Financial Toolbox

For Use with MATLAB®

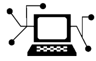
- Computation
- Visualization
- Programming

User's Guide

Version 3

The MathWorks

How to Contact The MathWorks:



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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Financial Toolbox User's Guide

© COPYRIGHT 1995 - 2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

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)

Contents

1 Getting Started

What Is the Financial Toolbox? **1-2**

Using Matrix Functions for Finance **1-4**
Key Definitions **1-4**
Referencing Matrix Elements **1-4**
Transposing Matrices **1-6**

Matrix Algebra Refresher **1-7**
Adding and Subtracting Matrices **1-7**
Multiplying Matrices **1-8**
Dividing Matrices **1-13**
Solving Simultaneous Linear Equations **1-13**
Operating Element-by-Element **1-16**

Function Input/Output Arguments **1-18**
Input Arguments **1-18**
Function Output Arguments **1-20**
Interest Rate Arguments **1-21**

2 Performing Common Financial Tasks

Handling and Converting Dates **2-4**
Date Formats **2-4**
Date Conversions **2-5**
Current Date and Time **2-8**
Determining Dates **2-9**

Formatting Currency **2-12**

Charting Financial Data **2-13**

High-Low-Close Chart Example **2-13**
Bollinger Chart Example **2-15**

Analyzing and Computing Cash Flows **2-17**
Interest Rates/Rates of Return **2-17**
Present or Future Values **2-18**
Depreciation **2-19**
Annuities **2-19**

Pricing and Computing Yields for Fixed-Income Securities **2-21**
Terminology **2-21**
SIA Framework **2-24**
SIA Default Parameter Values **2-25**
SIA Coupon Date Calculations **2-28**
SIA Semiannual Yield Conventions **2-28**
Pricing Functions **2-29**
Yield Functions **2-29**
Fixed-Income Sensitivities **2-30**
Term Structure of Interest Rates **2-31**

Pricing and Analyzing Equity Derivatives **2-34**
Sensitivity Measures **2-34**
Analysis Models **2-35**

3 Portfolio Analysis

Analyzing Portfolios **3-2**

Portfolio Optimization Functions **3-3**

Portfolio Construction Examples **3-5**
Efficient Frontier Example **3-5**

Portfolio Selection and Risk Aversion **3-8**
Optimal Risky Portfolio Example **3-9**

Constraint Specification **3-12**
Linear Constraint Equations **3-14**
Specifying Additional Constraints **3-17**

Active Returns and Tracking Error Efficient Frontier . . . **3-20**

4 Regression with Missing Data

Multivariate Normal Regression with Missing Data **4-2**
Multivariate Normal Regression **4-2**
Maximum Likelihood Estimation **4-3**
Special Case **4-4**
Least-Squares Regression **4-4**
Mean and Covariance Estimation **4-5**
Convergence **4-5**
Fisher Information **4-5**
Statistical Tests **4-6**

Maximum Likelihood Estimation with Missing Data **4-8**
ECM Algorithm **4-8**
Standard Errors **4-9**
Data Augmentation **4-9**
Multivariate Normal Regression Functions **4-10**
Multivariate Normal Regression without Missing Data **4-12**
Multivariate Normal Regression with Missing Data **4-12**
Least-Squares Regression with Missing Data **4-13**
Multivariate Normal Parameter Estimation
with Missing Data **4-14**
Support Functions **4-14**
Regressions **4-15**
Multivariate Normal Regression (MVNR) **4-15**
Least-Squares Regression (LSR) **4-16**
Covariance-Weighted Least Squares (CWLS) **4-17**
Feasible Generalized Least Squares (FGLS) **4-17**
Seemingly Unrelated Regression (SUR) **4-18**
Mean and Covariance Parameter Estimation **4-20**

Troubleshooting 4-20
Slow Convergence 4-21
Nonrandom Residuals 4-22
Nonconvergence 4-22
Examples 4-24
Portfolios with Missing Data 4-24

Valuation with Missing Data **4-32**
The Capital Asset Pricing Model 4-32
Estimation of the CAPM 4-33
Estimation with Missing Data 4-34
Separate Estimation of Some Technology Stock Betas 4-34
Grouped Estimation of Some Technology Stock Betas 4-37
References 4-40

5 Solving Sample Problems

Common Problems in Finance **5-3**
Sensitivity of Bond Prices to Changes in Interest Rates 5-3
Constructing a Bond Portfolio to Hedge Against Duration and Convexity 5-6
Sensitivity of Bond Prices to Parallel Shifts in the Yield Curve 5-8
Constructing Greek-Neutral Portfolios of European Stock Options 5-12
Term Structure Analysis and Interest Rate Swap Pricing . . . 5-15

Producing Graphics with the Toolbox **5-19**
Plotting an Efficient Frontier 5-19
Plotting Sensitivities of an Option 5-21
Plotting Sensitivities of a Portfolio of Options 5-23

6 Financial Time Series Analysis

Analyzing Financial Time Series **6-2**

Creating Financial Time Series Objects **6-3**
Using the Constructor **6-3**
Transforming a Text File **6-13**

Visualizing Financial Time Series Objects **6-17**
Using chartfts **6-17**
Zoom Tool **6-20**
Combine Axes Tool **6-23**

7 Using Financial Time Series

Introduction **7-2**

Working with Financial Time Series Objects **7-3**
Financial Time Series Object Structure **7-3**
Data Extraction **7-3**
Object to Matrix Conversion **7-5**
Indexing a Financial Time Series Object **7-7**
Operations **7-15**
Data Transformation and Frequency Conversion **7-19**

Demonstration Program **7-24**
Load the Data **7-24**
Create Financial Time Series Objects **7-25**
Create Closing Prices Adjustment Series **7-26**
Adjust Closing Prices and Make Them Spot Prices **7-26**
Create Return Series **7-27**
Regress Return Series Against Metric Data **7-27**
Plot the Results **7-28**
Calculate the Dividend Rate **7-29**

8 Financial Time Series Graphical User Interface

Introduction **8-2**
Main Window **8-2**

Using the Financial Time Series GUI **8-7**
Getting Started **8-7**
Data Menu **8-8**
Analysis Menu **8-13**
Graphs Menu **8-14**
Saving Time Series Data **8-18**

9 Technical Analysis

Introduction **9-2**

Examples **9-5**
Moving Average Convergence/Divergence (MACD) **9-5**
Williams %R **9-6**
Relative Strength Index (RSI) **9-8**
On-Balance Volume (OBV) **9-9**

10 Function Reference

Functions - By Category **10-2**
Handling and Converting Dates **10-3**
Formatting Currency and Price **10-6**
Charting Financial Data **10-6**
Analyzing and Computing Cash Flows **10-6**
Fixed-Income Securities **10-9**
Analyzing Portfolios **10-10**
Financial Statistics **10-12**

Pricing and Analyzing Derivatives **10-13**
GARCH Processes **10-14**
Financial Time Series Object and File Construction **10-14**
Financial Time Series Arithmetic Functions **10-15**
Financial Time Series Mathematical Functions **10-15**
Financial Time Series Utility Functions **10-16**
Financial Time Series Data Transformation Functions **10-17**
Financial Time Series Indicator Functions **10-17**
Financial Time Series Graphical User Interface Function . . **10-18**

Functions — Alphabetical List **10-19**

A Bibliography

Bond Pricing and Yields **A-2**
Term Structure of Interest Rates **A-2**
Derivatives Pricing and Yields **A-2**
Portfolio Analysis **A-3**
Financial Statistics **A-3**
Other References **A-3**

Glossary

Index

Getting Started

What Is the Financial Toolbox? (p. 1-2)	Overview of the product.
Using Matrix Functions for Finance (p. 1-4)	Elementary information about matrices.
Matrix Algebra Refresher (p. 1-7)	Matrix algebra you learned in school but may have forgotten.
Function Input/Output Arguments (p. 1-18)	Inputs and outputs for toolbox functions.

What Is the Financial Toolbox?

MATLAB® and the Financial Toolbox provide a complete integrated computing environment for financial analysis and engineering. The toolbox has everything you need to perform mathematical and statistical analysis of financial data and display the results with presentation-quality graphics. You can quickly ask, visualize, and answer complicated questions.

In traditional or spreadsheet programming you must deal with all sorts of housekeeping details: declaring, data typing, sizing, etc. MATLAB does all that for you. You just write expressions the way you think of problems. There is no need to switch tools, convert files, or rewrite applications.

With MATLAB and the Financial Toolbox, you can:

- Compute and analyze prices, yields, and sensitivities for derivatives and other securities, and for portfolios of securities.
- Perform Securities Industry Association (SIA) compatible fixed-income pricing, yield, and sensitivity analysis.
- Analyze or manage portfolios.
- Design and evaluate hedging strategies.
- Identify, measure, and control risk.
- Analyze and compute cash flows, including rates of return and depreciation streams.
- Analyze and predict economic activity.
- Visualize and analyze financial time series data
- Create structured financial instruments, including foreign-exchange instruments.
- Teach or conduct academic research.

This chapter uses MATLAB to review the fundamentals of matrix algebra you need for financial analysis and engineering applications. It contains these sections:

- "Using Matrix Functions for Finance" on page 1-4

  Reviews Key Definitions and some matrix algebra fundamentals, such as Referencing Matrix Elements and Transposing Matrices.

- "Matrix Algebra Refresher" on page 1-7

  Provides a brief refresher on using matrix functions in financial analysis and engineering
- "Function Input/Output Arguments" on page 1-18

  Describes acceptable formats for providing data to MATLAB and the resulting output from computations on the supplied data.

This material explains some MATLAB concepts and operations using financial examples to help get you started.

Using Matrix Functions for Finance

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 manipulates sets of numbers more quickly, easily, and naturally.

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 an “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.” Here is a 1-by-4 vector of cash flows in MATLAB notation.

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

Scalar. A 1-by-1 matrix; i.e., 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

Matrix algebra and matrix operations are fundamental to using MATLAB in financial analysis and engineering. The topics discussed in this section include:

- "Adding and Subtracting Matrices" on page 1-7
- "Multiplying Matrices" on page 1-8
- "Dividing Matrices" on page 1-13
- "Solving Simultaneous Linear Equations" on page 1-13
- "Operating Element-by-Element" on page 1-16

These explanations should help refresh your skills.

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/~wfsharpe/mia/mia.htm
```

Note 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" below, 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 =

        275.00        325.00
        700.00        600.00
        400.00        650.00
```

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 above.

Matrix multiplication also is *not* commutative; i.e., 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 and B is 3-by-1, A∗B yields a scalar (1-by-1) 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, simply multiply the vectors

```
PortfValue = ClosePrices * NumShares
```

which yields

```
PortfValue =

            35412.50
```

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 =

      4250.00       1500.00        7887.50
     21250.00       7500.00       39437.50
     12750.00       4500.00       23662.50
```

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

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

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

then the dot product

$$X \bullet Y = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

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 =

      35412.50

Value = sum(ClosePrices .* NumShares')

Value =

      35412.50
```

As expected, the value in these cases is exactly the same as 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 =

      35412.50
      35587.50
      35475.00
      35550.00
      35512.50
```

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

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

then $C = A*B$ is an m-by-p matrix; and the element c_{ij} in the ith row and jth 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 =

      35412.50     26331.25
      35587.50     26437.50
      35475.00     26325.00
      35550.00     26456.25
      35512.50     26493.75
```

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 =
        200.00        400.00
       1000.00        800.00
        600.00        300.00
```

Dividing Matrices

Matrix division is useful primarily for solving equations, and especially for solving simultaneous linear equations (see the next section). For example, you want to solve for X in $A*X = B$.

In ordinary algebra, you would simply 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 (`\` and `/`). In general,

`X = A\B` solves for `X` in `A*X = B`

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

In general, matrix `A` must be a nonsingular square matrix; i.e., 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, please consult a textbook on linear algebra such as the one by Hill listed in the "Bibliography.") 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 zero), 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 we usually want to solve at the same time; i.e., 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

$$2x + y = 13$$
$$x - 3y = -18$$

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

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

where $A*X = B$.

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

```
A = [2   1
      1  -3];

B = [13
    -18];

X = A \ B
```

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

```
X = [3
      7]
```

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

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

by representing them as matrices

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

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 =

       2400.00
       1600.00
```

Now the investor asks the 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 =

         43.75
         37.50
```

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 an array operation in MATLAB, 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
```

Function Input/Output Arguments

MATLAB 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 has the capability to 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.

Input Arguments

Matrix Input

Because MATLAB can process vectors and matrices easily, most functions in the Financial Toolbox 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 strings in MATLAB 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
```

Function 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

`amortize`	Amortization
`depfixdb`	Fixed declining-balance depreciation
`depgendb`	General declining-balance depreciation
`depsoyd`	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 `depfixdb` 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. `depfixdb` 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.

2

Performing Common Financial Tasks

Handling and Converting Dates (p. 2-4)	Date strings and serial date numbers. Date conversions. Holidays and cash-flow dates.
Formatting Currency (p. 2-12)	Decimal and fractional formats. Bank format.
Charting Financial Data (p. 2-13)	Useful functions for plotting financial data.
Analyzing and Computing Cash Flows (p. 2-17)	Rates of return. Present and future values. Depreciation.
Pricing and Computing Yields for Fixed-Income Securities (p. 2-21)	Securities Industry Association (SIA) conventions. Sensitivities. Term structure.
Pricing and Analyzing Equity Derivatives (p. 2-34)	Black-Scholes and binomial models.

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

- Handling and converting dates

  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.

- Formatting currency

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

- Charting financial data

  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

  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; analyzing the term structure of interest rates

  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

  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

  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. (The GARCH Toolbox provides a more comprehensive and integrated computing environment. For information see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod`.)

Handling and Converting Dates

Since virtually all financial data is dated or derives from a time series, financial functions must have extensive date-handling capabilities. This section discusses date handling in the Financial Toolbox, specifically the topics:

- "Date Formats" on page 2-4
- "Date Conversions" on page 2-5
- "Current Date and Time" on page 2-8
- "Determining Dates" on page 2-9

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.

Date Formats

You most often work with date strings (14-Sep-1999) when dealing with dates. The Financial Toolbox works internally with *serial date numbers* (e.g., 730377). A serial date number represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, 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 pm on 14-Sep-1999, in MATLAB, is date number `730377.75`.

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 toolbox provides functions that convert date strings to serial date numbers, and vice versa.

Date Conversions

Functions that convert between date formats are

`datedisp`	Displays a numeric matrix with date entries formatted as date strings
`datenum`	Converts a date string to a serial date number
`datestr`	Converts a serial date number to a date string
`m2xdate`	Converts MATLAB serial date number to Excel serial date number
`x2mdate`	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 the Financial Toolbox 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 to 31.
- The month field is either an integer from 1 to 12 or an alphabetic 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 1-19.

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, e.g., 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

Format	Description
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 toolbox 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 toolbox provides the `holidays` function, which contains holidays and special nontrading days for the New York Stock Exchange between 1950 and 2030, inclusive. You can edit the `holidays.m` file to customize it with your own holidays and nontrading days. In this example, use it 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
```

Formatting Currency

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

`cur2frac`	Converts decimal currency values to fractional values
`cur2str`	Converts a value to Financial Toolbox bank format
`frac2cur`	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; e.g., `($123.45)` and `$6789.01`. The standard MATLAB bank format uses two decimal places, no dollar sign, and a minus sign for negative numbers; e.g., `-123.45` and `6789.01`.

Charting Financial Data

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

`bolling`	Bollinger band chart
`bollinger`	Time series Bollinger band
`candle`	Candlestick chart
`candle`	Time series candle plot
`pointfig`	Point and figure chart
`highlow`	High, low, open, close chart
`highlow`	Time series High-Low plot
`movavg`	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. For lists of these, see "Charting Financial Data" on page 10-6 and "Financial Time Series Indicator Functions" on page 10-17.

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);
```

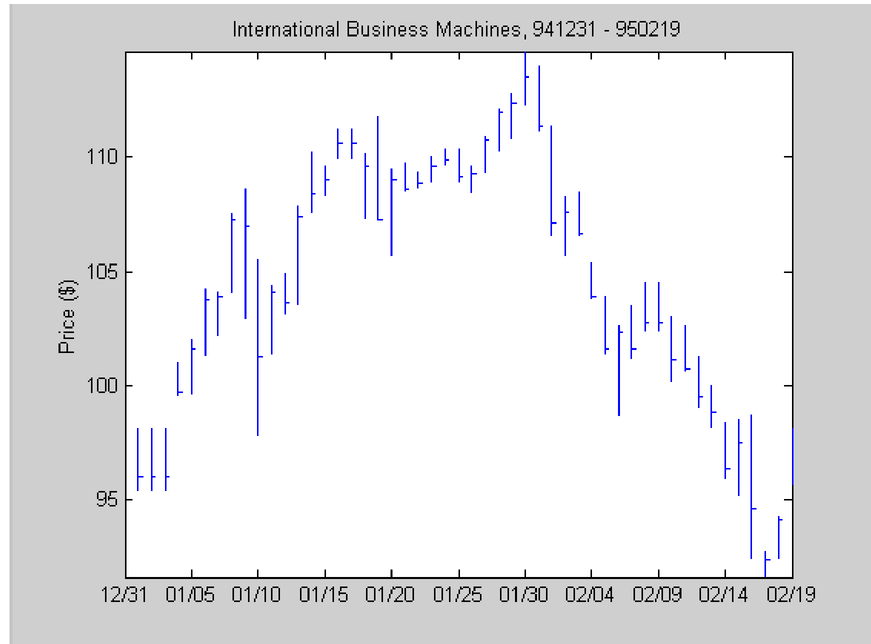
Open a figure window for the chart. Use the Financial Toolbox `highlow` function to plot high, low, and close prices for the last 50 trading days in the data file.

```
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 similar to this. The plotted data and axes you see may differ. Viewed online, the high-low-close bars are blue.



Bollinger Chart Example

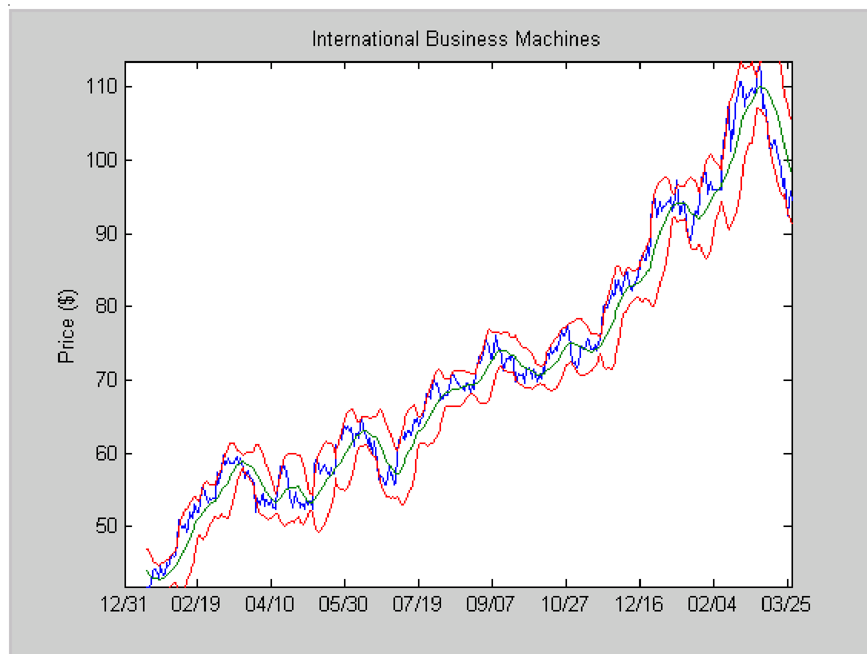
Next the Financial Toolbox `bolling` function produces a Bollinger band chart using all the closing prices in the same IBM stock price matrix. A Bollinger band chart plots actual data along with three other bands of data. The upper band is two standard deviations above a moving average; the lower band 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, simply execute the Financial Toolbox 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 "Creating Plots" in the MATLAB documentation. See the MATLAB documentation for details on the `axis`, `title`, `xlabel`, and `ylabel` functions.

Analyzing and Computing Cash Flows

The Financial Toolbox cash-flow functions compute interest rates, 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, simply 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

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

where *Investment* is a (negative) initial cash outlay at time 0, cf_n is the cash flow in the nth period, and n is the number of periods. Basically, `irr` finds the rate r such that the net present value of the cash flow equals the initial investment. If all of the cf_ns 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. There is usually only one answer that makes sense, but it is possible to get returns of both 5% and 11% (for example) from one income stream.

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, simply 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 = depgendb(15000, 1500, 5, 1.5)
Decline2 = depgendb(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 approximately 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

The Securities Industry Association (SIA) has established conventions regarding bond pricing, yield calculation and quotation, time factors and accrued interest, coupon and quasi-coupon dates, and duration and convexity sensitivity measures. The Financial Toolbox includes SIA-compliant functions to compute accrued interest, determine prices and yields, as well as calculate convexity and duration of fixed-income securities. It also includes a set of functions to generate and analyze term structure of interest rates.

SIA-compliant functions can be used with U.S. Treasury bills, bonds, and notes; corporate bonds; and municipal bonds. Bonds can have long, normal or short first or last coupon periods.

The "Function Reference" identifies SIA-compliant functions. These functions have been thoroughly tested against the benchmarks found in Jan Mayle's *Standard Securities Calculation Methods* document listed in the "Bibliography."

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; i.e., 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.

Table 2-1: 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 or not the given year is a leap year).

Table 2-2: 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 PSA (Public Securities 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.

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*.

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.

Table 2-3: End-of-Month Rule

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

SIA Framework

Many of the fixed-income related functions in the Financial Toolbox comply with the Securities Industry Association (SIA) conventions. Although not all SIA-compliant functions require the same input arguments, they all accept the following common set of input arguments.

Table 2-4: SIA Common Input Arguments

Input	Meaning
`Settle`	Settlement date
`Maturity`	Maturity date
`Period`	Coupon payment period
`Basis`	Day-count basis

Table 2-4: SIA Common Input Arguments

Input	Meaning
`EndMonthRule`	End-of-month payment rule
`IssueDate`	Bond issue date
`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 simply a standard bond with a coupon payment structure based solely on the maturity date.

SIA Default Parameter Values

To illustrate the use of default values in SIA-compliant 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 (e.g., zero divided by zero). However, NaN is also a very convenient placeholder. In the SIA functions of the Financial Toolbox, NaN indicates the presence of a nonapplicable value. It tells the SIA fixed-income 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 prior to 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 SIA-compliant 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 (i.e., 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.

SIA Coupon Date Calculations

Calculating coupon dates, either actual or quasi dates, is notoriously complicated. The Financial Toolbox 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, and finally (3) the maturity date.

In other words, an SIA-compliant function in the Financial Toolbox 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.

SIA Semiannual Yield Conventions

Within the SIA framework, all yields and time factors for price-to-yield conversion are quoted on a semiannual bond basis (see `bndprice`, `bndyield`, and `cfamounts`) regardless of the period of the bond's coupon payments (including zero-coupon bonds). In addition, any yield-related sensitivity (i.e., duration and convexity), when quoted on a periodic basis, assumes semiannual coupon periods. (See `bndconvp`, `bndconvy`, `bnddurp`, and `bnddury`).

Pricing Functions

This example shows how easily you can compute the price of a bond with an odd first period using the SIA-compliant 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, as well as prices of Treasury bills and discounted securities such as zero-coupon bonds.

Note `bndprice` and other SIA-compliant functions use nonlinear formulas to compute the price of a security. For this reason, the Financial Toolbox 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

The toolbox includes SIA-compliant 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; i.e., divided by (1+rate/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 (i.e., 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`).

Term Structure of Interest Rates

The toolbox 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. 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 `tbl2bond` function to convert the Treasury bill data to Treasury bond format.

```
TBTBond = tbl2bond(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 toolbox 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);
```

With this market data, you are now ready to use one of the 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 toolbox 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 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

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; i.e., 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 the book by John Hull listed in the *Bibliography*.

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, etc. 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.

3

Portfolio Analysis

Analyzing Portfolios (p. 3-2)	Managing risk and return
Portfolio Optimization Functions (p. 3-3)	Tables of functions for portfolio optimization
Portfolio Construction Examples (p. 3-5)	Constructing portfolios on the efficient frontier
Portfolio Selection and Risk Aversion (p. 3-8)	Controlling portfolio risk
Constraint Specification (p. 3-12)	Managing portfolio constraints
Active Returns and Tracking Error Efficient Frontier (p. 3-20)	Minimize the variance of the difference in returns with respect to a given target portfolio

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

The Financial Toolbox 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	
`portalloc`	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	
`frontcon`	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.
`frontier`	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.
`portopt`	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		
portcons	Generates the portfolio constraints matrix for a portfolio of asset investments using linear inequalities. The inequalities are of the type `A*Wts' <= b`, where `Wts` is a row vector of weights. The capabilities of `portcons` are also provided individually by the following functions.	
Constraint Specification (continued)		
	pcalims	Asset minimum and maximum allocation. Generates a constraint set to fix the minimum and maximum weight for each individual asset.
	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	
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.

Portfolio Construction Examples

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

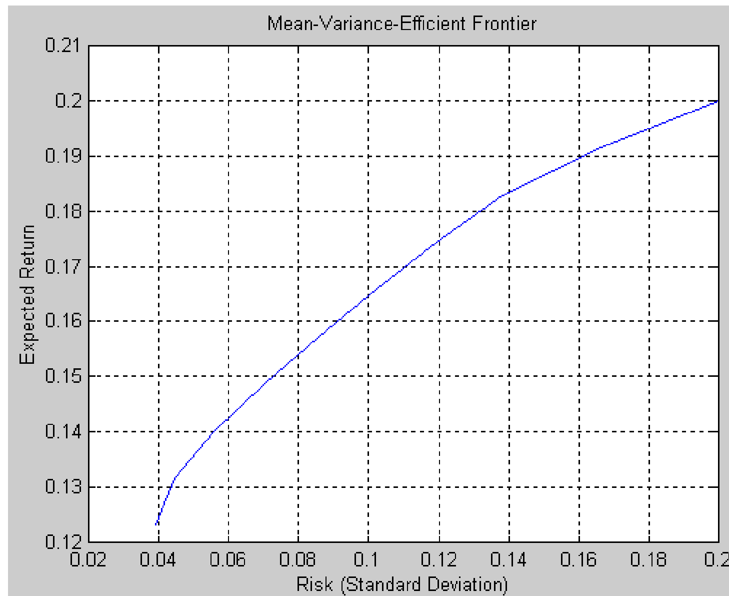
This example computes the efficient frontier of portfolios consisting of three different assets using the function `frontcon`. To visualize the efficient frontier curve clearly, consider 10 different evenly spaced portfolios.

