizes the deterministic portion of a stochastic differential equation. <i>See also</i> stochastic differential equation .
duration	The expected life of a fixed-income security considering its coupon yield, interest payments, maturity, and call features. As market interest rates rise, the duration of a financial instrument decreases. <i>See also</i> Macaulay duration .
efficient frontier	A graph representing a set of portfolios that maximizes expected return at each level of portfolio risk. <i>See also</i> Markowitz model .
efficient portfolio	Portfolios satisfying the criteria of minimum risk for a given level of return and maximum return for a given level of risk. <i>See also</i> Markowitz model .
elasticity	<i>See</i> Lambda .
Euler approximation	A simulation technique that provides a discrete-time approximation of a continuous-time stochastic process.
European option	An option that can be exercised only on its expiration date. Contrast with American option.
ex-ante	Referring to future events, such as the future price of a stock.

ex-post	Referring to past events, when uncertainty of the result has been eliminated.
exercise price	The price set for buying an asset (call) or selling an asset (put). The strike price.
face value	The maturity value of a security. Also known as par value, principal value, or redemption value.
fixed-income security	A security that pays a specified cash flow over a specific period. Bonds are typical fixed-income securities.
floor	Interest-rate option that guarantees that the rate on a floating-rate loan will not fall below a certain level.
forward curve	The curve of forward interest rates versus maturity dates for bonds.
forward rate	The future interest rate of a bond inferred from the term structure, especially from the yield curve of zero-coupon bonds, calculated from the growth factor of an investment in a zero held until maturity.
future value	The value that a sum of money (the present value) earning compound interest will have in the future.
gamma	The rate of change of delta for a derivative security relative to the price of the underlying asset; that is, the second derivative of the option price relative to the security price.
greeks	Collectively, “greeks” refer to the financial measures beta, delta, gamma, lambda, rho, theta, and vega, which are sensitivity measures used in evaluating derivatives.
ISDA	International Swaps and Derivatives Association.
ISMA	International Securities Market Association.
hedge	A securities transaction that reduces or offsets the risk on an existing investment position.

high-low-close chart	A financial chart usually used to plot the high, low, open, and close price of a security over time. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right.
implied volatility	For an option, the variance that makes a call option price equal to the market price. Given the option price, strike price, and other factors, the Black-Scholes model computes implied volatility.
information ratio	The ratio of relative return to relative risk.
internal rate of return	a. The average annual yield earned by an investment during the period held. b. The effective rate of interest on a loan. c. The discount rate in discounted cash flow analysis. d. The rate that adjusts the value of future cash receipts earned by an investment so that interest earned equals the original cost. See also yield .
issue date	The date a security is first offered for sale. That date usually determines when interest payments, known as coupons, are made.
Ito process	Statistical assumptions about the behavior of security prices. For details, see the book by Hull in “Derivatives Pricing and Yields” on page A-4.
key rate duration	Key rate duration measures the sensitivity of a portfolio’s (or security’s) value in relation to changes in specific maturities of the zero or spot curve.
Lambda	The percentage change in the price of an option relative to a 1% change in the price of the underlying security. Also known as elasticity.
long position	Outright ownership of a security or financial instrument. The owner expects the price to rise in order to make a profit on some future sale.
long rate	The yield on a zero-coupon Treasury bond.

lower partial moment	A model for the moments of asset returns that fall below a minimum acceptable level of return.
Macaulay duration	A widely used measure of price sensitivity to yield changes developed by Frederick Macaulay in 1938. It is measured in years and is a weighted average-time-to-maturity of an instrument. The Macaulay duration of an income stream, such as a coupon bond, measures how long, on average, the owner waits before receiving a payment. It is the weighted average of the times payments are made, with the weights at time T equal to the present value of the money received at time T.
Markowitz model	A model for selecting an optimum investment portfolio, devised by H. M. Markowitz. It uses a discrete-time, continuous-outcome approach for modeling investment problems, often called the mean-variance paradigm. See also efficient portfolio and efficient frontier .
maturity date	The date when the issuer returns the final face value of a bond to the buyer.
mean	a. A number that typifies a set of numbers, such as a geometric mean or an arithmetic mean. b. The average value of a set of numbers.
modified duration	The Macaulay duration discounted by the per-period interest rate; that is, divided by $(1 + \text{rate}/\text{frequency})$.
Monte-Carlo simulation	A mathematical modeling process. For a model that has several parameters with statistical properties, pick a set of random values for the parameters and run a simulation. Then pick another set of values, and run it again. Run it many times (often 10,000 times) and build up a statistical distribution of outcomes of the simulation. This distribution of outcomes is then used to answer whatever question you are asking.
moving average	A price average that is adjusted by adding other parametrically determined prices over some time period.

moving-averages chart	A financial chart that plots leading and lagging moving averages for prices or values of an asset.
normal (bell-shaped) distribution	In statistics, a theoretical frequency distribution for a set of variable data, usually represented by a bell-shaped curve symmetrical about the mean.
odd first or last period	Fixed-income securities may be purchased on dates that do not coincide with coupon or payment dates. The length of the first and last periods may differ from the regular period between coupons, and thus the bond owner is not entitled to the full value of the coupon for that period. Instead, the coupon is prorated according to how long the bond is held during that period.
on-the-run treasury bonds	The most recently auctioned issue of a U.S. Treasury bond or note of a particular maturity.
option	A right to buy or sell specific securities or commodities at a stated price (exercise or strike price) within a specified time. An option is a type of derivative.
par value	The maturity or face value of a security or other financial instrument.
par yield curve	The yield curve of bonds selling at par, or face, value.
point and figure chart	A financial chart usually used to plot asset price data. Upward price movements are plotted as X's and downward price movements are plotted as O's.
present value	Today's value of an investment that yields some future value when invested to earn compounded interest at a known interest rate; that is, the future value at a known period in time discounted by the interest rate over that time period.
principal value	See par value .
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**.