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

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 the Financial Toolbox is

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

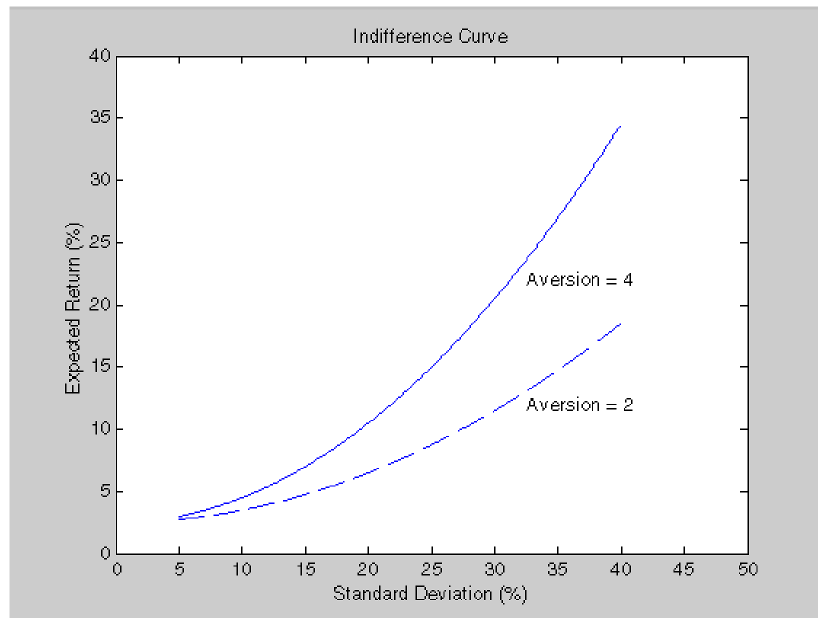
where:

`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 Example

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);
```

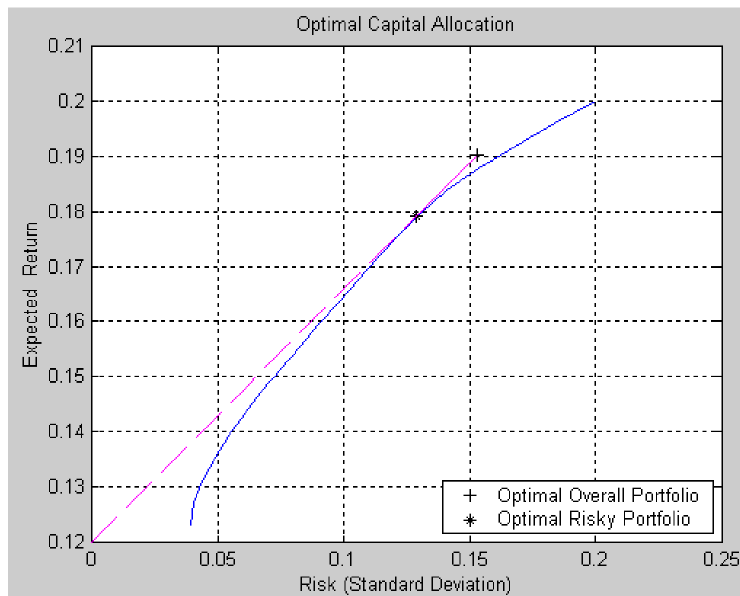
As with `frontcon`, calling `portopt` while specifying output arguments returns the corresponding vectors and arrays representing the risk, return, and weights for each of the portfolios along the efficient frontier. Use them as the first three input arguments to the function `portalloc`.

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

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 table below 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' <= 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.

Table 3-1: 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 Table 3-2.

Table 3-2: 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*Wts <= 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 = [-1    -1     0;
      1     1     0;
      0     0    -1;
      0     0     1;
     -1     0     0;
      1     0     0;
      0    -1    -1;
      0     1     1]

b = [-0.30;
      0.75;
     -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 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 Table 3-2, 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]
```

Table 3-1, 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 wish 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.

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

Next, assume that the target index portfolio is simply 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 via `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 <= 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 <= 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
         0    0.2000
         0    0.2000
```

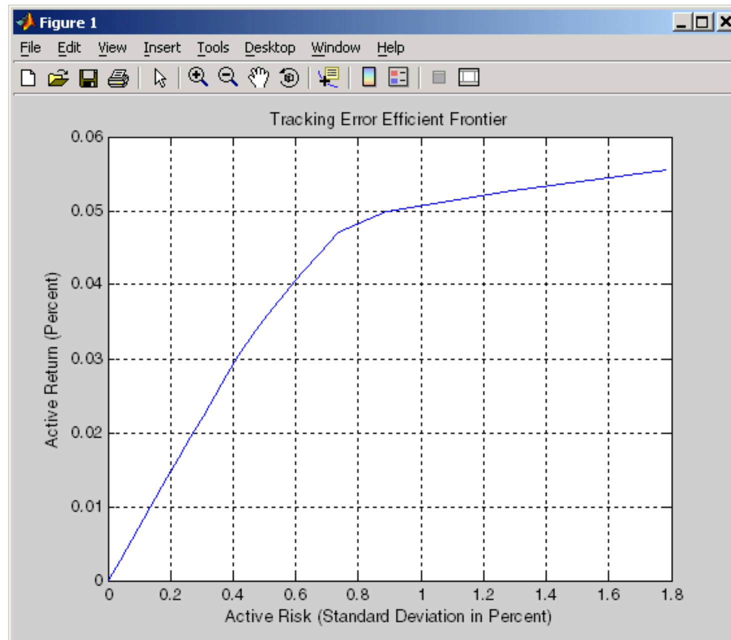
```
0      0.2000
0      0.2000
0      0.2000
```

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:

$$b_{active} = b_{absolute} - A \cdot Index$$

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-hand portfolio along the frontier. This zero-risk/zero-return portfolio has a very 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);
```

4

Regression with Missing Data

Multivariate Normal Regression with Missing Data (p. 4-2)	Introduces the multivariate normal linear regression model
Maximum Likelihood Estimation with Missing Data (p. 4-8)	Estimating the parameters of the model via maximum likelihood estimation
Multivariate Normal Regression Functions (p. 4-10)	Describes several multivariate normal regressions functions, with and without missing data
Regressions (p. 4-15)	Describes several regression methods used in functions that handle missing data
Troubleshooting (p. 4-20)	Handling difficulties that may arise with these methods
Examples (p. 4-24)	Examples of portfolio optimization and CAPM estimation with missing data

Multivariate Normal Regression with Missing Data

This section focuses on the use of likelihood-based methods for multivariate normal regression with missing data. 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 in Sexton and Swensen [9].

Multivariate Normal Regression

Suppose you have a multivariate normal linear regression model in the form

$$\begin{bmatrix} \mathbf{Z}_1 \\ \vdots \\ \mathbf{Z}_m \end{bmatrix} \sim N\left(\begin{bmatrix} \mathbf{H}_1\mathbf{b} \\ \vdots \\ \mathbf{H}_m\mathbf{b} \end{bmatrix}, \begin{bmatrix} \mathbf{C} & \cdots & \mathbf{0} \\ \vdots & & \vdots \\ \mathbf{0} & \cdots & \mathbf{C} \end{bmatrix} \right)$$

where it is assumed that the model has m observations of n-dimensional random variables $\mathbf{Z}_1, \ldots, \mathbf{Z}_m$ 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 $\mathbf{H}_1, \ldots, \mathbf{H}_m$, 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 $k = 1, \ldots, m$.

A concise way to write this model is

$$\mathbf{Z}_k \sim N(\mathbf{H}_k\,\mathbf{b}, \mathbf{C})$$

for $k = 1, \ldots, m$.

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 parameters to be estimated 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 that 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 via 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

$$L(\mathbf{z}_1, \dots, \mathbf{z}_m; \mathbf{b}, \mathbf{C}) = \frac{1}{2} mn \log(2\pi) + \frac{1}{2} m \log(\det(\mathbf{C})) + \frac{1}{2} \sum_{k=1}^{m} (\mathbf{z}_k - \mathbf{H}_k\mathbf{b})^T \mathbf{C}^{-1} (\mathbf{z}_k - \mathbf{H}_k\mathbf{b})$$

Although the cross-sectional residuals must be independent, this log-likelihood function can be used 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.

With the exception of an important special case, 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 must be solved by 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 $\mathbf{C}$ can be obtained by the two-stage iterative process with

$$\mathbf{b}^{(t+1)} = \left(\sum_{k=1}^{m} \mathbf{H}_k{}^T (\mathbf{C}^{(t)})^{-1} \mathbf{H}_k \right)^{-1} \left(\sum_{k=1}^{m} \mathbf{H}_k{}^T (\mathbf{C}^{(t)})^{-1} \mathbf{z}_k \right)$$

and

$$\mathbf{C}^{(t+1)} = \frac{1}{m} \sum_{k=1}^{m} (\mathbf{z}_k - \mathbf{H}_k \mathbf{b}^{(t+1)})(\mathbf{z}_k - \mathbf{H}_k \mathbf{b}^{(t+1)})^T$$

for $t = 0, 1, \ldots$.

Special Case

The special case mentioned above 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 **C** is a 1-by-1 matrix that drops out of the maximum likelihood iterates so that a single-step estimate for **b** and **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 **b** and **C**, where

$$\mathbf{b}^{LS} = \left(\sum_{k=1}^{m} \mathbf{H}_k{}^T \mathbf{H}_k \right)^{-1} \left(\sum_{k=1}^{m} \mathbf{H}_k{}^T \mathbf{z}_k \right)$$

and

$$\mathbf{C}^{LS} = \frac{1}{m} \sum_{k=1}^{m} (\mathbf{z}_k - \mathbf{H}_k \mathbf{b}^{LS})(\mathbf{z}_k - \mathbf{H}_k \mathbf{b}^{LS})^T$$

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 the Financial Toolbox, 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, and 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

$$\nabla^2 L(\mathbf{z}_1, \dots, \mathbf{z}_m; \theta)$$

and the Fisher information matrix is

$$\mathbf{I}(\theta) = -E[\nabla^2 L(\mathbf{z}_1, \dots, \mathbf{z}_m; \theta)]$$

where the partial derivatives of the ∇^2 operator are taken with respect to the combined parameter vector θ that contains the distinct components of **b** and **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

$$(\theta^{(t)} - \theta) \sim N(0, \mathbf{I}^{-1}(\theta^{(t)})) \text{ as } t \to \infty$$

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

$$s^2(\hat{\theta}_i) = (\mathbf{I}^{-1}(\hat{\theta}))_{ii}$$

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

$$\hat{\theta}_i \pm 1.96\, s(\hat{\theta}_i)$$

for $i = 1, \dots, q$.

Error ellipses at a level-of-significance $\alpha \in [0, 1]$ for the parameter estimates satisfy the inequality

$$(\theta - \hat{\theta})^T \mathbf{I}(\hat{\theta})(\theta - \hat{\theta}) \le \chi^2_{1-\alpha,q}$$

and follow a χ^2 distribution with q degrees-of-freedom. Note that similar inequalities can be formed for any subcollection of the parameters.

More generally, 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

Suppose that a portion of the sample data is missing, where missing values are represented as `NaN`s. 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 employ 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 the Financial Toolbox is the ECM algorithm of Meng and Rubin [8] with enhancements by Sexton and Swensen [9], where ECM stands for Expectation Conditional Maximization.

Each sample $\mathbf{z}_k$ for $k = 1, \ldots, m$, 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 prior to an IPO. For a counterexample, censored data, in which all values greater than some cutoff are replaced with `NaN`s, 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. We can define a permutation matrix $\mathbf{P}_\mathrm{k}$ so that

$$\mathbf{z}_k = \mathbf{P}_k \begin{bmatrix} \mathbf{x}_k \\ \mathbf{y}_k \end{bmatrix}$$

for $k = 1, \ldots, m$.

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

$$E [\, \mathbf{X}_k \mid \mathbf{Y}_k = \mathbf{y}_k ; \mathbf{b}^{(t)}, \mathbf{C}^{(t)} \,]$$

$$cov[\, \mathbf{X}_k \mid \mathbf{Y}_k = \mathbf{y}_k; \mathbf{b}^{(t)}, \mathbf{C}^{(t)} \,]$$

for each sample $k \in \{1, \ldots, m\}$ 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 need to “fill in” missing values as they estimate model parameters. In some cases you may want to fill in the missing values. Although it might seem that 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 $\hat{\mathbf{b}}$ and $\hat{\mathbf{C}}$, each observation has moments

$$E[\, \mathbf{Z}_k \,] = \mathbf{H}_k \hat{\mathbf{b}}$$

and

$$cov(\mathbf{Z}_k) = \mathbf{H}_k^T \hat{\mathbf{C}} \mathbf{H}_k^T$$

for $k = 1, \ldots, m$, where we 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 $E[\mathbf{Z}_k]$ and $cov(\mathbf{Z}_k)$ based on the parameter estimates, you can create conditional estimates

$$E[\mathbf{X}_k \mid \mathbf{y}_k]$$

and

$$cov(\mathbf{X}_k \mid \mathbf{y}_k)$$

using standard multivariate normal distribution theory. Given these conditional estimates, you can simulate random samples for the missing values from the conditional distribution

$$\mathbf{X}_k \sim N(E[\mathbf{X}_k \mid \mathbf{y}_k], cov(\mathbf{X}_k \mid \mathbf{y}_k))$$

The samples from this distribution reflect the pattern of missing and nonmissing values for observations $k = 1, \ldots, m$. 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

The Financial Toolbox has a number of functions for multivariate normal regression with or without missing data. In addition, the functions can be used for specialized regressions that are described in greater detail below.

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 4-12
- "Multivariate Normal Regression with Missing Data" on page 4-12
- "Least-Squares Regression with Missing Data" on page 4-13
- "Multivariate Normal Parameter Estimation with Missing Data" on page 4-14

Additional support functions are also provided. See "Support Functions" on page 4-14.

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

`Data(k, :)` = $\mathbf{z}_k^T$

for `k = 1, ..., 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

`Design{k}` = $\mathbf{H}_k$

for `k = 1, ... , NumSamples`. If `Design` has a single cell, it is assumed to be the same `Design` matrix for each sample such that

`Design{1}` = $\mathbf{H}_1 = \ldots = \mathbf{H}_m$

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, 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

The following functions can be used for multivariate normal regression without missing data.

`mvnrmle`	Estimate model parameters, residuals, and the residual covariance
`mvnrstd`	Estimate standard errors of model and covariance parameters
`mvnrfish`	Estimate the Fisher information matrix
`mvnrobj`	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

The following functions can be used for multivariate normal regression with missing data.

`ecmmvnrmle`	Estimate model parameters, residuals, and the residual covariance
`ecmmvnrstd`	Estimate standard errors of model and covariance parameters
`ecmmvnrfish`	Estimate the Fisher information matrix
`ecmmvnrobj`	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

The following functions can be used for least-squares regression with missing data and can also be used for covariance-weighted least-squares regression with a fixed covariance matrix.

`ecmlsrmle`	Estimate model parameters, residuals, and the residual covariance
`ecmlsrobj`	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.

`ecmmvnrstd`	Estimate standard errors of model and covariance parameters
`ecmmvnrfish`	Estimate the Fisher information matrix

Multivariate Normal Parameter Estimation with Missing Data

The following functions can be used to estimate the mean and covariance of multivariate normal data.

`ecmnmle`	Estimate the mean and covariance of the data
`ecmnstd`	Estimate standard errors of the mean and covariance of the data
`ecmnfish`	Estimate the Fisher information matrix
`ecmnhess`	Estimate the Fisher information matrix using the Hessian
`ecmnobj`	Calculate the log-likelihood function

These functions behave slightly differently from the more general regression functions since they solve a very specialized problem. Consult the function reference pages for details.

Support Functions

Two useful support functions are included.

`convert2sur`	Convert a multivariate normal regression model into an SUR model.
`ecmninit`	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 `ecmninit` 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.)

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. For details see

- "Multivariate Normal Regression (MVNR)" on page 4-15
- "Least-Squares Regression (LSR)" on page 4-16
- "Covariance-Weighted Least Squares (CWLS)" on page 4-17
- "Feasible Generalized Least Squares (FGLS)" on page 4-17
- "Seemingly Unrelated Regression (SUR)" on page 4-18
- "Mean and Covariance Parameter Estimation" on page 4-20

Multivariate Normal Regression (MVNR)

Multivariate normal regression is the "standard" implementation of the regression functions in the Financial Toolbox.

MVNR without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

MVNR with Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmmvnrmle(Data, Design);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Least-Squares Regression (LSR)

Least-squares regression, 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 = `NumSamples` observations, the typical least-squares regression model seeks to minimize the objective function

$$\sum_{k=1}^{m} (\mathbf{Z}_k - \mathbf{H}_k \mathbf{b})^T (\mathbf{Z}_k - \mathbf{H}_k \mathbf{b})$$

which, within the maximum likelihood framework of the multivariate normal regression routine `mvnrmle`, 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 `ecmlsrmle` to do least-squares instead of multivariate normal regression.

LSR without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design, 1);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

LSR with Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmlsrmle(Data, Design);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Covariance-Weighted Least Squares (CWLS)

Given m = NUMSAMPLES observations, the typical covariance-weighted least-squares regression model seeks to minimize the objective function

$$\sum_{k=1}^{m} (\mathbf{Z}_k - \mathbf{H}_k \mathbf{b})^T \mathbf{C}_0 (\mathbf{Z}_k - \mathbf{H}_k \mathbf{b})$$

with fixed covariance $\mathbf{C}_0$.

In most cases, $\mathbf{C}_0$ is a diagonal matrix. The inverse matrix $\mathbf{W} = \mathbf{C}_0^{-1}$ 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.

CWLS without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design, 1, [], [], [],
                                              Covar0);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);S
```

CWLS with Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmlsrmle(Data, Design, [], [], [], [],
                                                Covar0);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Feasible Generalized Least Squares (FGLS)

A somewhat ad hoc form of least squares that has surprisingly good properties for misspecified or non-normal 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.

FGLS without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design, 2, 0, 0);
```

or (to illustrate the FGLS process explicitly)

```
[Parameters, Covar0] = mvnrmle(Data, Design, 1);
[Parameters, Covariance] = mvnrmle(Data, Design, 1, [], [], [],
                                   Covar0);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

FGLS with Missing Data

Estimate Parameters

```
[Parameters, Covar0] = ecmlsrmle(Data, Design);
[Parameters, Covariance] = ecmlsrmle(Data, Design, [], [], [], [],
                                     Covar0);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Seemingly Unrelated Regression (SUR)

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, i.e., NUMSERIES = 1, then the SUR design array will be 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');
```

SUR without Missing Data

Form an SUR Design

```
DesignSUR = convert2sur(Design, Group);
```

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, DesignSUR);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, DesignSUR, Covariance);
```

SUR with Missing Data

Form an SUR Design

```
DesignSUR = convert2sur(Design, Group);
```

Estimate Parameters

```
[Parameters, Covariance] = ecmmvnrmle(Data, DesignSUR);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, DesignSUR, Covariance);
```

Mean and Covariance Parameter Estimation

Without missing data you can, of course, estimate the mean of your `Data` with the function `mean` and the covariance with the function `cov`. Nevertheless, the function `ecmnmle` will do this for you if it detects an absence of missing values. Otherwise, it will use the ECM algorithm to handle missing values.

Estimate Parameters

```
[Mean, Covariance] = ecmnmle(Data);
```

Estimate Standard Errors

```
StdMean = ecmnstd(Data, Mean, Covariance);
```

Troubleshooting

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

$$\mathrm{Count}/(\mathrm{Count}-1)$$

where `Count` is the actual number of samples used in the estimation with $\mathrm{Count} \le \mathrm{NumSamples}$. 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 of thumb, the multivariate normal regression functions require

$$\mathrm{Count}*\mathrm{NumSeries} \le \max\{\mathrm{NumParams}, \mathrm{NumSeries}*(\mathrm{NumSeries}+1)/2\}$$

and the least-squares regression functions require

$$\mathrm{Count}*\mathrm{NumSeries} \le \mathrm{NumParams}$$

where `Count` is the actual number of samples used in the estimation with

$$\mathrm{Count} \le \mathrm{NumSamples}.$$

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 below in greater detail.

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 trick 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.

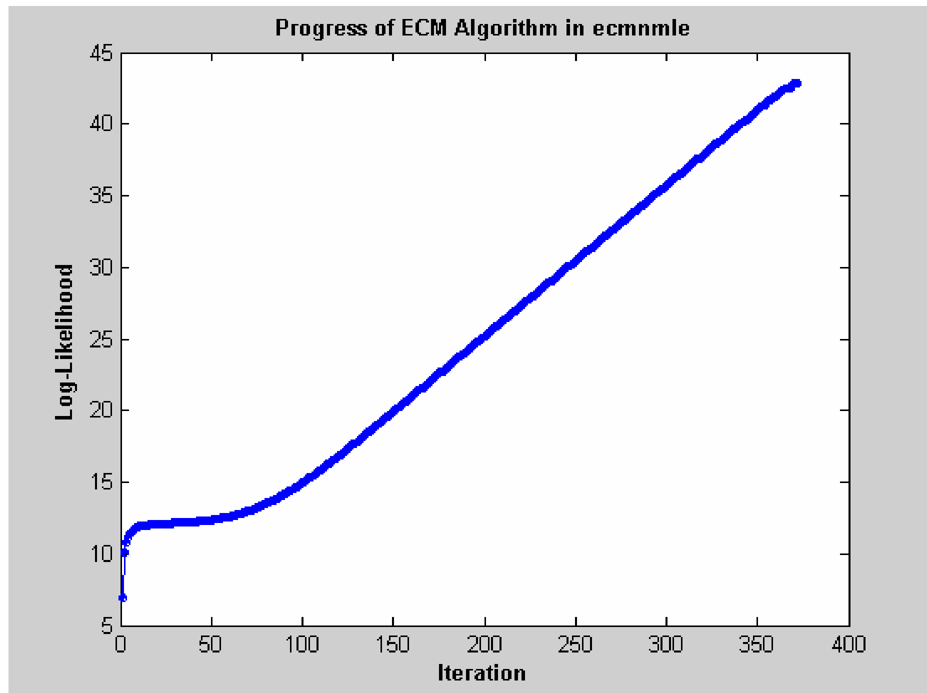
Other ad hoc approaches are possible although most approaches are problem-dependent. In particular, for mean and covariance estimation, the estimation function `ecmnmle` uses a function `ecmninit` to obtain an initial estimate.

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 may have exact results, the least-squares routine `ecmlsrmle` will still work although it will provide 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 very nearly singular covariance matrix and can be spotted if the algorithm fails to converge after about 100 iterations. If `MaxIterations` is increased to, say, 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 somewhat linearly to infinity as the likelihood function goes to zero. In this case, the function will converge but the covariance matrix is effectively singular with a smallest eigenvalue on the order of machine precision (`eps`).

For the function `ecmnmle`, a hard error looks like this

```
> In ecmninit at 60
  In ecmnmle at 140
??? Error using ==> ecmnmle
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, especially since 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/eps` should be considered suspect.

If either type of failure occurs, however, note that the regression routine is really telling you that something is probably wrong with the data. (Even with no missing data, two time series that are proportional to one another will produce a singular covariance matrix.)

Examples

This section has two examples that illustrate ways to use the missing data algorithms for portfolio optimization and for valuation.

Portfolios with Missing Data

This example works with five years of daily total return data for 12 computer technology stocks, with six hardware and six 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 the M-file `ecmtechdemo.m`.

To begin, load the data file

```
load ecmtechdemo
```

This file contains three quantities.

- `Assets` is a cell array of the tickers for the twelve stocks in the example.
- `Data` is a 1254 × 12 matrix of 1254 daily total returns for each of the 12 stocks
- `Dates` is a 1254 × 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 `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

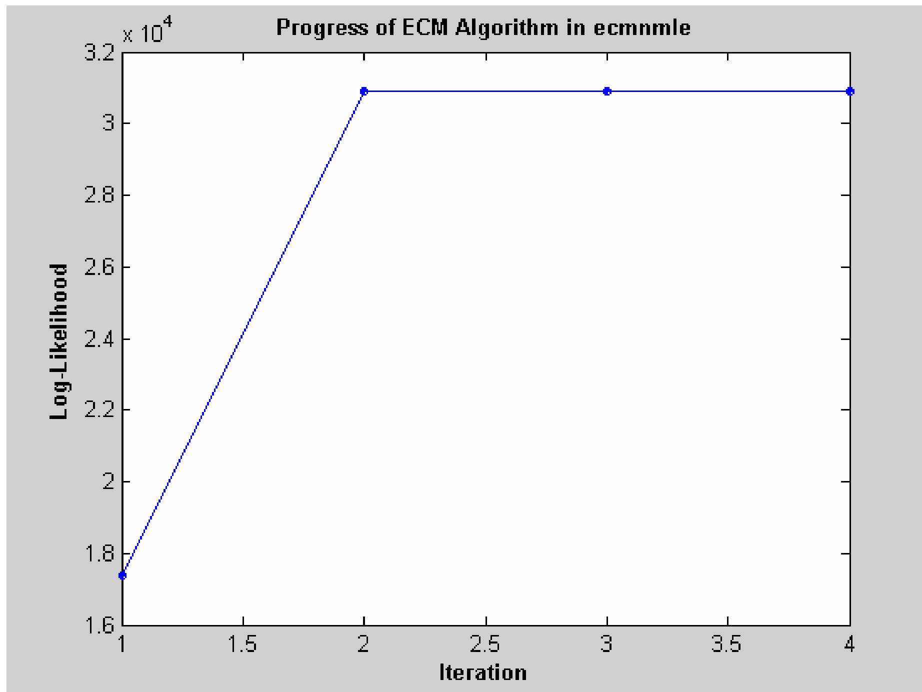
A naïve approach to the estimation of the mean and covariance for these 12 assets is to eliminate all days that have missing values for any of the 12 assets. Use the function `ecmninit` with the `nanskip` option to accomplish this.

```
[NaNMean, NaNCovar] = ecmninit(Data,'nanskip');
```

Contrast the result of this approach with using all available data and the function `ecmnmle` to compute the mean and covariance. First, call `ecmnmle` with no output arguments to establish that sufficient data is available to obtain meaningful estimates.

```
ecmnmle(Data);
```

The figure shows that, even with almost 87% of the Google data being `NaN` values, the algorithm converges after only four iterations.



Now estimate the mean and covariance as computed by `ecmnmle`.

```
>> [ECMMean, ECMCovar] = ecmnmle(Data)

ECMMean =
```

```
    0.0008
    0.0008
   -0.0005
    0.0002
    0.0011
    0.0038
   -0.0003
   -0.0000
   -0.0003
   -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
```

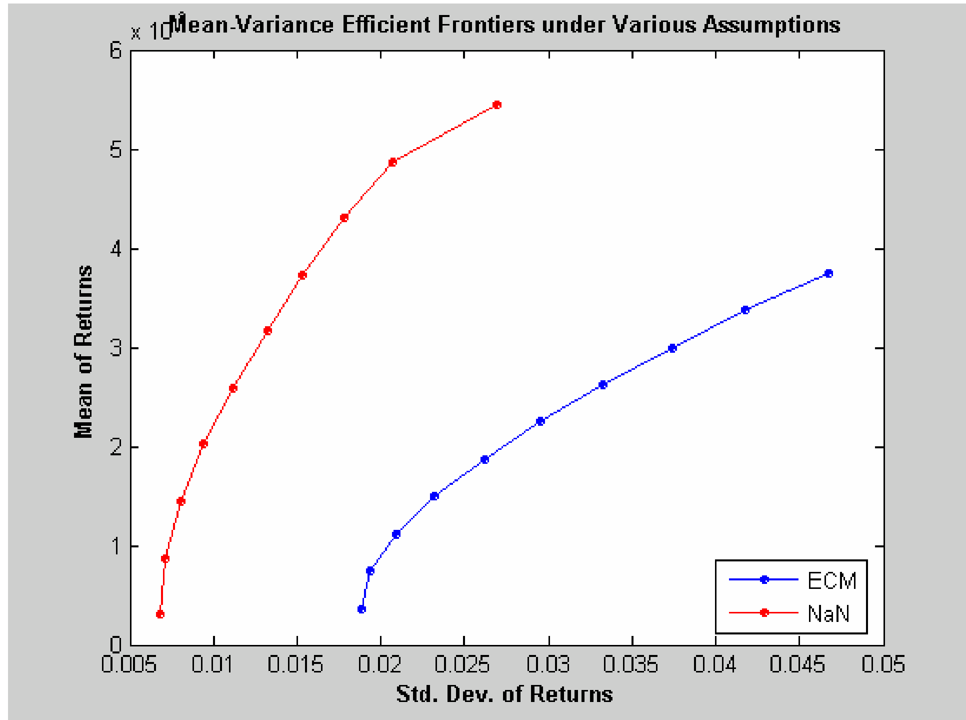
```
0.0005    0.0004    0.0007    0.0004    0.0013    0.0007
0.0006    0.0004    0.0007    0.0005    0.0007    0.0020
```

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);
```

Finally, plot the results on the same graph to illustrate the differences.

```
figure(gcf)
plot(ECMRisk,ECMReturn,'-bo','MarkerFaceColor','b','MarkerSize', 3);
hold all
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
```



Clearly, the naïve approach is extremely optimistic about the risk-return tradeoffs 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    0.0000    0.4639
    0.1648    0.0403    0.0000         0   -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    0.8733
         0         0   -0.0000    0.0000         0    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    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   -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 and to recommend increased weights in Google relative to the naïve weights.

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');
```

Now, 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

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 its 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 with the M-file `CAPMdemo.m`.

The 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

$$E[R_i] = C + \beta_i \ (E[M] - C)$$

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 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 $\beta_1, \dots, \beta_n$ is called asset betas.

Note that the beta of an asset has the form

$$\beta_i = \frac{\mathrm{cov}(R_i, M)}{\mathrm{var}(M)}$$

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; and 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

$$R_{k,i} = \alpha + C_k + \beta_i (M_k - C_k) + V_{k,i}$$

for samples $k = 1, \ldots, m$ and assets $i = 1, \ldots, n$, 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 $\alpha_1, \ldots, \alpha_n$ 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 $i, j = 1, \ldots, n$, where the parameters $S_{11}, \ldots, S_{nn}$ are called residual or nonsystematic variances/covariances.

The square root of the residual variance of each asset, i.e., $\mathrm{sqrt}(S_{ii})$ for $i = 1, \ldots, n$, is said to be the residual or nonsystematic risk of the asset since it characterizes the residual variation in asset prices that cannot be 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 extremely 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, e.g., 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.

Separate Estimation of Some Technology Stock Betas

To illustrate how to use the missing-data regression routines, we estimate betas for twelve technology stocks, where one stock (GOOG) is an IPO.

First, load dates, total returns, and ticker symbols for the twelve 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 `NaN`s. One of the two stocks had an IPO during this period and, consequently, has significantly less data than the other stocks.

The first step is to 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,'  ---- -------------------- ');
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] = ecmmvnrmle(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, i.e., 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 twelve stocks, set up a joint regression model that groups all twelve 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 `emcmvnrmle`, and the routine to estimate standard errors is `ecmmvnrstd`.

Since GOOG has a significant number of missing values, a direct use of the missing data routine `ecmmvnrmle` 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 will be 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 twelve 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] = ecmmvnrmle(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));

tdAlpha = 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] Peter E. Caines, *Linear Stochastic Systems*, John Wiley & Sons, Inc., 1988.

[2] Harald Cramér, *Mathematical Methods of Statistics*, Princeton University Press, 1946.

[3] 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.

[4] William H. Greene, *Econometric Analysis*, 5th ed., Pearson Education, Inc., 2003.

[5] R. A. Jarrow, *Finance Theory*, Prentice-Hall, Inc., 1988.

[6] J. Lintner, “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] Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[8] Xiao-Li Meng and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

[9] Joe Sexton and Anders Rygh Swensen, “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

[10] J. L. Shafer, *Analysis of Incomplete Multivariate Data*, Chapman & Hall/CRC, 1997.

[11] W. F. Sharpe, “Capital Asset Prices: A Theory of Market Equlibrium under Conditions of Risk,” *Journal of Finance*, Vol. 19, 1964, pp. 425-442.

[12] W. F. Sharpe, G. J. Alexander, and J. V. Bailey, *Investments*, 6th ed., Prentice-Hall, Inc., 1999.

5

Solving Sample Problems

Common Problems in Finance (p. 5-3)	Problems involving bond portfolios and equity options.
Producing Graphics with the Toolbox (p. 5-19)	Use of MATLAB graphics to illustrate financial data.

This section shows how Financial Toolbox functions solve real-world problems. The examples ship with the toolbox as M-files. Try them by entering the commands directly or by executing the M-files.

This chapter contains two major topics:

- Common Problems in Finance

  Shows how the toolbox solves real-world financial problems, specifically:

  - “Sensitivity of Bond Prices to Changes in Interest Rates” on page 5-3
  - “Constructing a Bond Portfolio to Hedge Against Duration and Convexity” on page 5-6
  - “Sensitivity of Bond Prices to Parallel Shifts in the Yield Curve” on page 5-8
  - “Constructing Greek-Neutral Portfolios of European Stock Options” on page 5-12
  - “Term Structure Analysis and Interest Rate Swap Pricing” on page 5-15

- Producing Graphics with the Toolbox

  Shows how the toolbox produces presentation-quality graphics by solving these problems:

  - “Plotting an Efficient Frontier” on page 5-19
  - “Plotting Sensitivities of an Option” on page 5-21
  - “Plotting Sensitivities of a Portfolio of Options” on page 5-23

Common Problems in Finance

Sensitivity of Bond Prices to Changes in 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 with 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 the Financial Toolbox. 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. The example M-file is `ftspex1.m`.

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 convextity 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 (verify these results by running the example M-file `ftspex1.m`):

- 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 below for “Constructing a Bond Portfolio to Hedge Against Duration and Convexity”.
- 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`.

Constructing a Bond Portfolio to Hedge Against Duration and Convexity

This example constructs a bond portfolio to hedge the portfolio of "Sensitivity of Bond Prices to Changes in Interest Rates." 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 very easy to do in MATLAB. The M-file for this example is `ftspex2.m`.

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 (i.e., 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 as well as 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 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. (Verify the analysis results by running the example M-file `ftspex2.m`.)

- 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.

Sensitivity of Bond Prices to Parallel Shifts in the Yield Curve

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 can 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 the 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. This example M-file is `ftspex3.m`.

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 (i.e., $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.

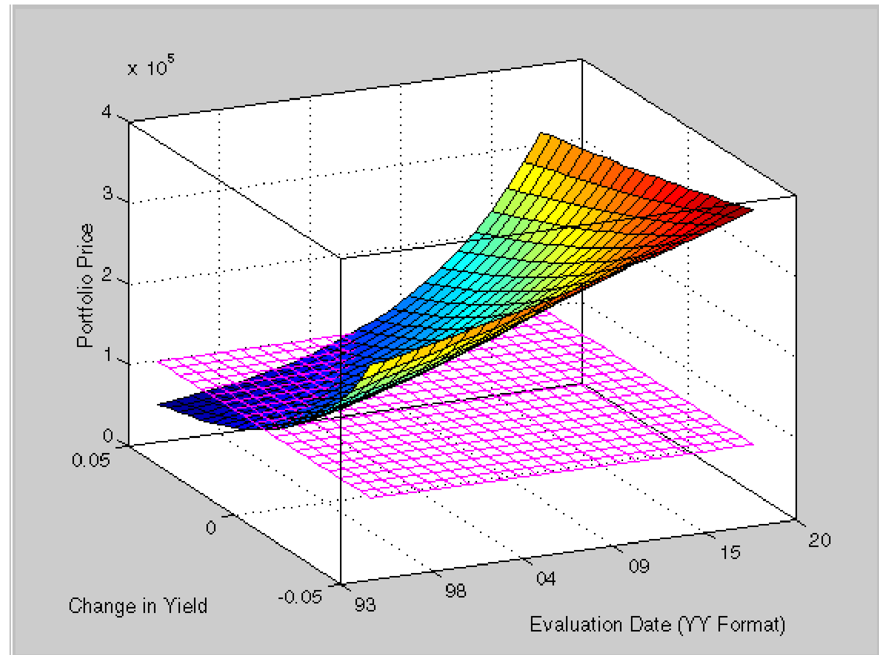
```
set(basemesh, 'facecolor', 'none');
set(basemesh, 'edgecolor', 'm');
set(gca, 'box', 'on');
```

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.

Constructing 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 2-34 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 wish 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 wish 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. This example M-file is `ftspex4.m`.

Step 1. Create an input data matrix to summarize the relevant information. Each row of the matrix contains the standard inputs to the 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, as well as 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 backslash (`\`) 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 example `ftspex4.m` performs these computations and displays the output on the screen.

```
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 Swap Pricing

This example illustrates some of the term-structure analysis functions found in the Financial Toolbox. 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 principle 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 present 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 us 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(strvcat(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 simply 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 example `ftspex5.m` performs these computations and displays the output on the screen.

```
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

The Financial Toolbox and MATLAB graphics functions work together to produce presentation quality graphics, as these examples show. The examples ship with the toolbox as M-files. Try them by entering the commands directly or by executing the M-files. For help using MATLAB plotting functions, see "Creating 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 example M-file is `ftgex1.m`.

First, specify the expected returns, standard deviations, and correlation matrix for a hypothetical portfolio of three assets.

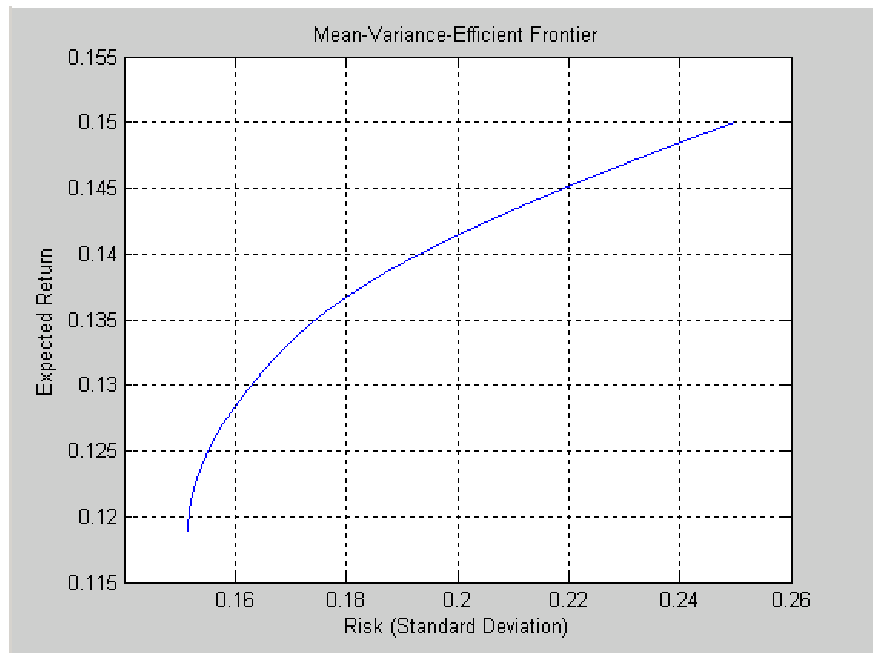
```
Returns      = [0.1 0.15 0.12];
STDs         = [0.2 0.25 0.18];

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.

```
rand('state', 0)
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.

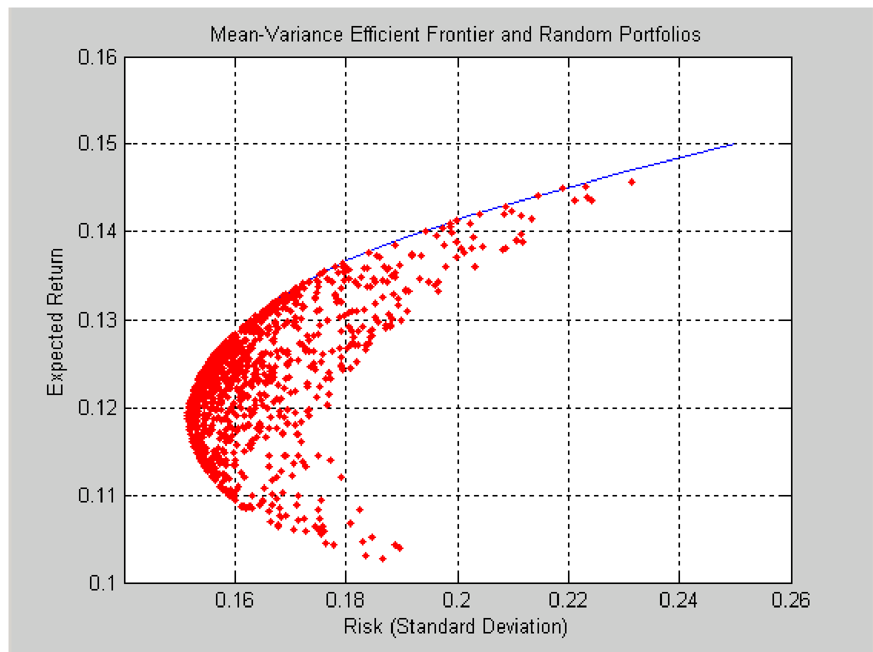
```
Total = sum(Weights, 2);      % Add the weights
Total = Total(:,ones(3,1));   % Make size-compatible matrix
Weights = Weights./Total;     % Normalize so sum = 1
```

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. This example M-file 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.

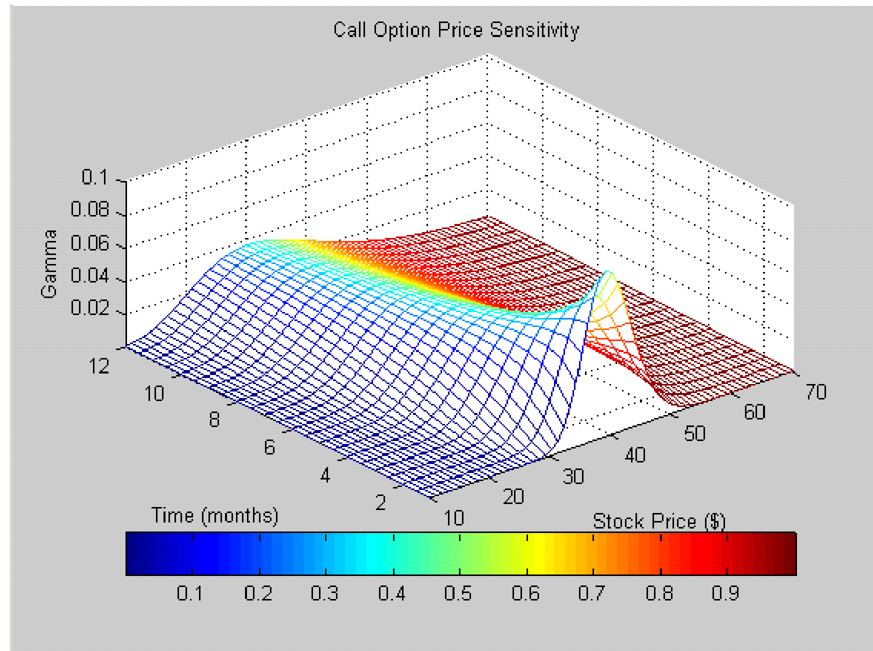
```
JSpan = ones(length(j),1);
NewRange = Range(JSpan,:);
Pad = ones(size(Newj));
```

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.

This example M-file 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.

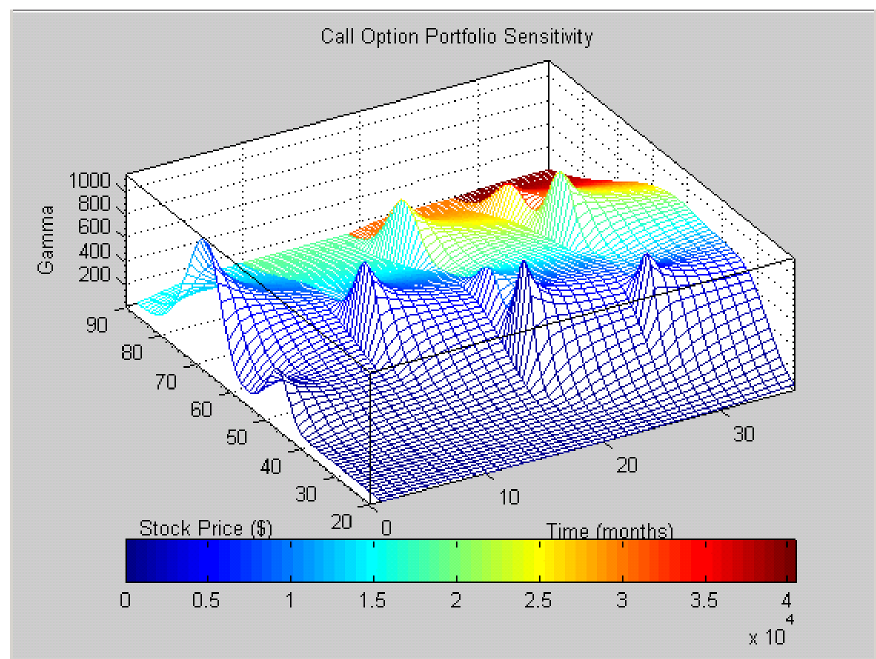
```
    ZVal(36-Time(i)+1:36,:) = ZVal(36-Time(i)+1:36,:) ...
        + NumOpt(i) * blsgamma(NewR, ExPrice(i)*Pad, ...
        Rate(i)*Pad, NewT/36, Sigma(i)*Pad);

    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');
```



6

Financial Time Series Analysis

Analyzing Financial Time Series (p. 6-2)	Analysis of time series data in the financial markets
Creating Financial Time Series Objects (p. 6-3)	Using the constructor or transforming a text file to create a financial time series object
Visualizing Financial Time Series Objects (p. 6-17)	Using `chartfts` and additional specialized tools to observe time series values

Analyzing Financial Time Series

The Financial Toolbox 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 chapter discusses how to create a financial time series object in one of two ways:

- "Using the Constructor" on page 6-3
- "Transforming a Text File" on page 6-13

The chapter also discusses `chartfts`, a graphical tool for visualizing financial time series objects. You can find this discussion in "Visualizing Financial Time Series Objects" on page 6-17.

Creating Financial Time Series Objects

The Financial Toolbox 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`, etc.

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 6-4.
  - See “Time-of-Day Information Included” on page 6-6.
- Separate Vector Input
  - See “Time-of-Day Information Excluded” on page 6-7.
  - See “Time-of-Day Information Included” on page 6-8.
- See “Data Name Input” on page 6-10.
- See “Frequency Indicator Input” on page 6-12.
- See “Description Field Input” on page 6-13.

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 "Handling and Converting 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` just created. 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'`

See the `fints` function description in the Function Reference for a list of frequency indicators.

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 a 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

- `'ddmmmyy hh:mm'` or `'ddmmmyyyy 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

- 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.
  - `'ddmmmyy'` or `'ddmmmyyyy'`
  - `'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 `<matlab>/toolbox/ftseries`.

The syntax of the function

```
fts = ascii2fts(filename, descrow, colheadrow, skiprows);
```

takes in the data filename (`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 as well as 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

The Financial Toolbox 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. The function additionally provides two specialized tools for extracting additional information about the displayed data series:

- "Zoom Tool" for focus on a specific time period within the time frame covered by the time series
- "Combine Axes Tool" to look for patterns among the various data series

---

Note Interactive charting is also available from the **Graphs** menu of the graphical user interface. See "Interactive Chart" on page 8-16 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.

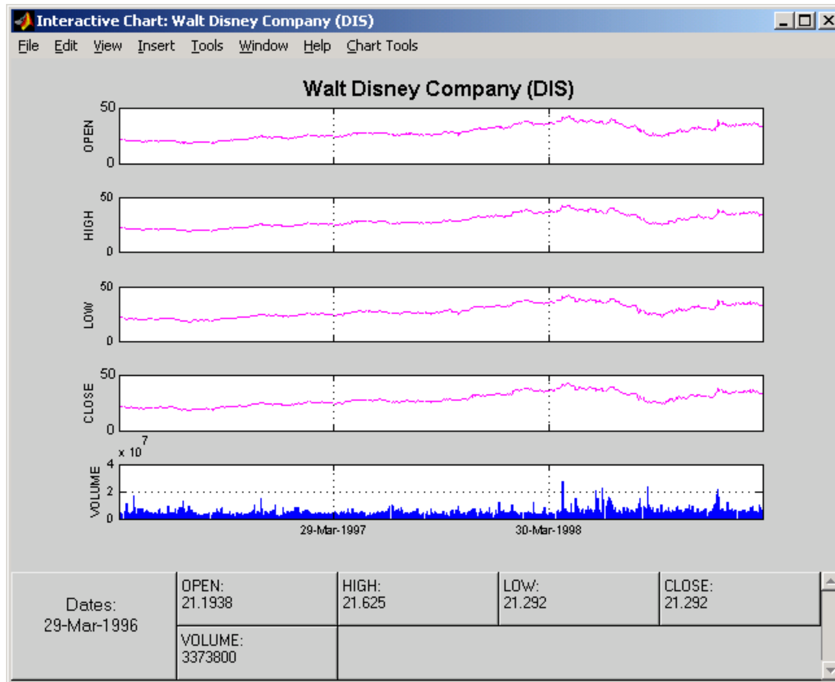
To illustrate the use of `chartfts`, use the equity price and volume traded data for the Walt Disney Corporation (NYSE: DIS) provided in the file `disney.mat`:

```
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.

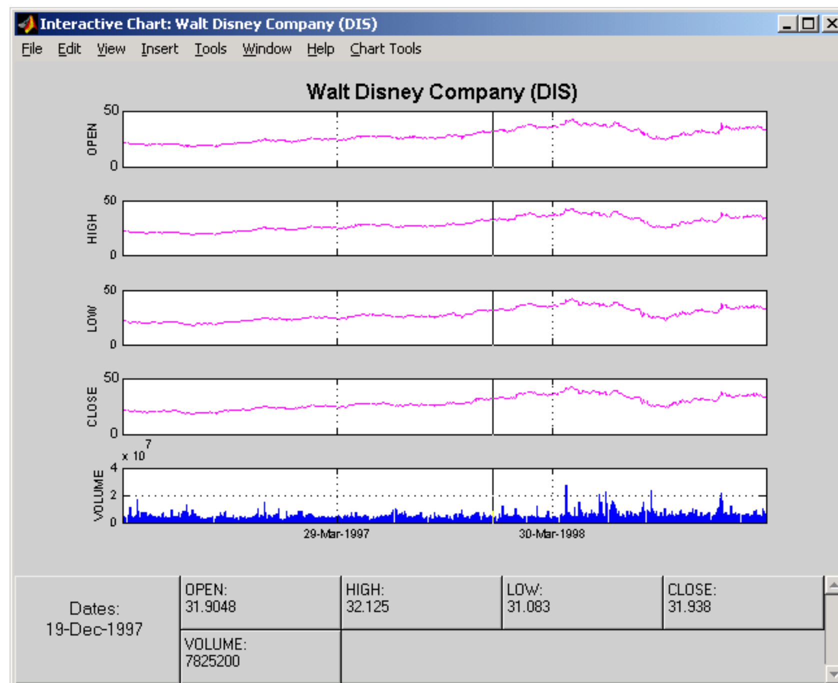
Use `chartfts(dis)` to observe the values.