PSA	Public Securities Association.
purchase price	Price actually paid for a security. Typically the purchase price of a bond is not the same as the redemption value.
put	An option to sell a stipulated amount of stock or securities within a specified time and at a fixed exercise price. See also call .
puttable bond	A bond that allows the holder to redeem the bond at a predetermined price at specified future dates. The bond contains an embedded put option; that is, the holder has bought a put option. See also callable bond .
Quant	A quantitative analyst; someone who does numerical analysis of financial information in order to detect relationships, disparities, or patterns that can lead to making money.
redemption value	See par value .
regression analysis	Statistical analysis techniques that quantify the relationship between two or more variables. The intent is quantitative prediction or forecasting, particularly using a small population to forecast the behavior of a large population.
rho	The rate of change in a derivative's price relative to the underlying security's risk-free interest rate.
return proxy	The proxy for return is a function that characterizes either the gross benefits or net benefits associated with portfolio choices.
risk proxy	The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices.

sensitivity	The “what if” relationship between variables; the degree to which changes in one variable cause changes in another variable. A specific synonym is volatility.
settlement date	The date when money first changes hands; that is, when a buyer actually pays for a security. It need not coincide with the issue date.
Sharpe ratio	The ratio of the excess return of an asset divided by the asset's standard deviation of returns.
short rate	The annualized one-period interest rate.
short sale, short position	The sale of a security or financial instrument not owned, in anticipation of a price decline and making a profit by purchasing the instrument later at a lower price, and then delivering the instrument to complete the sale. See also long position .
SIA	Securities Industry Association.
spot curve, spot yield curve	See zero curve, zero-coupon yield curve .
spot rate	The current interest rate appropriate for discounting a cash flow of some given maturity.
spread	For options, a combination of call or put options on the same stock with differing exercise prices or maturity dates.
standard deviation	A measure of the variation in a distribution, equal to the square root of the arithmetic mean of the squares of the deviations from the arithmetic mean; the square root of the variance.
stochastic	Involving or containing a random variable or variables; involving chance or probability.
stochastic differential equation	A generalization of an ordinary differential equation, with the addition of a noise process, that yields random variables as solutions.

straddle	A strategy used in trading options or futures. It involves simultaneously purchasing put and call options with the same exercise price and expiration date, and it is most profitable when the price of the underlying security is very volatile.
strata	See stratified sampling .
stratified sampling	A variance reduction technique that constrains a proportion of sample paths to specific subsets (or <i>strata</i>) of the sample space.
strike	Exercise a put or call option.
strike price	See exercise price .
swap	A contract between two parties to exchange cash flows in the future according to some formula.
swaption	A swap option; an option on an interest-rate swap. The option gives the holder the right to enter into a contracted interest-rate swap at a specified future date. See also swap .
term structure	The relationship between the yields on fixed-interest securities and their maturity dates. Expectation of changes in interest rates affects term structure, as do liquidity preferences and hedging pressure. A yield curve is one representation in the term structure.
theta	The rate of change in the price of a derivative security relative to time. Theta is usually very small or negative since the value of an option tends to drop as it approaches maturity.
tracking error	See active risk .
Treasury bill	Short-term U.S. government security issued at a discount from the face value and paying the face value at maturity.
Treasury bond	Long-term debt obligation of the U.S. government that makes coupon payments semiannually and is sold at or

	near par value in \$1000 denominations or higher. Face value is paid at maturity.
variance	The dispersion of a variable. The square of the standard deviation.
vega	The rate of change in the price of a derivative security relative to the volatility of the underlying security. When vega is large, the security is sensitive to small changes in volatility.
volatility	a. Another general term for sensitivity. b. The standard deviation of the annualized continuously compounded rate of return of an asset. c. A measure of uncertainty or risk.
Wiener process	See Brownian motion .
yield	a. Measure of return on an investment, stated as a percentage of price. Yield can be computed by dividing return by purchase price, current market value, or other measure of value. b. Income from a bond expressed as an annualized percentage rate. c. The nominal annual interest rate that gives a future value of the purchase price equal to the redemption value of the security. Any coupon payments determine part of that yield.
yield curve	Graph of yields (vertical axis) of a particular type of security versus the time to maturity (horizontal axis). This curve usually slopes upward, indicating that investors usually expect to receive a premium for securities that have a longer time to maturity. The benchmark yield curve is for U.S. Treasury securities with maturities ranging from three months to 30 years. See also term structure .
yield to maturity	A measure of the average rate of return that will be earned on a bond if held to maturity.
zero curve, zero-coupon yield curve	A yield curve for zero-coupon bonds; zero rates versus maturity dates. Since the maturity and duration (Macaulay duration) are identical for zeros, the zero curve is a pure depiction of supply/demand conditions

zero-coupon bond, or zero

for loanable funds across a continuum of durations and maturities. Also known as spot curve or spot yield curve.

A bond that, instead of carrying a coupon, is sold at a discount from its face value, pays no interest during its life, and pays the principal only at maturity.