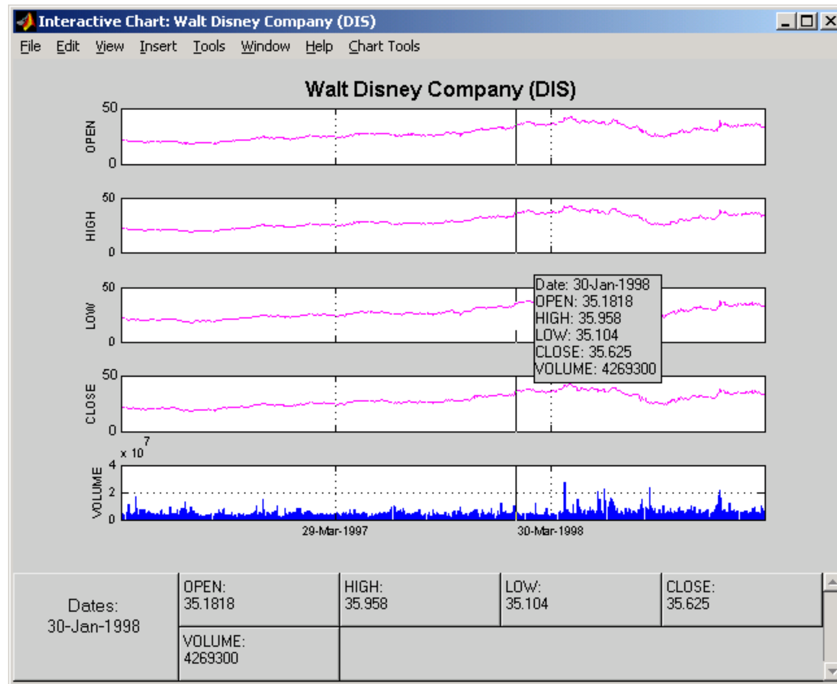
The chart contains five plots, each representing one of the series in the time series object. Boxes indicate the value of each individual plot. The date box is always on the left. The number of data boxes on the right depends upon the number of data series in the time series object, five in this case. 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.

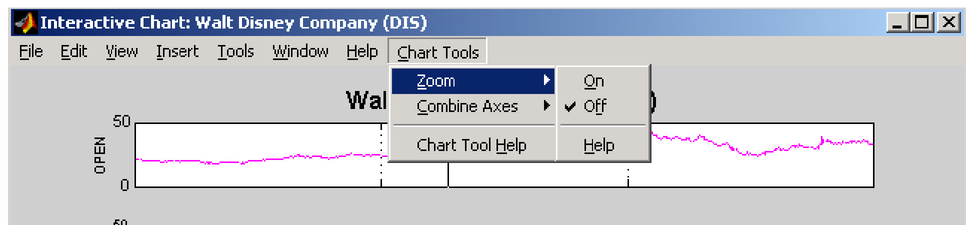


Click the plot. A small information box displays the data at the point where you click the mouse button.



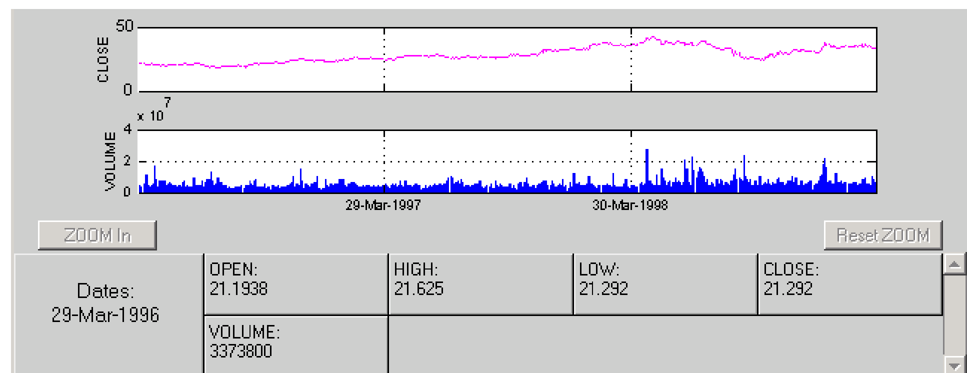
Zoom Tool

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.

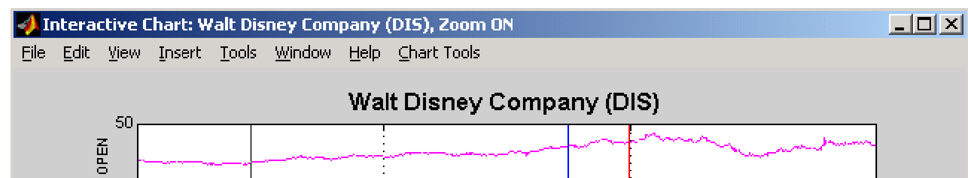


Note Due to the specialized nature of this feature, do not use the MATLAB `zoom` command or **Zoom In** and **Zoom Out** from the **Tools** menu.

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.

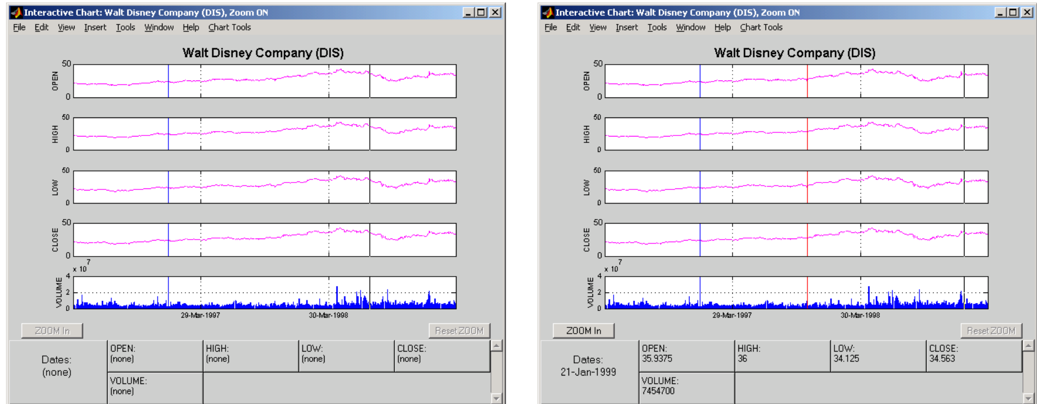


The figure window title bar displays the status of the chart tool that you are using. With the Zoom tool turned on, you see **Zoom ON** in the title bar in addition to the name of the time series you are working with. When the tool is off, no status is displayed.

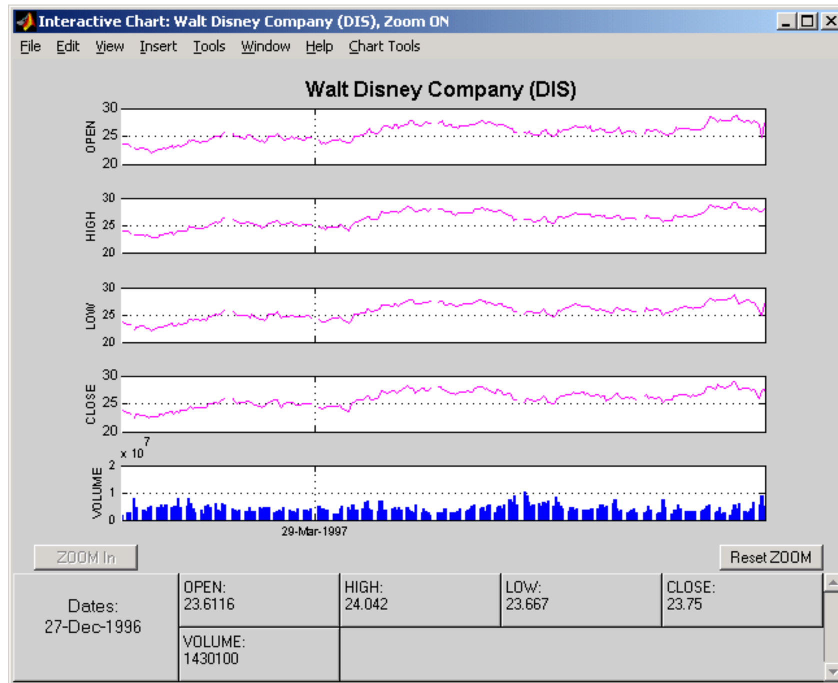


To zoom into the chart, you need to define the starting and ending dates. Define the starting date by moving the cursor over the chart until the desired date appears at the bottom left box and click the mouse button. A blue vertical line indicates the starting date you have selected. Next, again move 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.

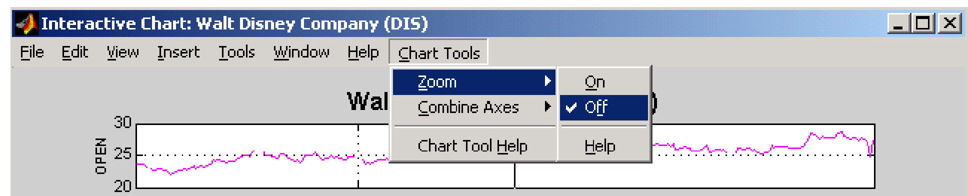


To zoom into the chart, click the **ZOOM In** button.



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**.



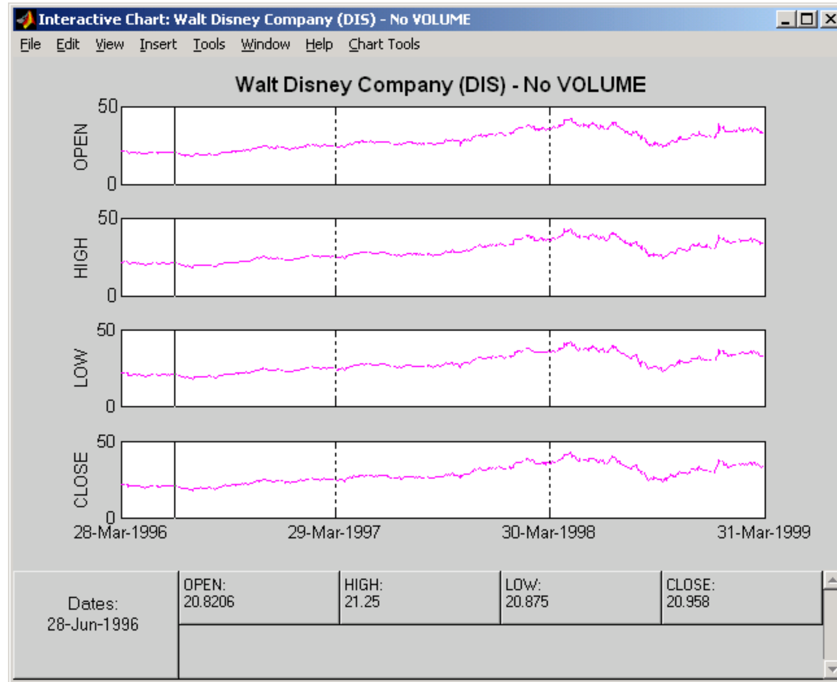
With the tool turned off, the chart stays at the last state that it was in. If you turn it off when the chart is zoomed in, the chart stays zoomed in. If you reset the zoom before turning it off, the chart becomes the original (not zoomed).

Combine Axes Tool

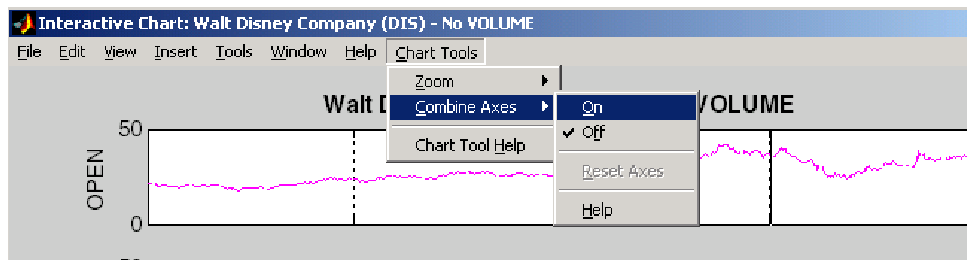
The Combine Axes tool allows you to combine all axes or specific axes into one. With axes combined you can visually spot any trends that can occur among the data series in a financial time series object.

To illustrate this tool, use `dis_nv`, the financial time series object that does not contain volume traded data:

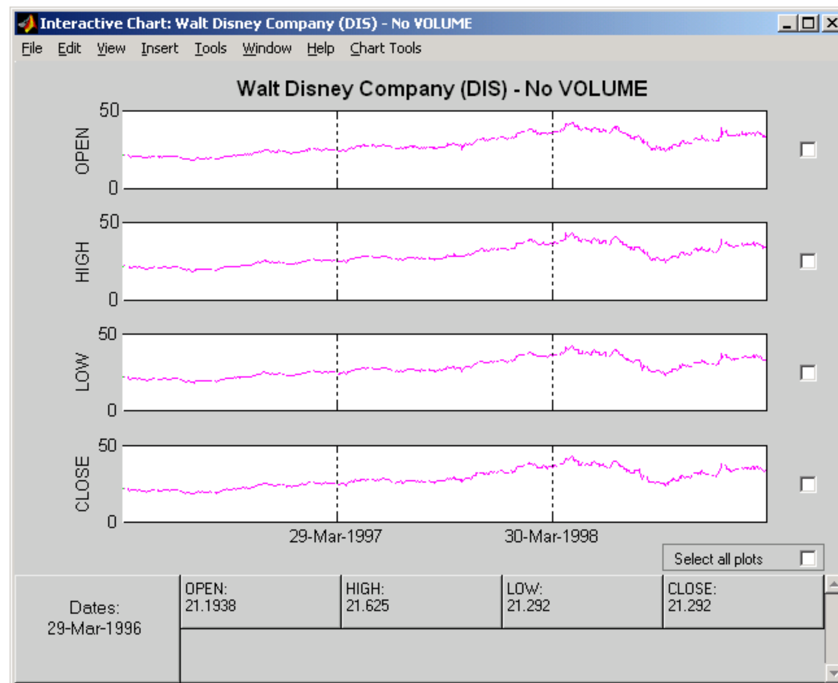
```
chartfts(dis_nv)
```



To combine axes, choose the **Chart Tools** menu, followed by **Combine Axes** and **On**.

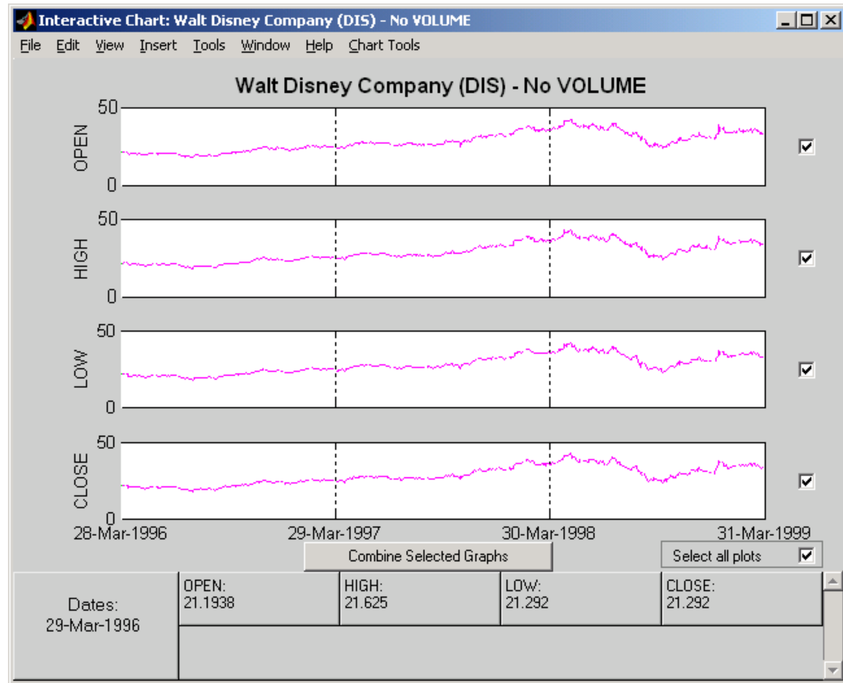


When the Combine Axes tool is on, check boxes appear beside each individual plot. An additional check box enables the combination of all plots.

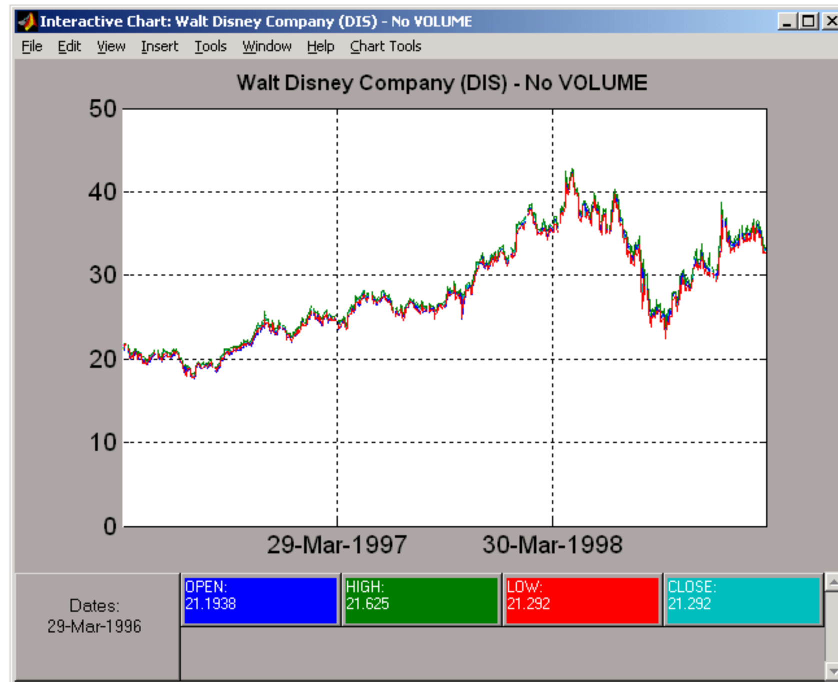


Combining All Axes

To combine all plots, click the check box for **Select all plots.**



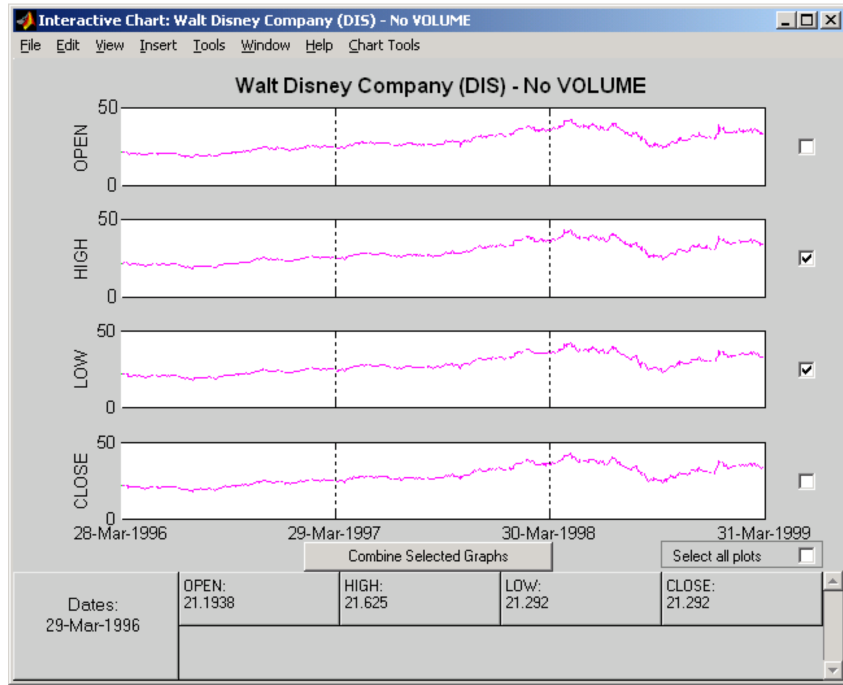
Now click the **Combine Selected Graphs** button to combine the chosen plots. In this case, all plots are combined.



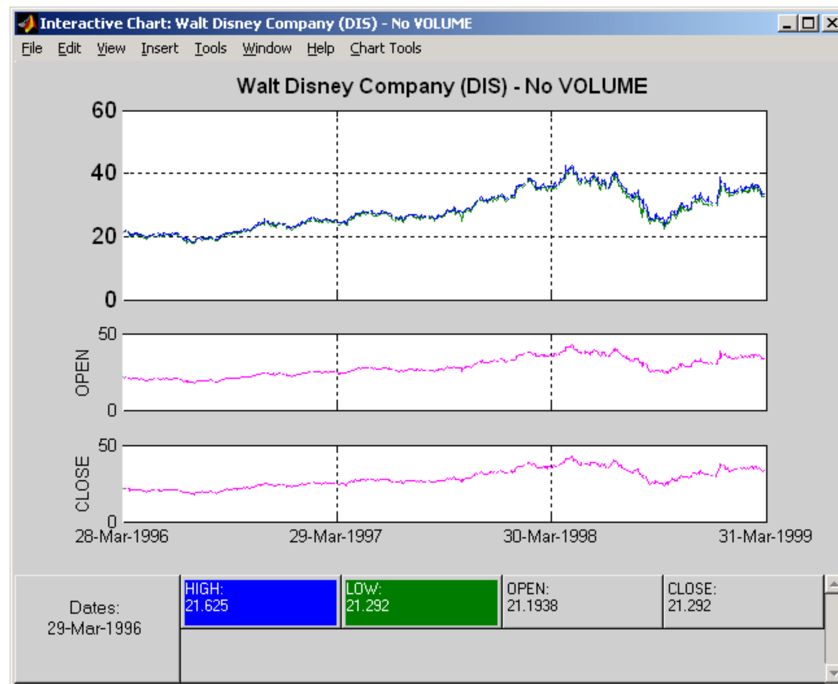
The combined plots have a single plot axis with all data series traced. The background of each data box has changed to the color corresponding to the color of the trace that represents the data series. After the axes are combined, the tool is turned off.

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.



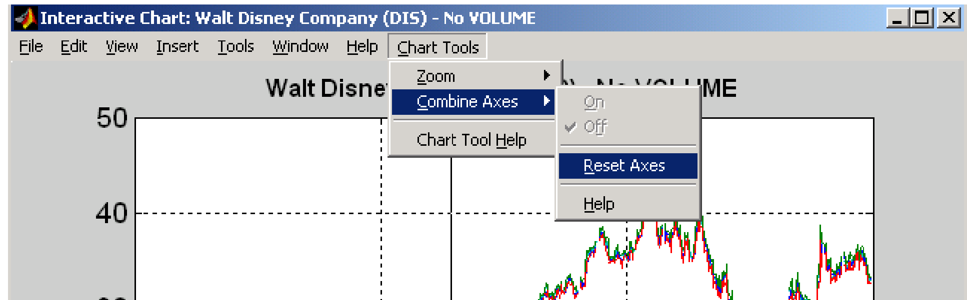
Click the **Combine Selected Graphs** button. The chart with the combined plots looks like the next figure.



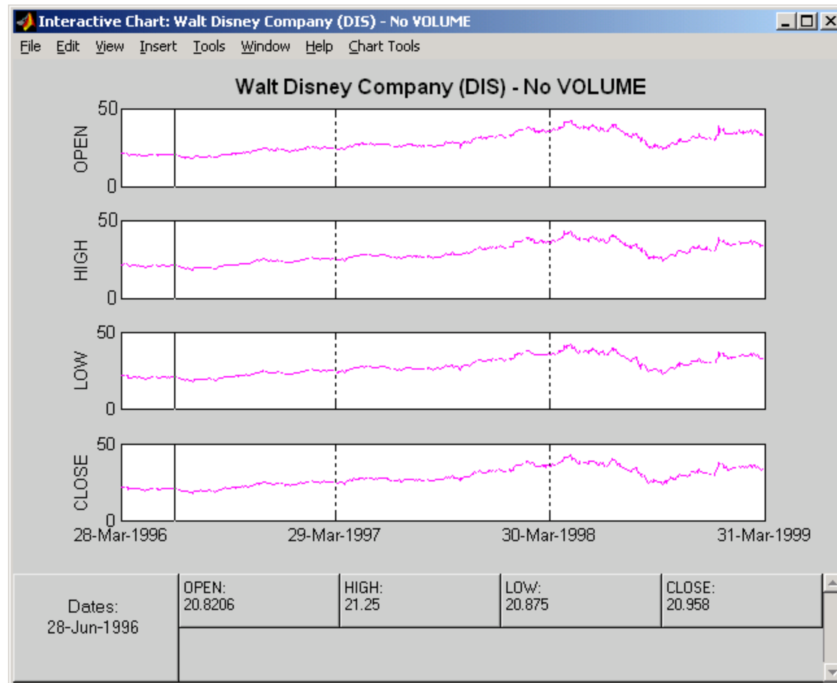
The plot with the combined axes is located at the top of the chart while the remaining plots follow it. The data boxes have also been changed. The boxes that correspond to the combined axes are relocated to the beginning, and the background colors are set to the color of the respective traces. The data 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.



7

Using Financial Time Series

Working with Financial Time Series Objects (p. 7-3)	Extracting time series data and performing operations on time series
Demonstration Program (p. 7-24)	A comprehensive example illustrating the use of the toolbox to predict the return on an equity

Introduction

This chapter discusses how to manipulate and analyze financial time series data. The major topics discussed include

- "Financial Time Series Object Structure" on page 7-3
- "Data Extraction" on page 7-3
- "Object to Matrix Conversion" on page 7-5
- "Indexing a Financial Time Series Object" on page 7-7
- "Operations" on page 7-15
- "Data Transformation and Frequency Conversion" on page 7-19

Much of this information is summarized in the "Demonstration Program" on page 7-24.

Working with Financial Time Series Objects

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 afterwards 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 key, 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
  - `'ddmmmyy hh:mm'` or `'ddmmmyyyy 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, etc., 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
```

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 as well as 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, refer to "Functions - By Category" or 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, etc.) 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. See the “Function Reference” chapter for more details.

Data Transformation and Frequency Conversion

The data transformation and the frequency conversion functions convert a data series into a different format.

Table 7-1: Data Transformation Functions

Function	Purpose
`boxcox`	Box-Cox transformation
`diff`	Differencing
`fillts`	Fill missing values
`filter`	Filter
`lagts`	Lag time series object
`leadts`	Lead time series object
`peravg`	Periodic average

Table 7-1: Data Transformation Functions (Continued)

Function	Purpose
smoothts	Smooth data
tsmovavg	Moving average

Table 7-2: Frequency Conversion Functions

Function	New Frequency
convertto	As specified
resamplets	As specified
toannual	Annual
todaily	Daily
tomonthly	Monthly
toquarterly	Quarterly
tosemi	Semiannually
toweekly	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_whrl = fillts(whrl);
```

Transform the nonnormally distributed filled data series f_whrl into a normally distributed one using Box-Cox transformation:

```
bc_whrl = boxcox(f_whrl);
```

Compare the result of the Close data series with a normal (Gaussian) probability distribution function as well as the nonnormally distributed f_whrl:

```
subplot(2, 1, 1);
hist(f_whrl.Close);
grid; title('Nonnormally Distributed Data');
subplot(2, 1, 2);
hist(bc_whrl.Close);
grid; title('Box-Cox Transformed Data');
```

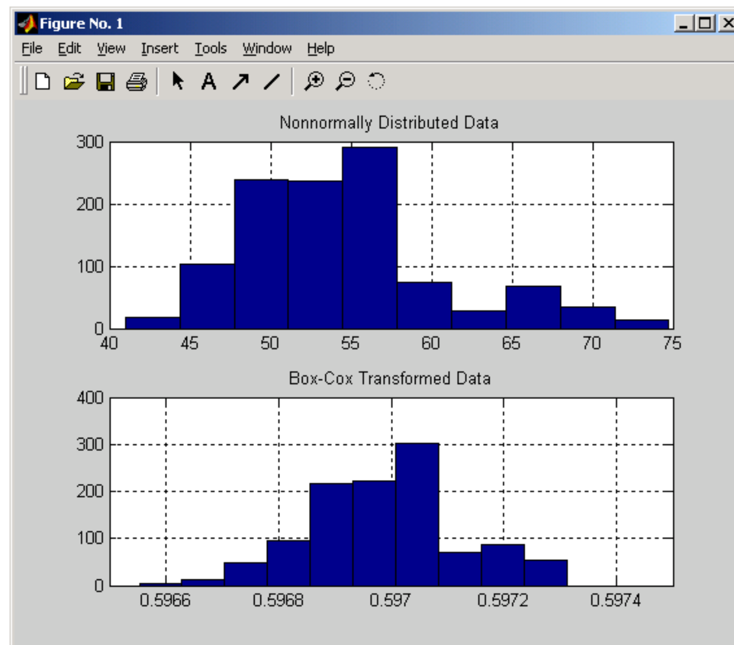


Figure 7-1: Box-Cox Transformation

The bar chart on the top represents the probability distribution function of the filled data series, f_whrl, which is the original data series whrl with the missing values interpolated using the linear method. The distribution is skewed towards 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_whrl`, you find an identical object to the original object `whrl` but the contents are the transformed data series. If you have the Statistics Toolbox, 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.

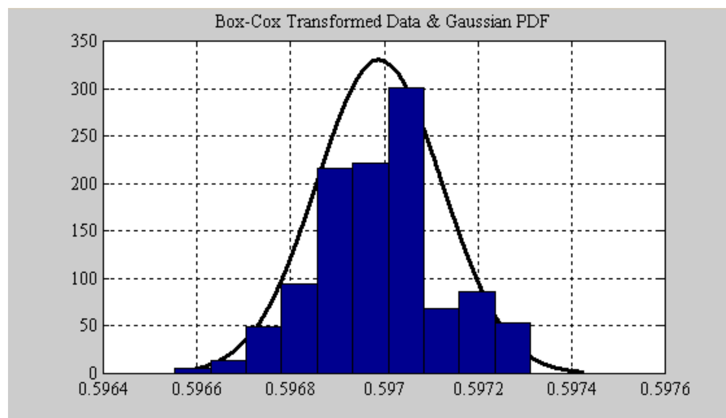


Figure 7-2: Overlay of Gaussian PDF

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:

```
plot(f_ibm.CLOSE('11/01/97::02/28/98'), 'r')
datetick('x', 'mmmyy')
hold on
plot(sm_ibm.CLOSE('11/01/97::02/28/98'), 'b')
hold off
datetick('x', 'mmmyy')
legend('Filled', 'Smoothed')
title('Filled IBM Close Price vs. Smoothed Series')
```

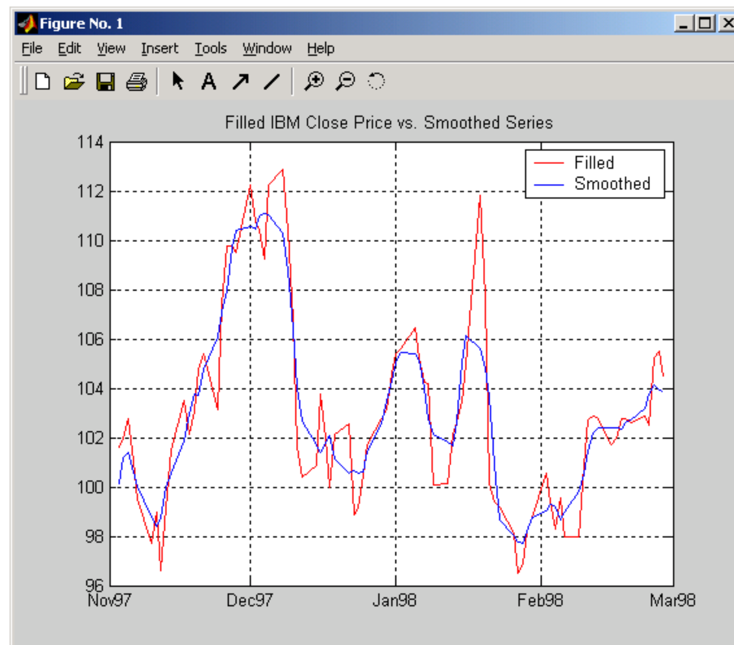


Figure 7-3: Smoothed Data Series

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 the these objects. The overloaded functions are those most commonly needed to work with time series data.

Demonstration Program

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 script M-file for this demonstration program in the directory `<matlab>/toolbox/ftseries/ftsdemos` on your MATLAB path. The script is named `predict_ret.m`.

To perform these computations follow these steps:

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 (e.g., 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 advantageous 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 `todaily` 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 = todaily(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');
```

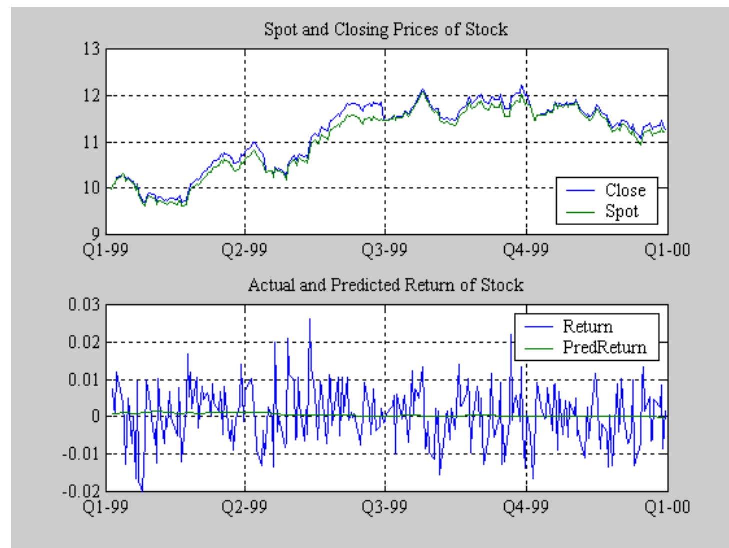


Figure 7-4: Closing Prices and Returns

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. See `fillts` in the "Function Reference" for details.

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'
```

8

Financial Time Series Graphical User Interface

Introduction (p. 8-2)	Menus available on the main window of the financial time series GUI
Using the Financial Time Series GUI (p. 8-7)	A more in-depth exploration of the capabilities of the financial time series GUI

Introduction

Use the financial time series graphical user interface (GUI) to analyze your time series data and display the results graphically without resorting to the command line. The GUI lets you visualize the data and the results at the same time.

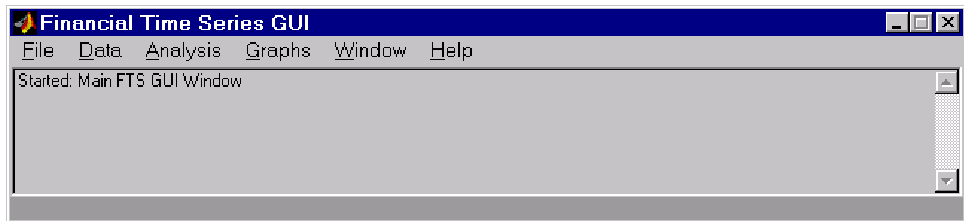
"Using the Financial Time Series GUI" on page 8-7 provides a discussion about how to use this GUI.

Main Window

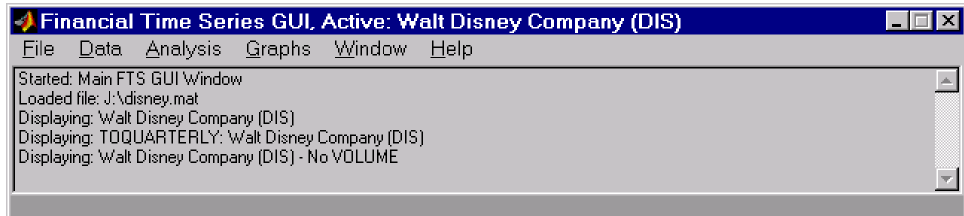
Start the financial time series GUI with the command

```
ftsgui
```

The main financial time series GUI window appears.

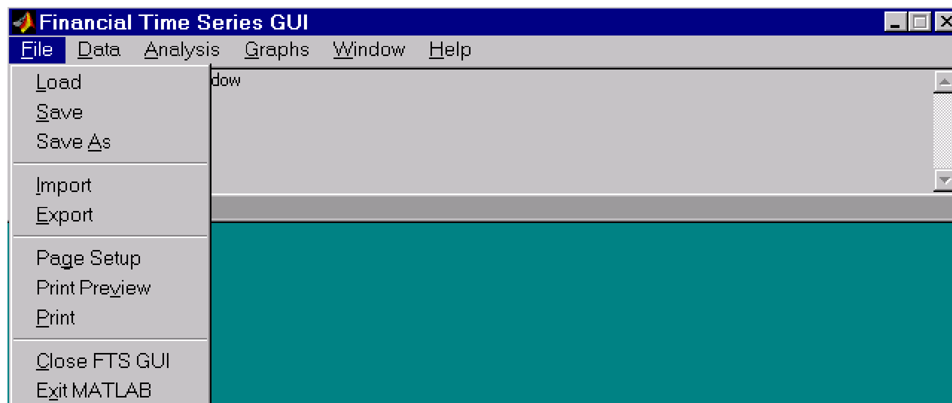


The title bar acts as an active time series object indicator (indicates the currently active financial time series object). For example, if you load the file `disney.mat` and want to use the time series data in the file `dis`, the title bar on the main GUI would read as shown.



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.

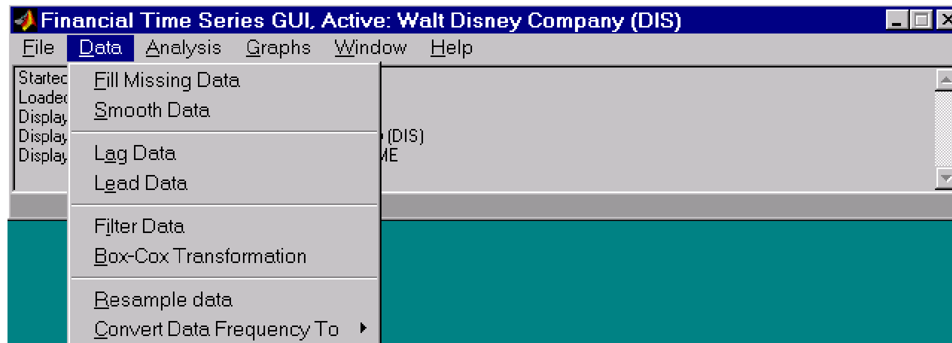
File Menu



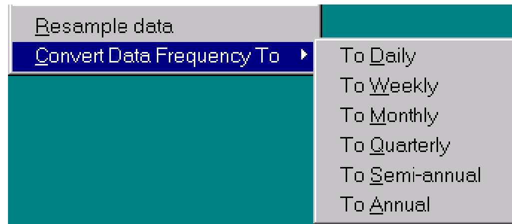
The **File** menu contains the commands for input and output. You can read and save (**Load**, **Save**, and **Save As**) MATLAB MAT-files, ASCII (text) data files, as well as import (**Import**) Microsoft Excel XLS files. MATLAB does not support the export of XLS files at this time.

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

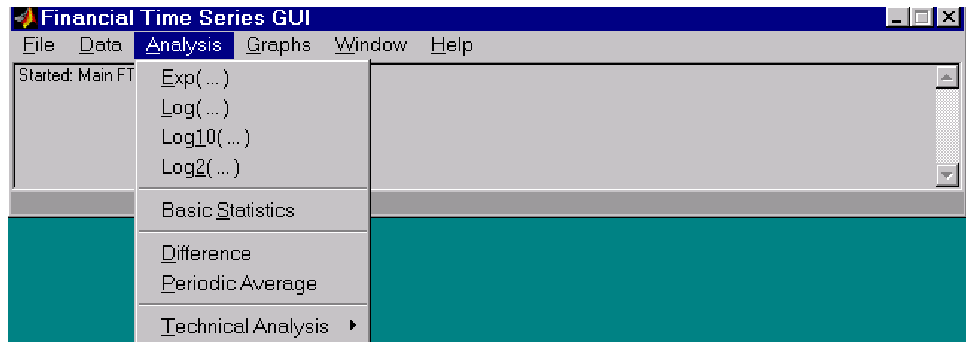


The **Data** menu item provides a collection of data manipulation functions and data conversion functions.



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

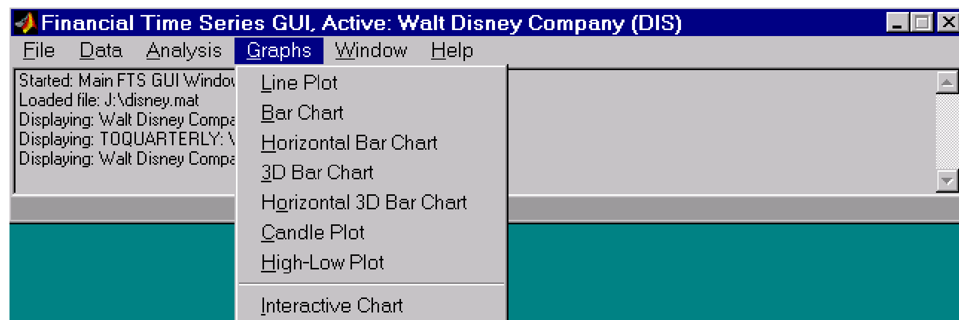


The **Analysis** menu provides

- 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, etc. The periodic average function calculates the average per defined length period, for example, averages of every five days.
- Technical analysis functions. See Chapter 9, “Technical Analysis,” for a list of the provided 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

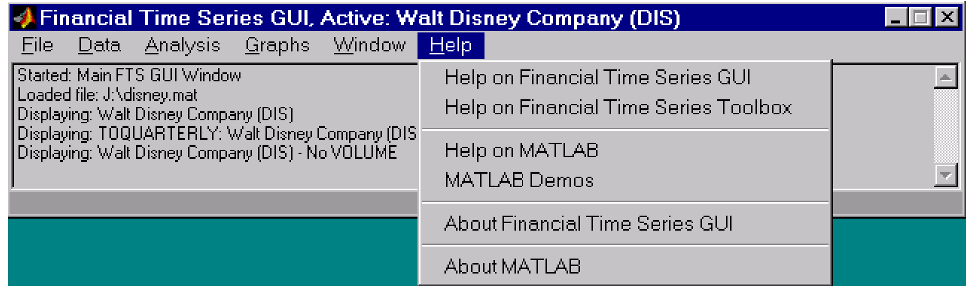


The **Graphs** menu contains functions that graphically display the current (active) financial time series object. You can also invoke the interactive charting function (`chartfts`) from this menu.

Window Menu

The **Window** menu lists open windows under the current MATLAB session.

Help Menu



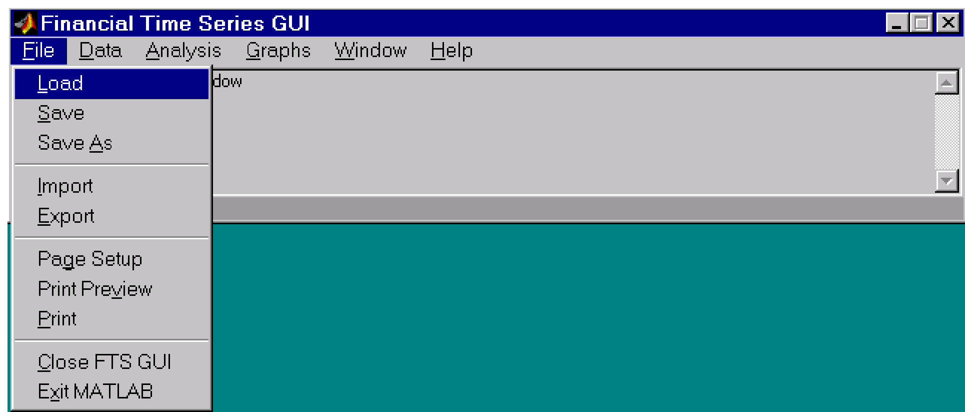
The **Help** menu provides a standard set of Help menu links.

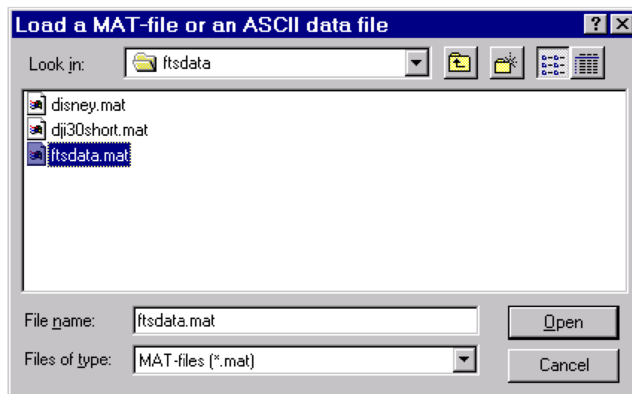
Using the Financial Time Series GUI

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.



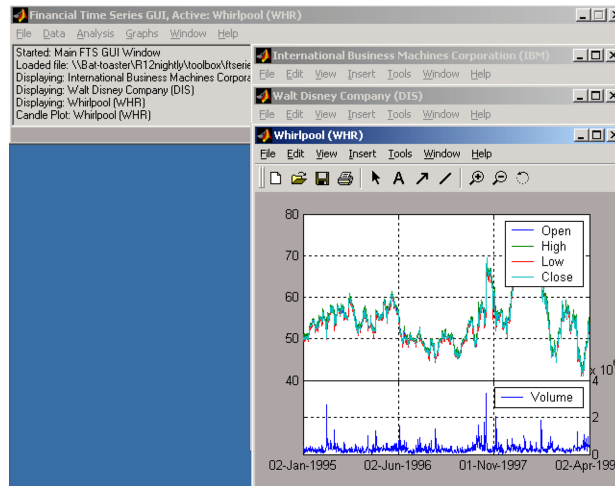


For illustration purposes, choose the file `ftsdata.mat` from the dialog presented.

If you don't see the MAT-file, look in the directory `<matlab>\toolbox\finance\findemos`, where `<matlab>` 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.

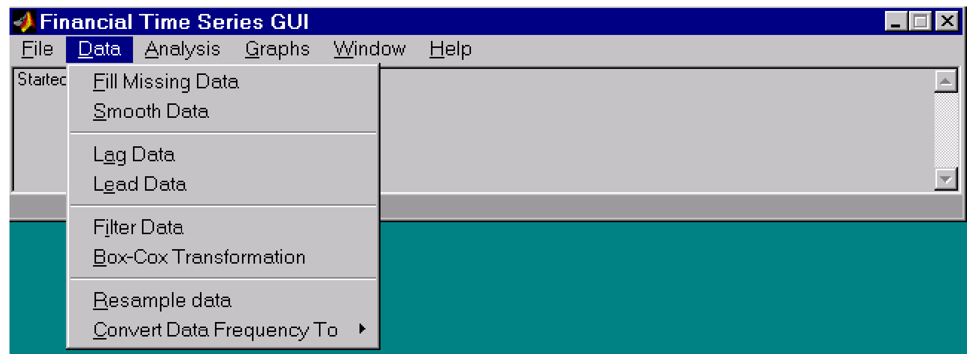
Each financial time series object inside the MAT-file is presented as a line plot in a separate window. The status window is updated accordingly.



Whirlpool (WHR) is the last plot displayed, as indicated on the title bar of the main window.

Data Menu

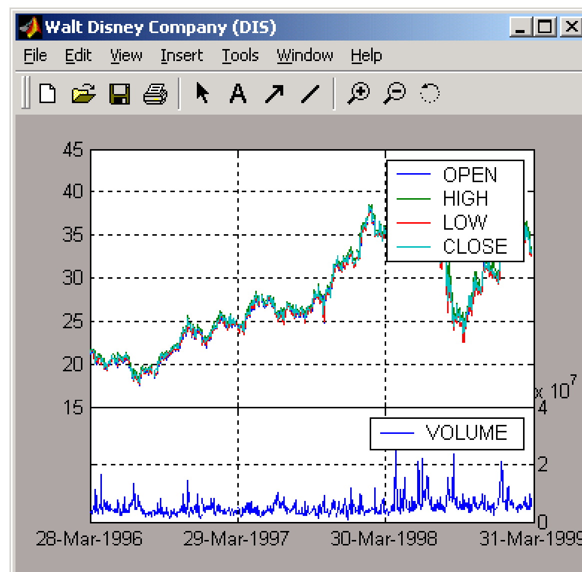
The **Data** menu provides functions that manipulate time series data.



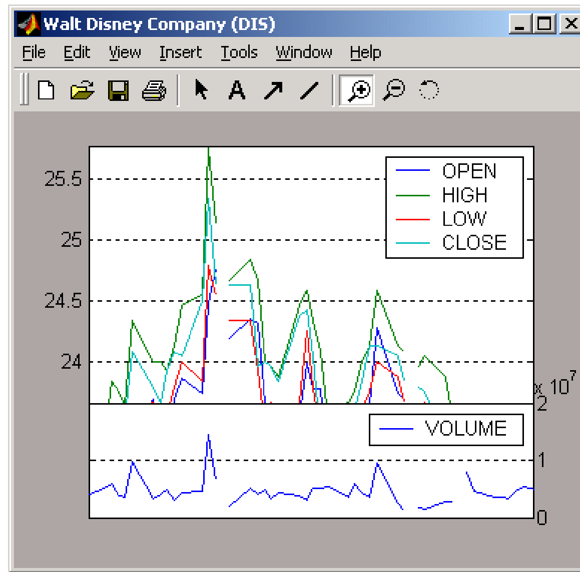
Here are some example tasks that illustrate the use of the functions on this menu.

Fill Missing Data

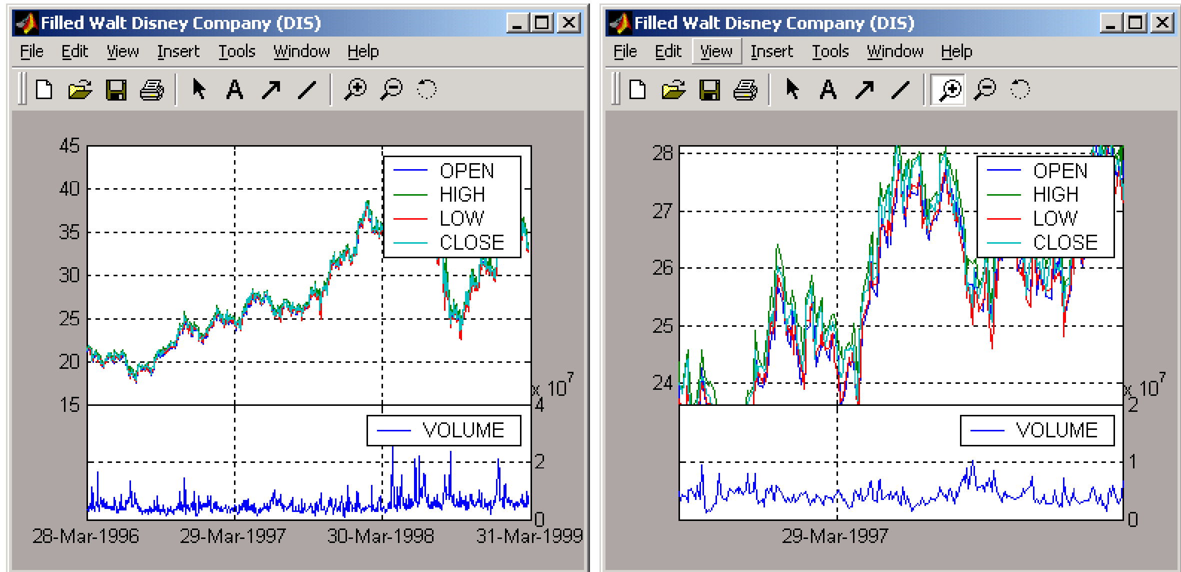
First, look at filling missing data. The **Fill Missing Data** item uses the toolbox function `fillts`. With the data loaded from the file `ftsdata`, you have three time series: IBM Corp. (IBM), Walt Disney Co. (DIS), and Whirlpool (WHR). Click on the window that shows the time series data for Walt Disney 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.



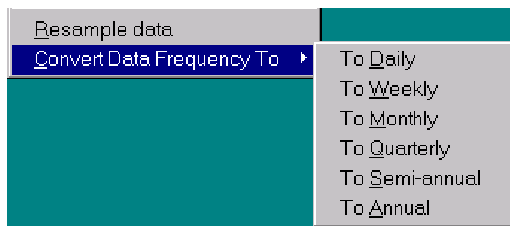
The gaps represent the missing data in the series. To fill these gaps, go to the **Data** menu and choose **Fill Missing Data**. This selection automatically fills the gaps and generates a new plot that displays the filled time series data.



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 on the Walt Disney Company (DIS) window to make it active (current) again. Then, from the **Data** menu, choose **Convert Data Frequency To** and **To Monthly**.

Active variable: **telephone**

Data Table

☑ Show data

	Dates/Ti...	Close	High	Low	Open	Volume
1	31-Jan-20...	22.875	23.16399...	20.719	21.30600...	8848257.0
2	28-Feb-20...	18.4	18.935	17.3975	18.0475	5077970.0
3	31-Mar-20...	17.215	17.655	16.3475	16.82249...	4722830.0
4	30-Apr-20...	15.07500...	16.00999...	13.915	14.94250...	9589337.5
5	30-May-2...	17.33000...	18.398	16.856	18.028	6806132.0
6	30-Jun-20...	20.20999...	20.9025	19.2425	20.0075	5226625.0
7	31-Jul-2003	19.6	19.83	19.02	19.25	3797700.0

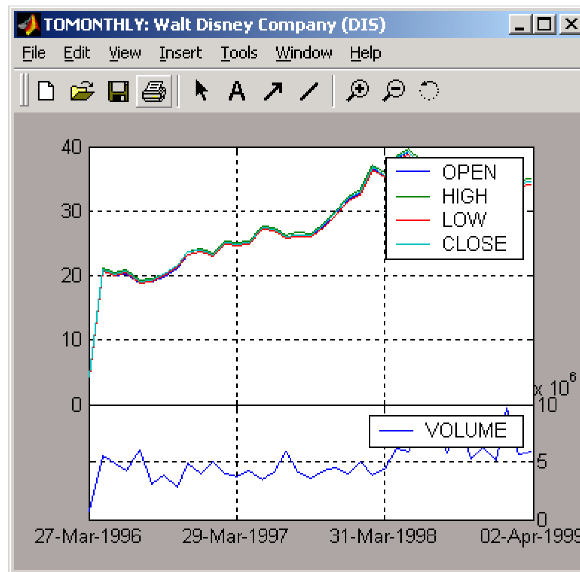
Remove rows: ○ via row #'s ○ via dates

Remove columns: ○ via column #'s ○ via series names

Start:

Columns:

A new figure window displays the result of this conversion.

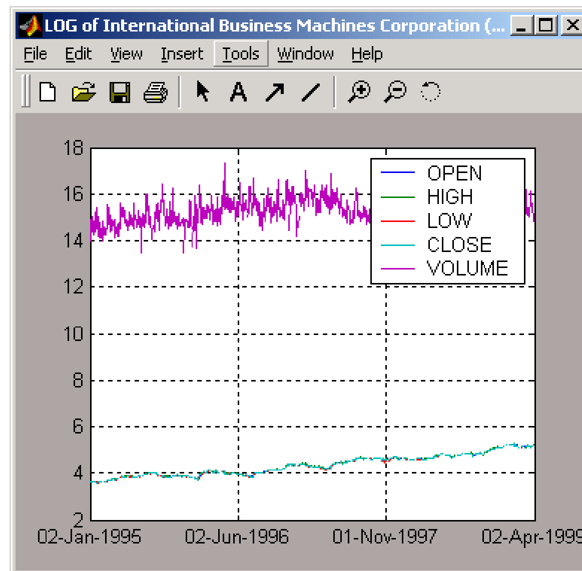


The title reflects that the data displayed had its frequency changed to monthly.

Analysis Menu

The **Analysis** menu provides functions that analyze time series data, including the technical analysis functions. (See Chapter 9, “Technical Analysis,” for a complete list of the technical analysis functions and several usage examples.)

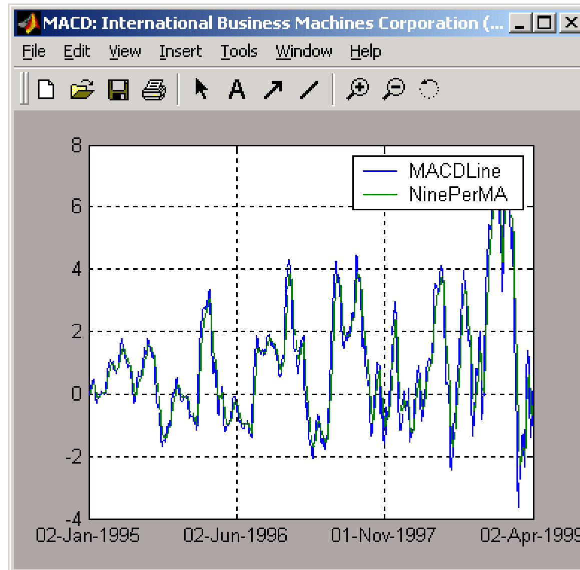
For example, you can use the **Analysis** menu to calculate the natural logarithm (`log`) of the data contained within the data set `ftsdata.mat`. This data file provides time series data for IBM (IBM), Walt Disney (DIS), and Whirlpool (WHR). Click on the window displaying the data for IBM Corporation (IBM) to make it active (current). Then choose the **Analysis** menu, followed by the **Log(...)** menu item. 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, choose **Technical Analysis**, and the **MACD** item. The result, again, is displayed in its own window.



Other analysis functions work similarly.

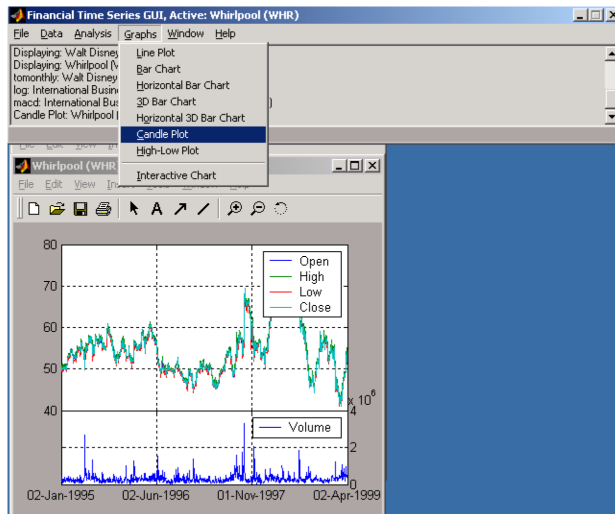
Graphs Menu

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`, `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`.

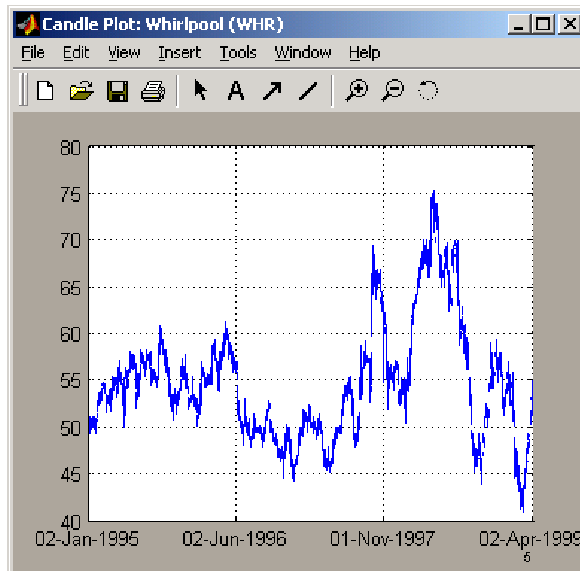
Candle Plot

For example, you can display the candle plot of a set of time series data and invoke the interactive chart on the same data set.

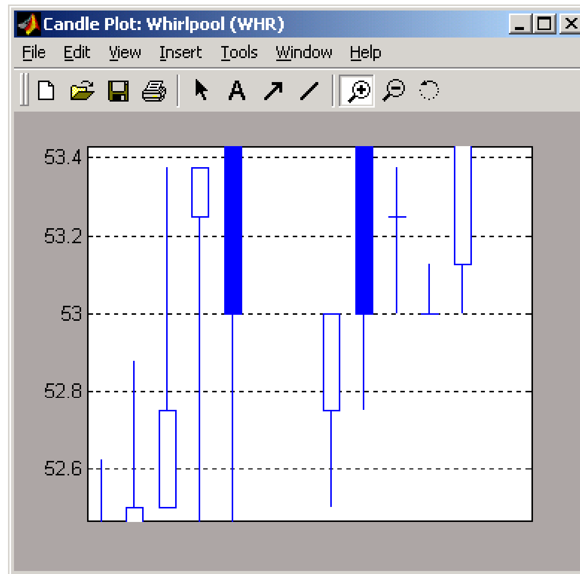
Load the `ftsdata.mat` data set, and click on the window that displays the Whirlpool (WHR) time series data to make it active (current). From the main window choose the **Graphs** menu and **Candle Plot** menu item.



The result is shown below.

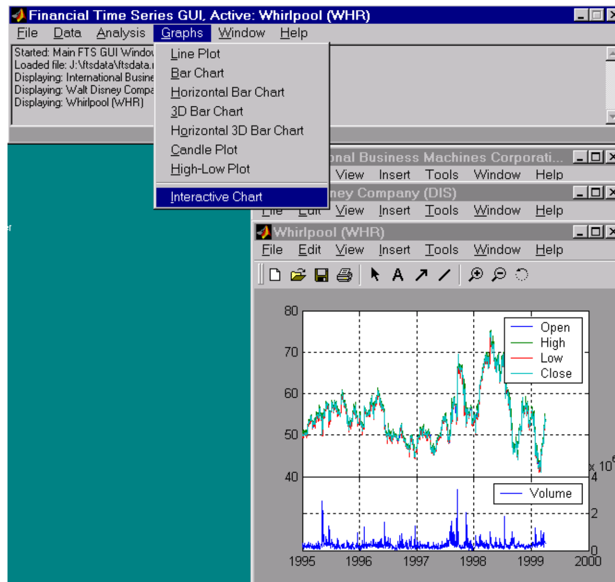


This does not look much like a candle plot because there are too many data points in the data set. All the candles are too compressed for effective viewing. However, when you zoom into a region of this plot, the candles become apparent.

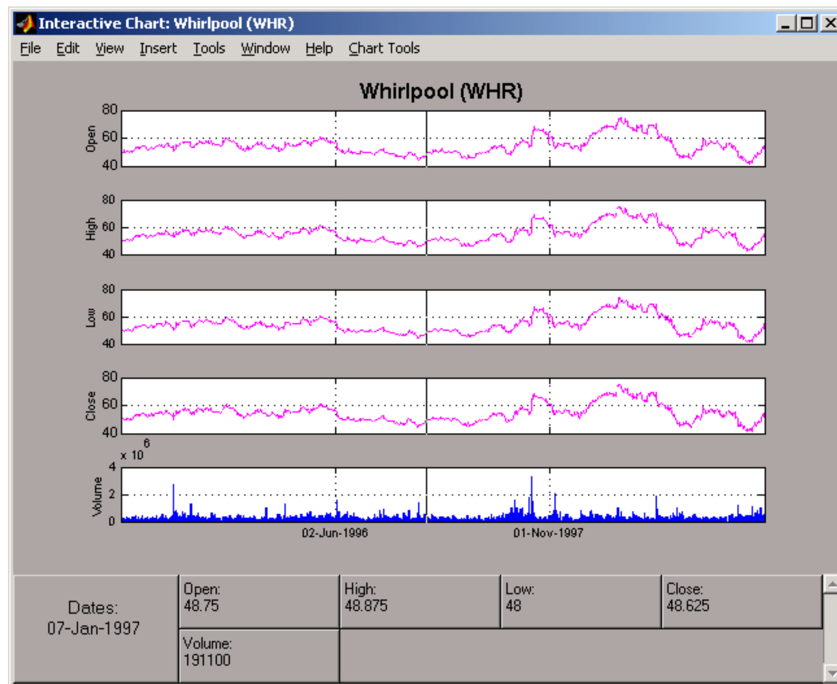


Interactive Chart

To create an interactive chart (`chartfts`) on the Whirlpool data, click on the window that displays the Whirlpool (WHR) data to make it active (current). Then, go to the **Graphs** menu and choose **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 6-17.

Saving Time Series Data

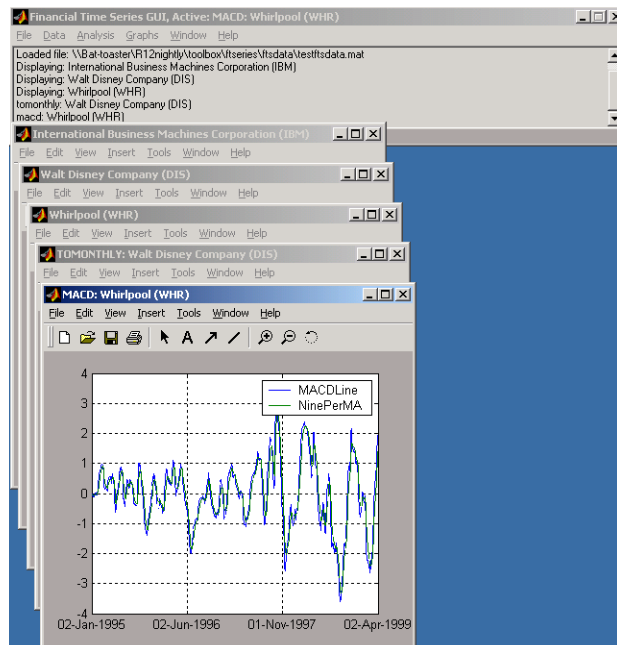
The **Save** and **Save As** items on the main window **File** menu let you save the time series data that results from your analyses and computations. These items save *all* time series data that has been loaded or processed during the current session, even if the window displaying the results of a computation has previously been dismissed.

Note The **Save** and **Save As** items on the **File** menu of individual plot windows are not available for use.

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 dialog window (**Save As**)

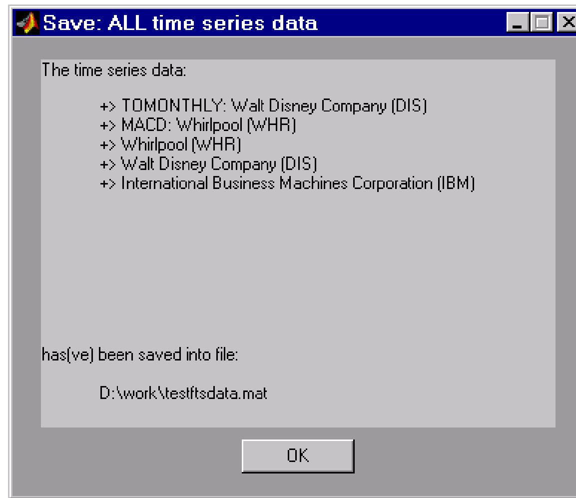
To illustrate this, start by loading the data file `testftsdata.mat` (located in `<matlab>/toolbox/finance/findemos>`. Then, convert the Disney (DIS) data from daily (the original frequency) to monthly data. Next, run the MACD analysis on the Whirlpool (WHR) data. You now have a set of five open figure windows.



Saving into the Original File (Save)

To save the data back into the original file (`testftsdata.mat`), choose **Save** on the **File** menu.

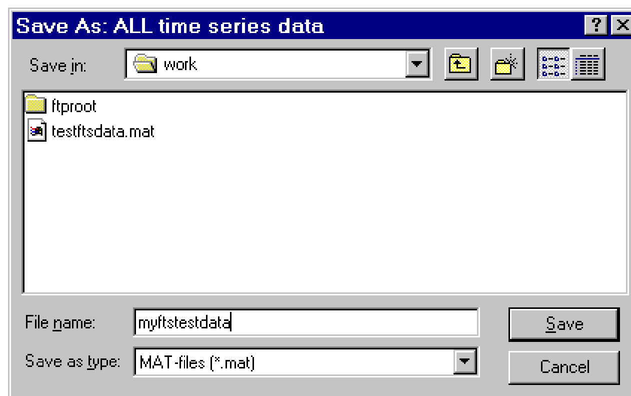
A confirmation window appears. It confirms that the data has been saved in the latest MAT-file loaded (`testftsdata.mat` in this example).



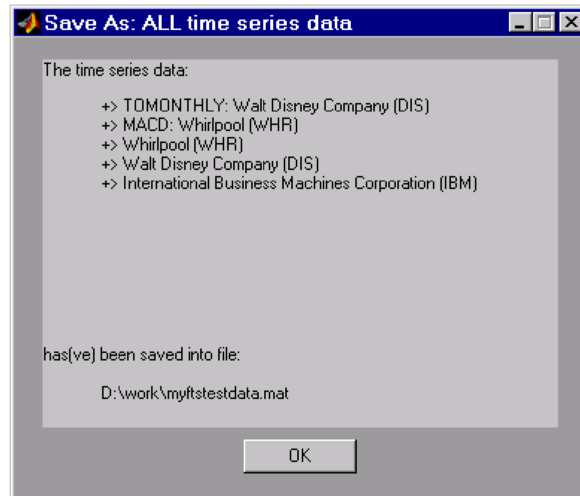
Saving into a New File (Save As)

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`.

9

Technical Analysis

Introduction (p. 9-2)	Tables of technical analysis functions listed by category
Examples (p. 9-5)	Examples showing the use of several technical analysis functions

Introduction

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.

Table 9-1: Technical Analysis: Oscillators

Function	Type
`adosc`	Accumulation/distribution oscillator
`chaikosc`	Chaikin oscillator
`macd`	Moving Average Convergence/Divergence
`stochosc`	Stochastic oscillator
`tsaccel`	Acceleration
`tsmom`	Momentum

Table 9-2: Technical Analysis: Stochastics

Function	Type
`chaikvolat`	Chaikin volatility
`fpctkd`	Fast stochastics
`spctkd`	Slow stochastics
`willpctr`	Williams %R

Table 9-3: Technical Analysis: Indexes

Function	Type
negvolidx	Negative volume index
posvolidx	Positive volume index
rsindex	Relative strength index

Table 9-4: 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

The chapter provides examples for several types of technical analysis:

- “Moving Average Convergence/Divergence (MACD)” on page 9-5
- “Williams %R” on page 9-6
- “Relative Strength Index (RSI)” on page 9-8
- “On-Balance Volume (OBV)” on page 9-9

Examples

To illustrate some 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 three-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);
```

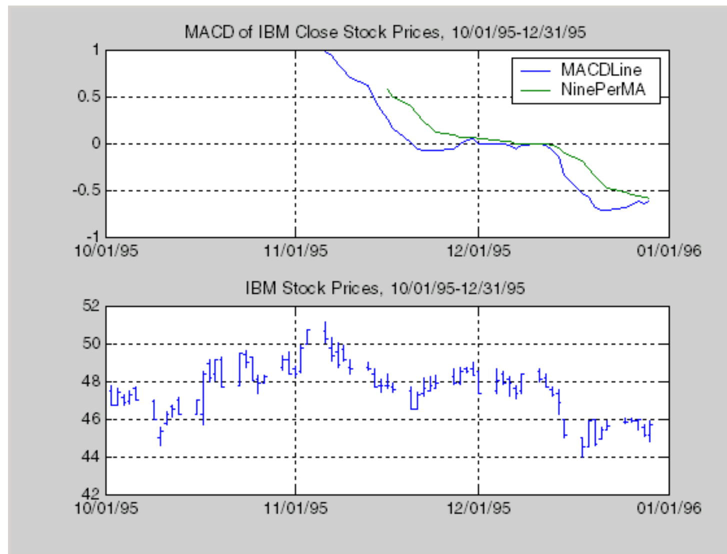
Note When you call `macd` without giving it a second input argument to specify a particular data series name, it searches for a closing price series named `Close` (in all combinations of letter cases).

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')
```

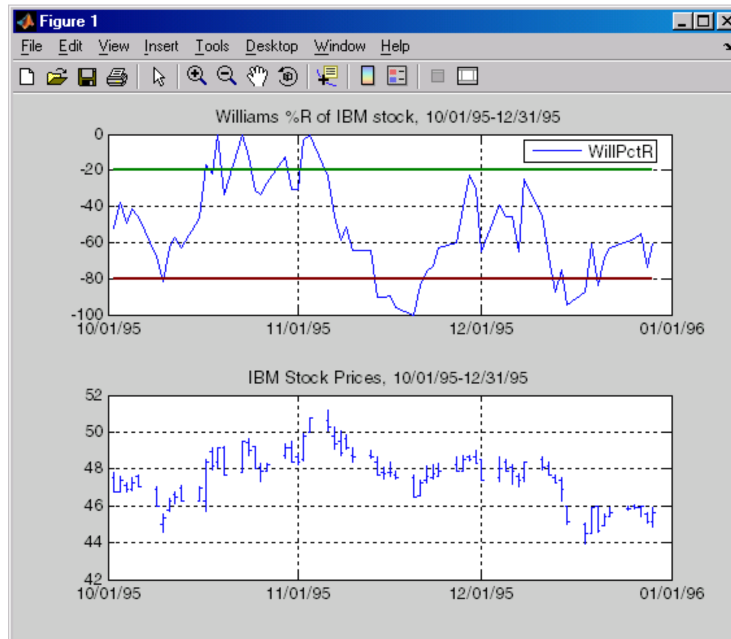
The function `willpctr` now assumes that your high price series is named `Hi`, low price series is named `Lo`, and closing price series is named `Closing`.

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 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`.

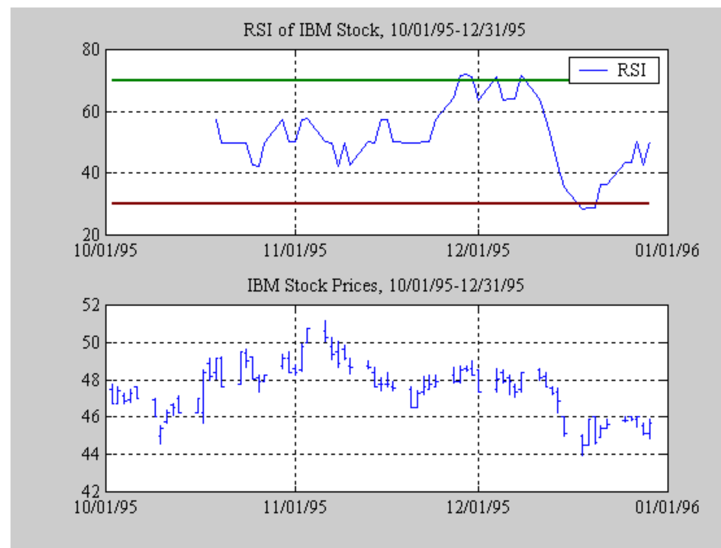
The `rsindex` function needs a series that contains the closing price of a stock. The default period length for the RSI calculation is 14 periods. This length can be changed by providing a second input argument to the function. Similar to the previous commands, if your closing price series is not named `Close`, you can provide the correct name.

Calculate and plot the RSI for IBM 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)

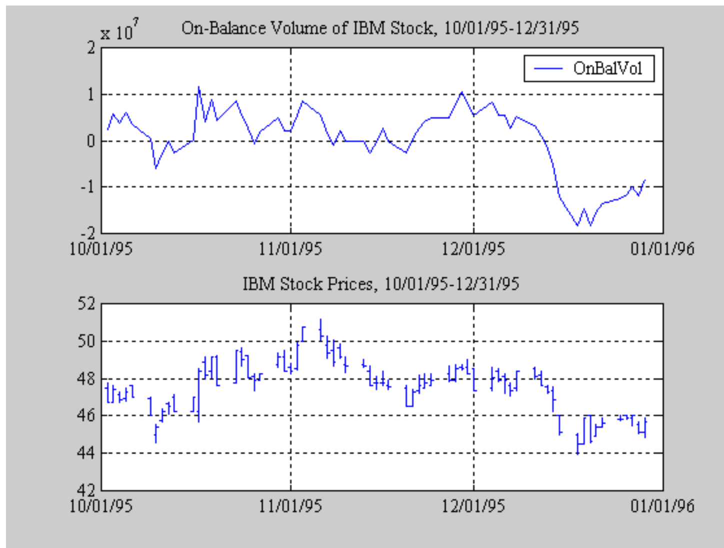
On-Balance Volume (OBV) relates volume to price change. The function `onbalvol` requires you to have the closing price (`Close`) series as well as the volume traded (`Volume`) series.

Calculate and plot the OBV for IBM 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.



10

Function Reference

Functions - By Category (p. 10-2)	Toolbox functions listed by category.
Functions — Alphabetical List (p. 10-19)	Toolbox functions listed alphabetically.

Functions - By Category

"Handling and Converting Dates"

"Formatting Currency and Price"

"Charting Financial Data"

"Analyzing and Computing Cash Flows"

"Fixed-Income Securities"

"Analyzing Portfolios"

"Financial Statistics"

"GARCH Processes"

"Financial Time Series Object and File Construction"

"Financial Time Series Arithmetic Functions"

"Financial Time Series Mathematical Functions"

"Financial Time Series Utility Functions"

"Financial Time Series Data Transformation Functions"

"Financial Time Series Indicator Functions"

"Financial Time Series Graphical User Interface Function"

Handling and Converting Dates

"Current Time and Date"

"Date and Time Components"

"Date Conversion"

"Financial Dates"

"Coupon Bond Dates"

Current Time and Date

`now`	Current date and time.
`today`	Current date.

Date and Time Components

`datefind`	Indices of date numbers in matrix.
`datevec`	Date components.
`day`	Day of month.
`eomdate`	Last date of month.
`eomday`	Last day of month.
`hour`	Hour of date or time.
`lweekdate`	Date of last occurrence of weekday in month.
`minute`	Minute of date or time.
`month`	Month of date.
`months`	Number of whole months between dates.
`nweekdate`	Date of specific occurrence of weekday in month.
`second`	Second of date or time.
`weekday`	Day of week.
`year`	Year of date.
`yeardays`	Number of days in year.

Date Conversion

`date2time`	Time and frequency from dates
`datedisp`	Display date entries.
`datenum`	Create date number.
`datestr`	Create date string.
`dec2thirtytwo`	Decimal quotation to thirty-second.
`m2xdate`	MATLAB serial date number to Excel serial date number.
`thirtytwo2dec`	Thirty-second quotation to decimal.
`time2date`	Dates from time and frequency.
`x2mdate`	Excel serial date number to MATLAB serial date number.

Financial Dates

`busdate`		Next or previous business day.
`busdays`		Business days in serial date format.
`datemnth`		Date of day in future or past month.
`datewrkdy`		Date of future or past workday.
`days360`	SIA[1]	Days between dates based on 360-day year.
`days360e`		Days between dates based on 360-day year (European).
`days360isda`	ISDA[2]	Days between dates based on 360-day year.
`days360psa`	PSA[3]	Days between dates based on 360-day year.
`days365`		Days between dates based on 365-day year.
`daysact`		Actual number of days between dates.
`daysadd`		Date away from starting date for any day-count basis
`daysdif`		Days between dates for any day-count basis.
`fbusdate`		First business date of month.

holidays	Holidays and non-trading days.
isbusday	True for dates that are business days.
lbusdate	Last business date of month.
thirdwednesday	Third Wednesday of month.
wrkdydif	Number of working days between dates.
yearfrac	Fraction of year between dates.

1. Securities Industry Association compliant

2. International Swap Dealer Association

3. Public Securities Association

Coupon Bond Dates

accrfrac	SIA	Fraction of coupon period before settlement.
cfamounts	SIA	Cash flow and time mapping for bond portfolio.
cfdates	SIA	Cash flow dates for fixed-income security with periodic payments.
cfport		Portfolio form of cash flow amounts.
cftimes	SIA	Time factors corresponding to bond cash flow dates.
cpncount	SIA	Coupon payments remaining until maturity.
cpndaten	SIA	Next coupon date after settlement date.
cpndatenq	SIA	Next quasi coupon date for fixed income security.
cpndatep	SIA	Previous coupon date before settlement date.
cpndatepq	SIA	Previous quasi coupon date for fixed income security.
cpndaysn	SIA	Number of days between settlement date and next coupon date.
cpndaysp	SIA	Number of days between previous coupon date and settlement date.
cpnpersz	SIA	Number of days in coupon period containing settlement date.

Formatting Currency and Price

`cur2frac`	Decimal currency value to fractional value.
`cur2str`	Bank-formatted text.
`dec2thirtytwo`	Decimal to thirty-second quotation
`frac2cur`	Fractional currency value to decimal value.
`thirtytwo2dec`	Thirty-second quotation to decimal

Charting Financial Data

`bar`	Bar chart
`bar3`	3-Dl bar chart
`bar3h`	3-D bar chart (horizontal)
`barh`	Bar chart (horizontal)
`bolling`	Bollinger band chart.
`candle`	Candlestick chart.
`candle`	Time series candle plot
`chartfts`	Interactive display
`dateaxis`	Convert serial-date axis labels to calendar-date axis labels.
`highlow`	High, low, open, close chart.
`highlow`	Time series High-Low plot
`movavg`	Leading and lagging moving averages chart.
`plot`	Plot data series
`pointfig`	Point and figure chart.

Analyzing and Computing Cash Flows

"Annuities"

"Amortization and Depreciation"

"Present Value"

"Future Value"

"Payment Calculations"

"Rates of Return"

"Cash Flow Sensitivities"

Annuities

`annurate`	Periodic interest rate of annuity.
`annuterm`	Number of periods to obtain value.

Amortization and Depreciation

`amortize`	Amortization.
`depfixdb`	Fixed declining-balance depreciation.
`depgendb`	General declining-balance depreciation.
`deprdv`	Remaining depreciable value.
`depsoyd`	Sum of years' digits depreciation.
`depstln`	Straight-line depreciation.

Present Value

`pvfix`	Present value with fixed periodic payments.
`pvvar`	Present value of varying cash flow.

Future Value

`fvdisc`	Future value of discounted security.
`fvfix`	Future value with fixed periodic payments.
`fvvar`	Future value of varying cash flow.

Payment Calculations

`payadv`	Periodic payment given number of advance payments.
`payodd`	Payment of loan or annuity with odd first period.
`payper`	Periodic payment of loan or annuity.
`payuni`	Uniform payment equal to varying cash flow.

Rates of Return

`effrr`	Effective rate of return.
`irr`	Internal rate of return.
`mirr`	Modified internal rate of return.
`nomrr`	Nominal rate of return.
`taxedrr`	After-tax rate of return.
`xirr`	Internal rate of return for nonperiodic cash flow.

Cash Flow Sensitivities

`cfconv`	Cash flow convexity.
`cfdur`	Cash flow duration and modified duration.

Fixed-Income Securities

"Accrued Interest"

"Prices"

"Term Structure of Interest Rates"

"Yields"

"Spreads"

"Interest Rate Sensitivities"

Accrued Interest

`acrubond`	Accrued interest of security with periodic interest payments.
`acrudisc`	Accrued interest of discount security paying at maturity.

Prices

`bndprice`	SIA	Price fixed income security from yield to maturity.
`prdisc`		Price of discounted security.
`prmat`		Price with interest at maturity.
`prtbill`		Price of Treasury bill.

Term Structure of Interest Rates

`disc2zero`	Zero curve given discount curve.
`fwd2zero`	Zero curve given forward curve.
`prbyzero`	Price bonds in portfolio by set of zero curves.
`pyld2zero`	Zero curve given par yield curve.
`tbl2bond`	Treasury bond parameters given Treasury bill parameters.
`tr2bonds`	Term-structure parameters given Treasury bond parameters.
`zbtprice`	Zero curve bootstrapping from coupon bond data given price.

zbtyield	Zero curve bootstrapping from coupon bond data given yield.
zero2disc	Discount curve given zero curve.
zero2fwd	Forward curve given zero curve.
zero2pyld	Par yield curve given zero curve.

Yields

beytbill		Bond equivalent yield for Treasury bill.
bndyield	SIA	Yield to maturity for fixed income security.
discrate		Bank discount rate of money market security.
ylddisc		Yield of discounted security.
yldmat		Yield of security with interest at maturity.
yldtbill		Yield of Treasury bill.

Spreads

bndspread	SIA	Static spread over spot curve

Interest Rate Sensitivities

bndconvp	SIA	Bond convexity given price.
bndconvy	SIA	Bond convexity given yield.
bnddurp	SIA	Bond duration given price.
bnddury	SIA	Bond duration given yield.

Analyzing Portfolios

Portfolio Analysis

abs2active	Convert constraints from absolute to active format
active2abs	Convert constraints from active to absolute format

`corr2cov`	Convert standard deviation and correlation to covariance.
`cov2corr`	Convert covariance to standard deviation and correlation coefficient.
`emaxdrawdown`	Expected maximum drawdown
`ewstats`	Expected return and covariance from return time series.
`frontcon`	Mean-variance efficient frontier.
`frontier`	Rolling efficient frontier
`holdings2weights`	Portfolio holdings into portfolio weights
`maxdrawdown`	Maximum drawdown
`pcalims`	Linear inequalities for individual asset allocation.
`pcgcomp`	Linear inequalities for asset group comparison constraints.
`pcglims`	Linear inequalities for asset group minimum and maximum allocation.
`pcpval`	Linear inequalities for fixing total portfolio value.
`periodicreturns`	Periodic total returns from daily total return prices
`portalloc`	Optimal capital allocation to efficient frontier portfolios.
`portcons`	Portfolio constraints.
`portopt`	Portfolios on constrained efficient frontier.
`portrand`	Randomized portfolio risks, returns, and weights.
`portstats`	Portfolio expected return and risk.
`portsim`	Monte Carlo simulation of correlated asset returns.
`portvrisk`	Portfolio value at risk
`ret2tick`	Convert return series to price series
`selectreturn`	Portfolio configurations from 3-D efficient frontier
`targetreturn`	Portfolio weight accuracy

`totalreturnprice`	Total return price time series
`tick2ret`	Convert price series to return series
`weights2holdings`	Portfolio values and weights into holdings

Financial Statistics

"Expectation Conditional Maximization"

"Multivariate Normal Regression"

"Expectation Conditional Maximization – Multivariate Normal Regression"

"Expectation Conditional Maximization – Least Squares Regression"

"Seemingly Unrelated Regression"

Expectation Conditional Maximization

`ecmnfish`	Fisher information matrix
`ecmnhess`	Hessian of negative log-likelihood function
`ecmninit`	Initial mean and covariance
`ecmnmle`	Mean and covariance of incomplete multivariate normal data
`ecmnobj`	Multivariate normal negative log-likelihood function
`ecmnstd`	Standard errors for mean and covariance of incomplete data

Multivariate Normal Regression

`mvnrfish`	Fisher information matrix for multivariate normal or least-squares regression
`mvnrmle`	Multivariate normal regression (ignore missing data)
`mvnrobj`	Log-likelihood function for multivariate normal regression without missing data
`mvnrstd`	Evaluate standard errors for multivariate normal regression model

Expectation Conditional Maximization – Multivariate Normal Regression

`ecmmvnrfish`	Fisher information matrix for multivariate normal regression model
`ecmmvnrmle`	Multivariate normal regression with missing data
`ecmmvnrobj`	Log-likelihood function for multivariate normal regression with missing data
`ecmmvnrstd`	Evaluate standard errors for multivariate normal regression model

Expectation Conditional Maximization – Least Squares Regression

`ecmlsrmle`	Least-squares regression with missing data
`ecmlsrobj`	Log-likelihood function for least-squares regression with missing data

Seemingly Unrelated Regression

`convert2sur`	Convert multivariate normal regression model to seemingly unrelated regression model

Pricing and Analyzing Derivatives

Option Valuation and Sensitivity

`binprice`	Binomial put and call pricing.
`blkimpv`	Implied volatility for futures options from Black's model.
`blkprice`	Black's model for pricing futures options.
`blsdelta`	Black-Scholes sensitivity to underlying price change.
`blsgamma`	Black-Scholes sensitivity to underlying delta change.
`blsimpv`	Black-Scholes implied volatility.
`blslambda`	Black-Scholes elasticity.
`blsprice`	Black-Scholes put and call pricing.

`blsrho`	Black-Scholes sensitivity to interest rate change.
`blstheta`	Black-Scholes sensitivity to time-until-maturity change.
`blsvega`	Black-Scholes sensitivity to underlying price volatility.
`opprofit`	Option profit.

GARCH Processes

The Financial Toolbox provides these representative functions to help you familiarize yourself with Generalized Autoregressive Conditional Heteroskedasticity (GARCH) in the MATLAB context. The GARCH Toolbox provides a more comprehensive and integrated computing environment that includes maximum likelihood parameter estimation, volatility forecasting, Monte Carlo simulation, diagnostic and hypothesis testing, graphical analysis, and data manipulation. For information see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod/`.

Univariate GARCH Processes

`ugarch`	GARCH parameter estimation.
`ugarchllf`	Log-likelihood objective function.
`ugarchpred`	Forecast conditional variance.
`ugarchsim`	Simulate GARCH process.

Financial Time Series Object and File Construction

`ascii2fts`	Create financial time-series object from ASCII data file
`fints`	Construct financial time-series object
`fts2ascii`	Write elements of time-series data into ASCII file
`fts2mat`	Convert to matrix
`merge`	Merge multiple financial time-series objects

Financial Time Series Arithmetic Functions

`end`	Last date entry
`horzcat`	Concatenate financial time series objects horizontally
`length`	Get number of dates (rows)
`minus`	Financial time series subtraction
`mrdivide`	Financial time series matrix division
`mtimes`	Financial time series matrix multiplication
`plus`	Financial time series addition
`power`	Financial time series power
`rdivide`	Financial time series division
`size`	Number of dates and data series
`subsasgn`	Content assignment
`subsref`	Subscripted reference
`times`	Financial time series multiplication
`uminus`	Unary minus of financial time series object
`uplus`	Unary plus of financial time series object
`vertcat`	Concatenate financial time series objects vertically

Financial Time Series Mathematical Functions

`cumsum`	Cumulative sum
`exp`	Exponential values
`hist`	Histogram
`log`	Natural logarithm
`log2`	Base 2 logarithm
`log10`	Common logarithm
`max`	Maximum value

`mean`	Arithmetic average
`min`	Minimum value
`std`	Standard deviation

Financial Time Series Utility Functions

`chfield`	Change data series name
`extfield`	Data series extraction
`fetch`	Data from financial time-series object
`fieldnames`	Get names of fields
`freqnum`	Convert string frequency indicator to numeric frequency indicator
`freqstr`	Convert numeric frequency indicator to string representation
`ftsbound`	Start and end dates
`ftsinfo`	Financial time series object information
`ftsuniq`	Determine uniqueness
`getfield`	Content of specific field
`getnameidx`	Find name in list
`iscompatible`	Structural equality
`isequal`	Multiple object equality
`isfield`	Check if string is field name
`issorted`	Check if dates and times are monotonically increasing
`rmfield`	Remove data series
`setfield`	Set content of specific field
`sortfts`	Sort financial time series

Financial Time Series Data Transformation Functions

boxcox	Box-Cox transformation
convert2sur	Convert multivariate normal regression model into SUR regression model
convertto	Convert to specified frequency
diff	Differencing
fillts	Fill missing values in time series
filter	Linear filtering
lagts	Lag time series object
leadts	Lead time series object
peravg	Periodic average
resamplets	Downsample data
smoothts	Smooth data
toannual	Convert to annual
todaily	Convert to daily
todecimal	Fractional to decimal conversion
tomonthly	Convert to monthly
toquarterly	Convert to quarterly
toquoted	Decimal to fractional conversion
tosemi	Convert to semiannual
toweekly	Convert to weekly
tsmovavg	Moving average

Financial Time Series Indicator Functions

adline	Accumulation/Distribution line
adosc	Accumulation/Distribution oscillator
bollinger	Bollinger band

chaikosc	Chaikin oscillator
chaikvolat	Chaikin volatility
fpctkd	Fast stochastic
hhigh	Highest high
llow	Lowest low
macd	Moving Average Convergence/Divergence (MACD)
medprice	Median price
negvolidx	Negative volume index
onbalvol	On-Balance Volume (OBV)
posvolidx	Positive volume index
prcroc	Price rate of change
pvtrend	Price and Volume Trend (PVT)
rsindex	Relative Strength Index (RSI)
spctkd	Slow stochastic
stochosc	Stochastic oscillator
tsaccel	Acceleration between periods
tsmom	Momentum between periods
typprice	Typical price
volroc	Volume rate of change
wclose	Weighted close
willad	Williams Accumulation/Distribution line
willpctr	Williams %R

Financial Time Series Graphical User Interface Function

ftsgui	Financial time series GUI

Functions — Alphabetical List

This section contains function reference pages listed alphabetically.

abs2active

Purpose

Convert constraints from absolute to active format

Syntax

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

Arguments

`AbsConSet` — Portfolio linear inequality constraint matrix expressed in absolute weight format. `AbsConSet` is formatted as `[A b]` such that `A*w <= 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 (e.g., a standard portfolio optimization imposes a sum-to-one budget constraint).

Description

`ActiveConSet = abs2active(AbsConSet, Index)` transforms a constraint matrix to an equivalent matrix expressed in active weight format (relative to the index). The transformation equation is

$$Aw_{absolute} = A(w_{active} + w_{index}) \le b_{absolute}$$

Therefore

$$Aw_{active} \le b_{absolute} - Aw_{index} = b_{active}$$

The initial constraint matrix consists of `NCONSTRAINTS` portfolio linear inequality constraints expressed in absolute weight format. The index portfolio vector contains `NASSETS` assets.

`ActiveConSet` is the transformed portfolio linear inequality constraint matrix expressed in active weight format, also of the form `[A b]` such that `A*w <= b`. The value `w` represents a vector of active asset weights (relative to the index portfolio) whose elements sum to zero.

See Also `active2abs`, `pcalims`, `pcgcomp`, `pcglims`, `pcpval`, `portcons`

Purpose Fraction of coupon period before settlement (SIA compliant)

Syntax

```
Fraction = accrfrac(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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.

Vector arguments must have consistent dimensions, or they must be scalars.

Description

`Fraction = accrfrac(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns the fraction of the coupon period before settlement. This function is used for computing accrued interest.

Examples

Given data for three bonds

```
Settle = '14-Mar-1997';
Maturity = ['30-Nov-2000'
            '31-Dec-2000'
            '31-Jan-2001'];
Period = 2;
Basis = 0;
EndMonthRule = 1;
```

Execute the function.

```
Fraction = accrfrac(Settle, Maturity, Period, Basis,...
                    EndMonthRule)
Fraction =
    0.5714
    0.4033
    0.2320
```

See Also

cfamounts, cfdates, cpncount, cpndaten, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpndaysp, cpnpersz

Purpose

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`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

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

```
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

Purpose 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 or equal to `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. 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).

Description `AccruInterest = acrudisc(Settle, Maturity, Face, Discount, Period, Basis)` returns the accrued interest of a discount security paid at maturity.

Examples

```
AccruInterest = acrudisc('05/01/1992', '07/15/1992', ...
                100, 0.1, 2, 0)

AccruInterest =
                2.0604  (or $2.06)
```

See Also `acrubond`, `prdisc`, `prmat`, `ylddisc`, `yldmat`

References Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula D.

active2abs

Purpose 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*w <= 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 (e.g., 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

$$Aw_{active} = A(w_{absolute} - w_{index}) \le b_{active}$$

Therefore

$$Aw_{absolute} \le b_{active} + Aw_{index} = b_{absolute}$$

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 <= 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

adline

Purpose

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

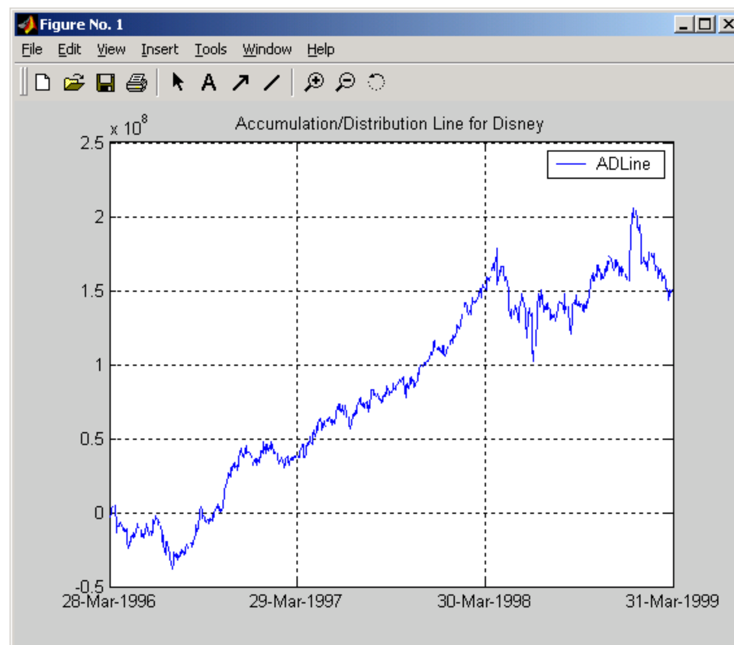
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

Parameter values are the strings that represent the valid parameter names.

Examples

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')
```



See Also

`adosc`, `willad`, `willpctr`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 56 - 58.

Purpose Accumulation/Distribution oscillator

Syntax

```
ado = adosc(highp, lowp, openp, closep)
ado = adosc([highp lowp openp closep])
adots = adosc(tsobj)
adots = adosc(tsojb, 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

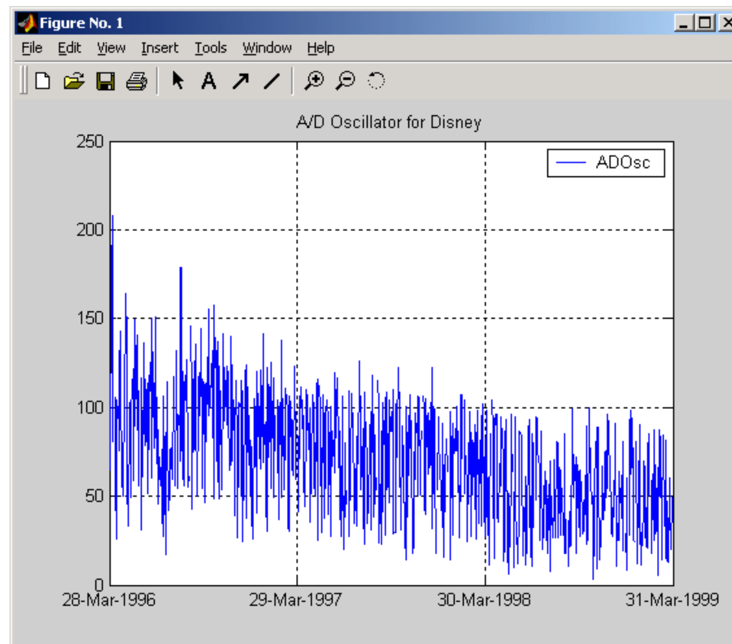
- `CloseName`: closing prices series name

Parameter values are the strings that represents the valid parameter names.

Examples

Compute 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')
```



See Also

`adline`, `willad`

Purpose 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. Assume a fixed rate of 12% APR and an initial loan amount of $100,000.

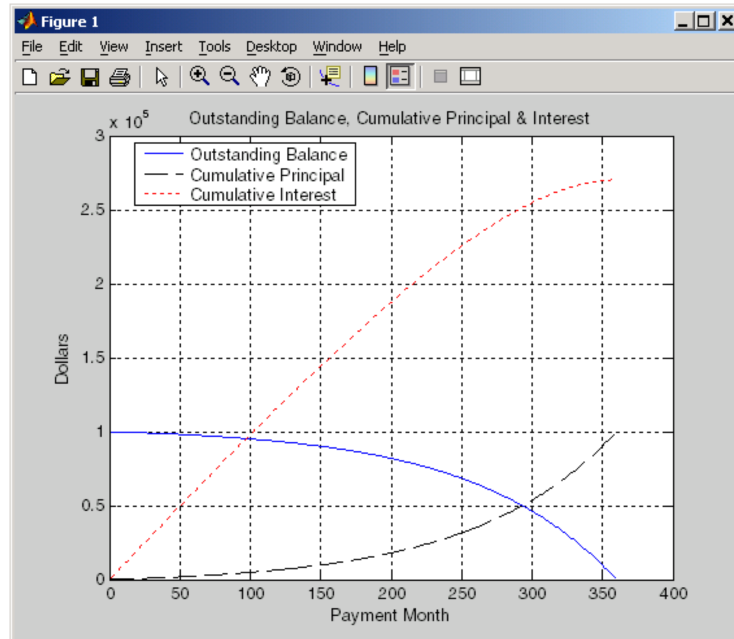
```
Rate        = 0.12/12;   % 12 percent APR = 1 percent per month
NumPeriods  = 30*12;     % 30 years = 360 months
PresentValue = 100000;

[Principal, Interest, Balance, Payment] = amortize(Rate,
NumPeriods, PresentValue);
```

The output argument Payment contains the fixed monthly payment.

```
format bank

Payment

Payment =

       1028.61
```

Finally, 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', 'TL')
```



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`, `annuterm`, `payadv`, `payodd`, `payper`

annurate

Purpose 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

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`

Purpose 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

A savings account has 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 =
            15.68  months or 1.31 years.
```

See Also

`annurate`, `amortize`, `fvfix`, `pvfix`

Purpose

Create financial time series object from ASCII data 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:
  - `'ddmmmyy'` or `'ddmmmyyyy'`
  - `'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.

`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.

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, e.g.,

```
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

Purpose Bar chart

Syntax

```
bar(tsobj)
bar(tsobj, width)
bar(..., 'style')
hbar = bar(...)

barh(...)
hbarh = barh(...)
```

Arguments

`tsobj`	Financial time series object
`width`	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.
`style`	`'grouped'` (default) or `'stacked'`

Description

`bar` and `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.

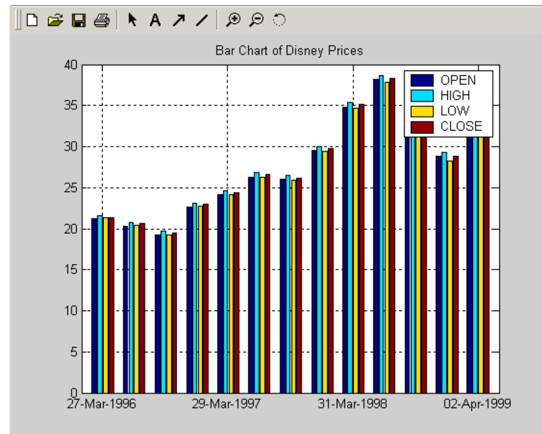
`bar(..., '`*`style`*`')` changes the style of the bar chart.

`hbar = bar(...)` returns a vector of bar handles.

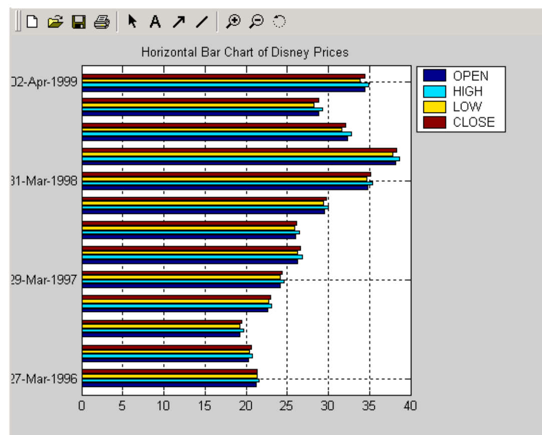
Use the MATLAB command `shading faceted` to put edges on the bars. Use `shading flat` to turn edges off.

Examples

Create bar charts for Disney stock showing high, low, opening, and closing prices.




```
load disney
bar(q_dis)
title('Bar Chart of Disney Prices')
```




```
load disney
bar(q_dis)
title('Horizontal Bar Chart of Disney Prices')
```

See Also `bar3`, `bar3h`, `candle`, `highlow`

bar3, bar3h

Purpose 3-D bar chart

Syntax

```
bar3(tsobj)
bar3(tsobj, width)
bar3(..., 'style')
hbar3 = bar3(...)

bar3h(...)
hbar3h = bar3h(...)
```

Arguments

`tsobj`	Financial time series object
`width`	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.
`style`	`'detached'` (default), `'grouped'`, or `'stacked'`

Description

`bar3` and `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.

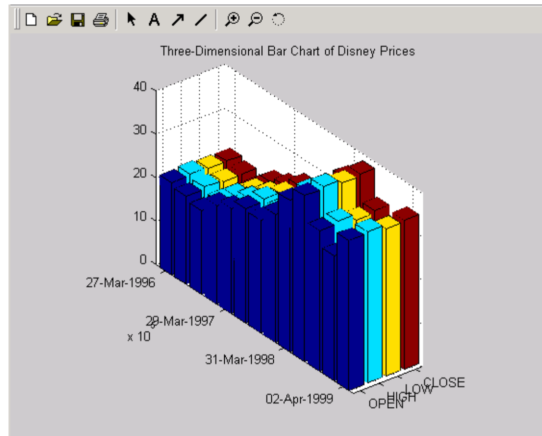
`bar3(..., 'style')` changes the style of the bar chart.

`hbar3 = bar3(...)` returns a vector of bar handles.

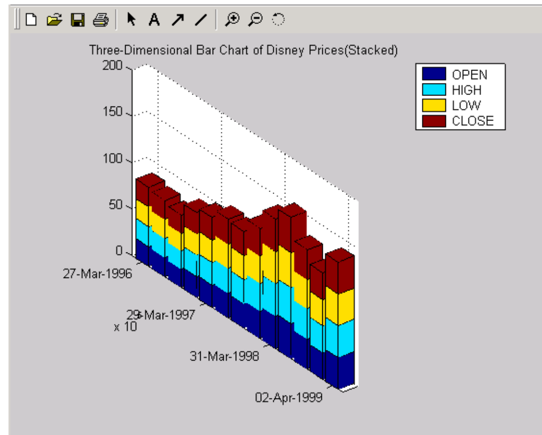
Use the MATLAB command `shading faceted` to put edges on the bars. Use `shading flat` to turn edges off.

Examples

Create three-dimensional bar charts for Disney stock showing high, low, opening, and closing prices.




```
load disney
bar3(q_dis, 'stacked')
title('Three-Dimensional Bar Chart of Disney Prices')
```




```
load disney
bar3(q_dis, 'stacked')
title('Three-Dimensional Bar Chart of Disney Prices (Stacked)')
```

See Also `bar`, `barh`, `candle`, `highlow`

beytbill

Purpose

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 or equal to `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

The settlement date of a Treasury bill is February 11, 2000, the maturity date is August 7, 2000, and the discount rate is 5.77%. The bond equivalent yield is

```
Yield = beytbill('2/11/2000', '8/7/2000', 0.0577)

Yield =
        0.0602
```

See Also

`datenum`, `prtbill`, `yldtbill`

Purpose

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 option using the Cox-Ross-Rubinstein binomial pricing model.

Examples

For a put option, the asset price is $52, option exercise price is $50, risk-free interest rate is 10%, option matures in 5 months, volatility is 40%, and there is one dividend payment of $2.06 in 3-1/2 months.

```
[Price, Option] = binprice(52, 50, 0.1, 5/12, 1/12, 0.4, 0, 0,...
2.06, 3.5)
```

returns the asset price and option value at each node of the binary tree.

```
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
```

See Also

`blkprice`, `blsprice`

References

Cox, J.; S. Ross; and M. Rubenstein, “Option Pricing: A Simplified Approach”, *Journal of Financial Economics* 7, Sept. 1979, pp. 229 - 263

Hull, *Options*, *Futures*, *and Other Derivative Securities*, 2nd edition, Chapter 14.

blkimpv

Purpose Implied volatility for futures options from Black's model

Syntax

```
Volatility = blsimpv(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 `Limit` 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 `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 = 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 non-scalar inputs must be identical.

`Rate` and `Time` must be expressed in consistent units of time.

Examples

Consider a European call futures option that expires in four months, trading at $1.1166, with 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)
```

See Also

`blkprice`, `blsimpv`, `blsprice`

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.

blkprice

Purpose Black's 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

Consider 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. Using this data

```
[Call, Put] = blkprice(20, 20, 0.09, 4/12, 0.25)
```

returns equal call and put prices of $1.1166.

See Also

`binprice`, `blsprice`

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.

blsdelta

Purpose

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.

Examples

```
[CallDelta, PutDelta] = blsdelta(50, 50, 0.1, 0.25, 0.3, 0)

CallDelta =
    0.5955

PutDelta =
   -0.4045
```

See Also `blsgamma`, `blslambda`, `blsprice`, `blsrho`, `blstheta`, `blsvega`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

blsgamma

Purpose 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.

Examples

```
Gamma = blsgamma(50, 50, 0.12, 0.25, 0.3, 0)

Gamma =
    0.0512
```

See Also `blsdelta`, `blslambda`, `blsprice`, `blsrho`, `blstheta`, `blsvega`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

blsimpv

Purpose

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

Consider 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)
```

See Also

`blsdelta`, `blsgamma`, `blslambda`, `blsprice`, `blsrho`, `blstheta`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

Luenberger, David G., *Investment Science*, Oxford University Press, 1998.

blslambda

Purpose

Black-Scholes elasticity

Syntax

```
[CallEl, PutEl] = blslambda(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] = blslambda(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.

Examples

```
[CallEl, PutEl] = blslambda(50, 50, 0.12, 0.25, 0.3)

CallEl =
    8.1274

PutEl =
   -8.6466
```

See Also `blsdelta`, `blsgamma`, `blsprice`, `blsrho`, `blstheta`, `blsvega`

References Daigler, *Advanced Options Trading*, Chapter 4.

blsprice

Purpose 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.

Examples Consider 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. Using this data

```
[Call, Put] = blsprice(100, 95, 0.1, 0.25, 0.5)
```

returns call and put prices of \$13.70 and \$6.35, respectively.

See Also

`blkprice`, `blsdelta`, `blsgamma`, `blsimpv`, `blslambda`, `blsrho`, `blstheta`, `blsvega`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

Luenberger, David G., *Investment Science*, Oxford University Press, 1998.

blsrho

Purpose 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, `Yield` could represent the dividend yield. For currency options, `Yield` 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.

Examples

```
[CallRho, PutRho] = blsrho(50, 50, 0.12, 0.25, 0.3, 0)

CallRho =
    6.6686

PutRho =
   -5.4619
```

See Also `blsdelta`, `blsgamma`, `blslambda`, `blsprice`, `blstheta`, `blsvega`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

blstheta

Purpose

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.

Examples

```
[CallTheta, PutTheta] = blstheta(50, 50, 0.12, 0.25, 0.3, 0)

CallTheta =
   -8.9630

PutTheta =
   -3.1404
```

See Also

`blsdelta`, `blsgamma`, `blslambda`, `blsprice`, `blsrho`, `blsvega`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

blsvega

Purpose

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, `Yield` could represent the dividend yield. For currency options, `Yield` 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.

Examples

```
Vega = blsvega(50, 50, 0.12, 0.25, 0.3, 0)

Vega =
    9.6035
```

See Also

`blsdelta`, `blsgamma`, `blslambda`, `blsprice`, `blsrho`, `blstheta`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003.

Purpose

Bond convexity given price (SIA compliant)

Syntax

```
[YearConvexity, PerConvexity] = bndconvp(Price, CouponRate, Settle,
   Maturity, Period, Basis, EndMonthRule, IssueDate,
   FirstCouponDate, LastCouponDate, StartDate, Face)
```

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
StartDate	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If StartDate is not explicitly specified, the effective start date is the settlement date.
Face	(Optional) Face or par value. Default = 100.

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 unspecified entries in input vectors with NaN. Dates can be serial date numbers or date strings.

Description

[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. This function determines the convexity for a bond whether or not the first or last coupon periods in the coupon structure are short or long (i.e., whether or not the coupon structure is synchronized to maturity). This function also determines the convexity of a zero coupon bond.

YearConvexity is the yearly (annualized) convexity; PerConvexity is the periodic convexity reported on a semiannual bond basis (in accordance with SIA convention). Both outputs are NUMBONDS-by-1 vectors.

Examples

Find 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
```

See Also

`bndconvy`, `bnddurp`, `bnddury`, `cfconv`, `cfdur`

bndconvy

Purpose

Bond convexity given yield (SIA compliant)

Syntax

```
[YearConvexity, PerConvexity] = bndconvy(Yield, CouponRate, Settle,
   Maturity, Period, Basis, EndMonthRule, IssueDate,
   FirstCouponDate, LastCouponDate, StartDate, Face)
```

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
StartDate	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If StartDate is not explicitly specified, the effective start date is the settlement date.
Face	(Optional) Face or par value. Default = 100.

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 unspecified entries in input vectors with NaN. Dates can be serial date numbers or date strings.

Description

[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. This function determines the convexity for a bond whether or not the first or last coupon periods in the coupon structure are short or long (i.e., whether or not the coupon structure is synchronized to maturity). This function also determines the convexity of a zero coupon bond.

YearConvexity is the yearly (annualized) convexity; PerConvexity is the periodic convexity reported on a semiannual bond basis (in accordance with SIA convention). Both outputs are NUMBONDS-by-1 vectors.

Examples

Find 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
```

See Also

bndconvp, bnddurp, bnddury, cfconv, cfdur

Purpose Bond duration given price (SIA compliant)

Syntax

```
[ModDuration, YearDuration, PerDuration] = bnddurp(Price,
   CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,
   IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)
```

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.
`Face`	(Optional) Face or par value. Default = 100.

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 unspecified entries in input vectors with `NaN`. Dates can be serial date numbers or date strings.

Description

```
[ModDuration, YearDuration, PerDuration] = bnddurp(Price,
CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,
IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)
```

computes the duration of `NUMBONDS` fixed income securities given a clean price for each bond. This function determines the Macaulay and modified duration for a bond whether or not the first or last coupon periods in the coupon structure are short or long (i.e., whether or not the coupon structure is synchronized to maturity). This function also determines the Macaulay and modified duration for a zero coupon bond.

`ModDuration` is the modified duration in years, reported on a semiannual bond basis (in accordance with SIA convention); `YearDuration` is the Macaulay duration in years; `PerDuration` is the periodic Macaulay duration reported on a semiannual bond basis (in accordance with SIA convention.) Outputs are `NUMBONDS`-by-1 vectors.

Examples

Find 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
```

See Also

bndconvp, bndconvy, bnddury

bnddury

Purpose Bond duration given yield (SIA compliant)

Syntax

```
[ModDuration, YearDuration, PerDuration] = bnddury(Yield,
   CouponRate, Settle, Maturity, Period, Basis, EndMonthRule,
   IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)
```

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
StartDate	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If StartDate is not explicitly specified, the effective start date is the settlement date.
Face	(Optional) Face or par value. Default = 100.

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 unspecified entries in input vectors with NaN. Dates can be serial date numbers or date strings.

Description

[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. This function determines the duration for a bond whether or not the first or last coupon periods in the coupon structure are short or long (i.e., whether or not the coupon structure is synchronized to maturity). This function also determines the Macaulay and modified duration for a zero coupon bond.

ModDuration is the modified duration in years, reported on a semiannual bond basis (in accordance with SIA convention); YearDuration is the Macaulay duration in years; PerDuration is the periodic Macaulay duration reported on a semiannual bond basis (in accordance with SIA convention). Outputs are NUMBONDS-by-1 vectors.

Examples

Find 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
```

See Also

bndconvp, bndconvy, bnddurp

Purpose Price fixed income security from yield to maturity (SIA compliant)

Syntax

```
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, Maturity)
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, Maturity,
   Period, Basis, EndMonthRule, IssueDate, FirstCouponDate,
   LastCouponDate, StartDate, Face)
```

Arguments Required and optional inputs can be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalar arguments. Optional inputs can also be passed as empty matrices (`[]`) or omitted at the end of the argument list. The value `NaN` in any optional input invokes the default value for that entry. Dates can be serial date numbers or date strings.

`Yield`	Bond 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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.
`Face`	(Optional) Face or par value. Default = 100.

Description

`[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, returns the clean prices and accrued interest due.

`Price` is the clean price of the bond (current price without accrued interest).

`AccruedInt` is the accrued interest payable at settlement.

`Price` and `Yield` are related by the formula

```
Price + Accrued_Interest = sum(Cash_Flow*(1+Yield/2)^(-Time))
```

where the sum is over the bonds' cash flows and corresponding times in units of semiannual coupon periods.

Examples

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
```

See Also

`cfamounts`, `bndyield`

bndspread

Purpose Static spread over spot curve

Syntax

```
Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity,
   Period, Basis, EndMonthRule, IssueDate, FirstCouponDate,
   LastCouponDate)
```

Arguments

`SpotInfo`	Two-column matrix: `[SpotDates ZeroRates]` Zero rates correspond to maturities on the spot dates, continuously compounded. You will obtain the best results if you choose evenly spaced rates close together, for example, by using the three-month deposit rates.
`Price`	Price for every $100 notional amount of bonds whose spreads are computed.
`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 or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A scalar or 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. 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).

`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.

Description

`Spread = bndspread(SpotInfo, Price, Coupon, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` computes the static spread to benchmark in basis points.

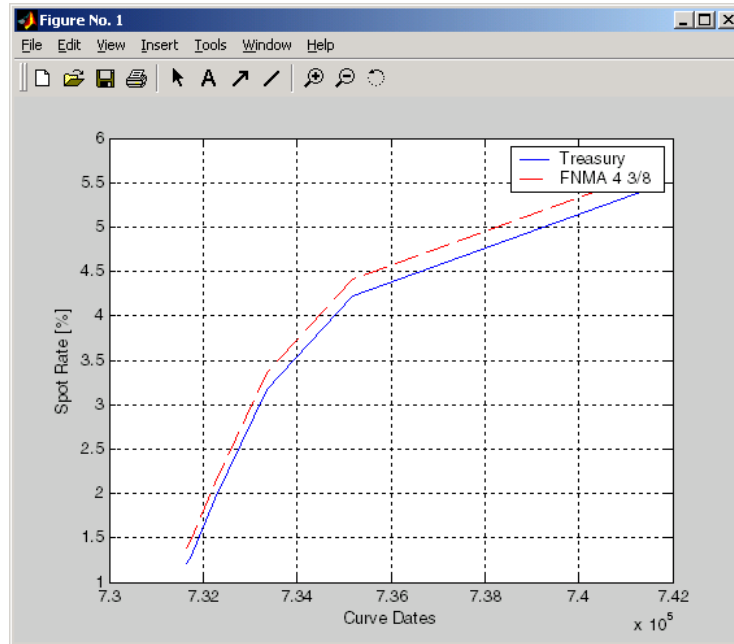
Examples

Compute a FNMA 4 3/8 spread over a Treasury spot-curve.

```
% Build spot curve.

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, Nov 26, 2002
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 =

   18.7582
```



See Also `bndprice`, `bndyield`

Purpose Yield to maturity for fixed income security (SIA compliant)

Syntax

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

Arguments Required and optional inputs can be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalar arguments. Optional inputs can also be passed as empty matrices (`[]`) or omitted at the end of the argument list. The value `NaN` in any optional input invokes the default value for that entry. Dates can be serial date numbers or date strings.

`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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.
`Face`	(Optional) Face or par value. Default = 100.

Description

`Yield = bndyield(Price, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` given `NUMBONDS` bonds with SIA date parameters and clean prices (excludes accrued interest), returns the bond equivalent yields to maturity.

`Yield` is a `NUMBONDS`-by-1 vector of the bond equivalent yields to maturity with semiannual compounding.

`Price` and `Yield` are related by the formula

```
Price + Accrued_Interest = sum(Cash_Flow*(1+Yield/2)^(-Time))
```

where the sum is over the bonds' cash flows and corresponding times in units of semiannual coupon periods.

Examples

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
```

See Also

`bndprice`, `cfamounts`

bolling

Purpose Bollinger band chart

Syntax

```
bolling(Asset, Samples, Alpha)
[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`, `highlow`, `movavg`, `pointfig`

bollinger

Purpose 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, upper, and lower 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 default window size of 20. `uppr` and `lowr` are vectors that represent the upper and lower bands. These bands are +2 times and -2 times moving standard deviations away from the middle band.

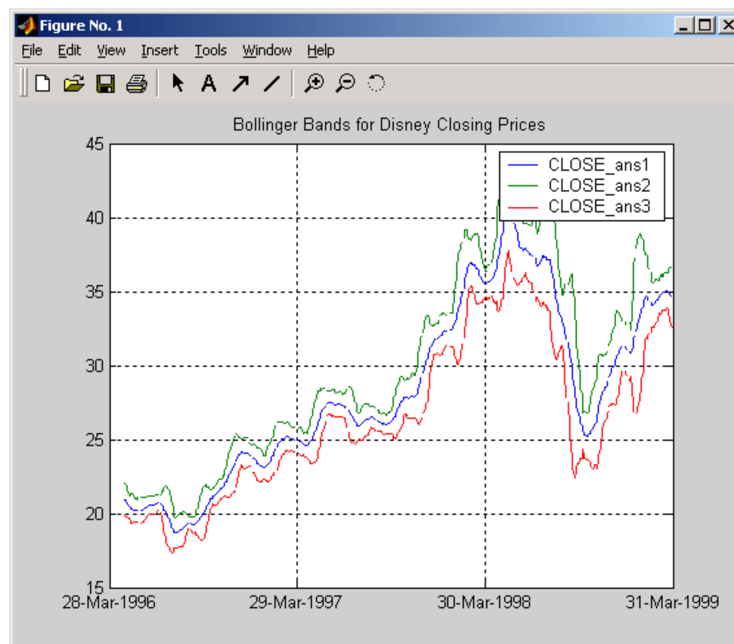
`[midfts, upprfts, lowrfts] = bollinger(tsobj, wsize, wts, nstd)` calculates the middle, upper, and lower bands that make up the Bollinger bands from a financial time series object `tsobj`.

`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 +2 times and -2 times moving standard deviations away from the middle band.

Examples

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')
```



See Also

`tsmovavg`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 72 - 74.

boxcox

Purpose Box-Cox transformation

Syntax

```
[transdat, lambda] = boxcox(data)
[transfts, lambdas] = boxcox(tsobj)
transdat = boxcox(lambda, data)
transfts = boxcox(lambda, tsobj)
```

Arguments

`data`	Data vector. Must be positive.
`tsobj`	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 defined by

$$data(\lambda) = \begin{cases} \dfrac{data^{\lambda} - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(data) & \text{if } \lambda = 1 \end{cases}$$

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 calculates the transformation parameter λ.

`[transfts, lambda] = boxcox(tsojb)` transforms the financial time series object `tsobj` using the Box-Cox transformation method into `transfts`. It also calculates 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, e.g., 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, tsobj)` 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

Purpose

Next or previous business day

Syntax

```
Busday = busdate(Date, Direction, Holiday, Weekend)
```

Arguments

`Date`	Reference date. Enter as serial date number or date string.
`Direction`	(Optional) Direction. 1 = next (default) or -1 = previous business day.
`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 serial 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), `Weekend = [1 0 0 0 0 0 1]`.

Description

`Busday = busdate(Date, Direction, Holiday, Weekend)` returns the serial date number of the next or previous business day from the reference date.

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

Purpose 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`	(Optional) Frequency of business days:
	`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`
	Strings must be enclosed in single quotation marks.

`holvec`	(Optional) Holiday dates vector in string or serial date format

Description

`bdates = busdays(sdate, edate, bdmode)` generates a vector of business days, `bdates`, in serial date format between the start date, `sdate`, and end date, `edate`, with frequency, `bdmode`. The dates are generated based on United States holidays. If you do not supply `bdmode`, `busdays` generates a daily vector.

If the ending date, `edate`, is not the last business day of the specified frequency, the next business date in line is returned.

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.

If you want a weekday vector without the holidays, set `holvec` to `''` (empty string) or `[]` (empty vector).

candle

Purpose Candlestick chart

Syntax

```
candle(High, Low, Close, Open, Color)
```

Arguments

`High`	High prices for a security. A column vector.
`Low`	Low prices for a security. An column vector.
`Close`	Closing prices for a security. A column vector.
`Open`	Opening prices for a security. A column vector.
`Color`	(Optional) Candlestick color. A string. MATLAB 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.

Description

`candle(High, Low, Close, Open, Color)` 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.

Examples

Given `High`, `Low`, `Close`, and `Open` as equal-size vectors of stock price data

```
candle(High, Low, Close, Open, 'cyan')
```

plots a candlestick chart with cyan candles.

See Also

`bolling`, `dateaxis`, `highlow`, `movavg`, `pointfig`

Purpose

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

`tsobj`	Financial time series object
`color`	(Optional) A three-element row vector representing RGB or a color identifier. (See `plot` in the MATLAB documentation.)
`dateform`	(Optional) Date string format used as the *x*-axis tick labels. (See `datetick` in the MATLAB documentation.) You can specify a `dateform` only when `tsobj` does not contain time-of-day data. If `tsobj` contains time-of-day data, `dateform` 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

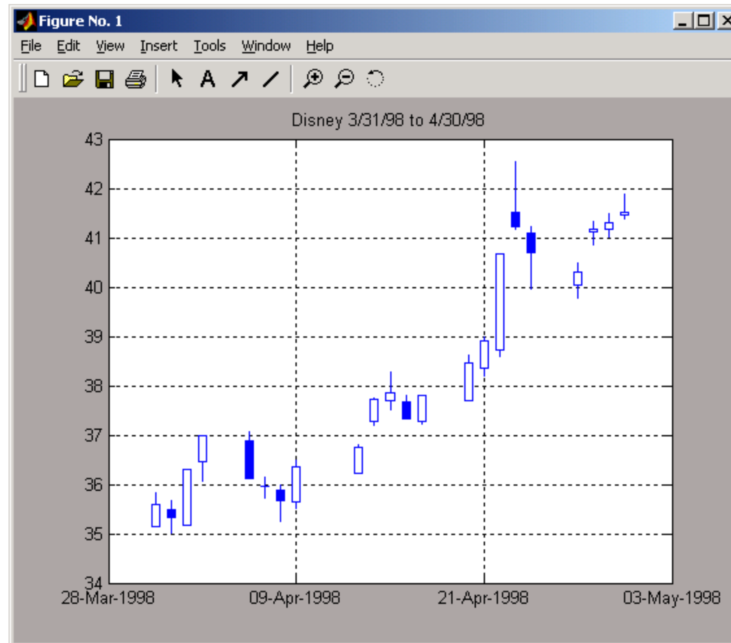
- `CloseName`: closing prices series name

`hcdl = candle(tsobj, color, dateform, ParameterName, ParameterValue, ...)` returns the handle to the patch objects and the line object that make up the candle plot. `hdcl` 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 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

`chartfts`, `highlow`, `plot`

Purpose

Cash flow and time mapping for bond portfolio (SIA compliant)

Syntax

```
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] =
   cfamounts(CouponRate, Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
   StartDate, Face)
```

Arguments

`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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
StartDate	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If StartDate is not explicitly specified, the effective start date is the settlement date.
Face	(Optional) Face or par value. Default = 100.

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

```
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] =
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. The elements contained in the 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 is zero in the case of all zero coupon bonds. This function 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.

CFlowAmounts is the 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` is the 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` is the 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 are useful in determining the present value of a stream of cash flows. The term "time factor" refers to the exponent *TF* in the discounting equation

$$PV = CF/(1+z/2)^{TF}$$

where:

PV =	present value of a cash flow
CF =	the cash flow amount
z =	the risk-adjusted annualized rate or yield corresponding to given cash flow. The yield is quoted on a semiannual basis.
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, we use SIA actual/actual day count conventions for all time factor calculations.

`CFlowFlags` is the 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 (e.g., nominal coupon cash flow, front or end partial or "stub" coupon, maturity cash flow). Possible values are shown in the table.

Flag	Cash Flow Type
0	Accrued interest due on a bond at settlement.
1	Initial cash flow amount smaller than normal due to "stub" coupon period. A stub period is created when the time from issue date to first coupon is shorter than normal.

Flag	Cash Flow Type
2	Larger than normal initial cash flow amount because 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 abnormally short and actual maturity cash flow is smaller than normal).
6	Larger than normal maturity cash flow because 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 bond has less than one coupon period to maturity.
9	Larger than normal maturity cash flow when bond has less than one coupon period to maturity.
10	Maturity cash flow on a zero coupon bond.

Examples

Consider a portfolio containing a corporate bond paying interest quarterly and a treasury bond paying interest semiannually. Compute the cash flow structure and the time factors for each bond.

```
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
```

See Also

accrfrac, cfdates, cpncount, cpndaten, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpndaysp, cpnpersz

cfconv

Purpose 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 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 (periods)
```

See Also `bndconvp`. `bndconvy`, `bnddurp`, `bnddury`, `cfdur`

Purpose

Cash flow dates for fixed-income security (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement 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 `NaN`s 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

Purpose Cash-flow duration and modified duration

Syntax

```
[Duration, ModDuration] = cfdur(CashFlow, Yield)
```

Arguments

`CashFlow`	A vector of real numbers.
`Yield`	Periodic yield. A scalar. Enter as a decimal fraction.

Description

`[Duration, ModDuration] = cfdur(CashFlow, Yield)` calculates the duration and modified duration of a cash flow in periods.

Examples

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 (periods)

ModDuration =
              8.7521 (periods)
```

See Also

`bndconvp`, `bndconvy`, `bnddurp`, `bnddury`, `cfconv`

Purpose

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 `NaN`s.
`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.

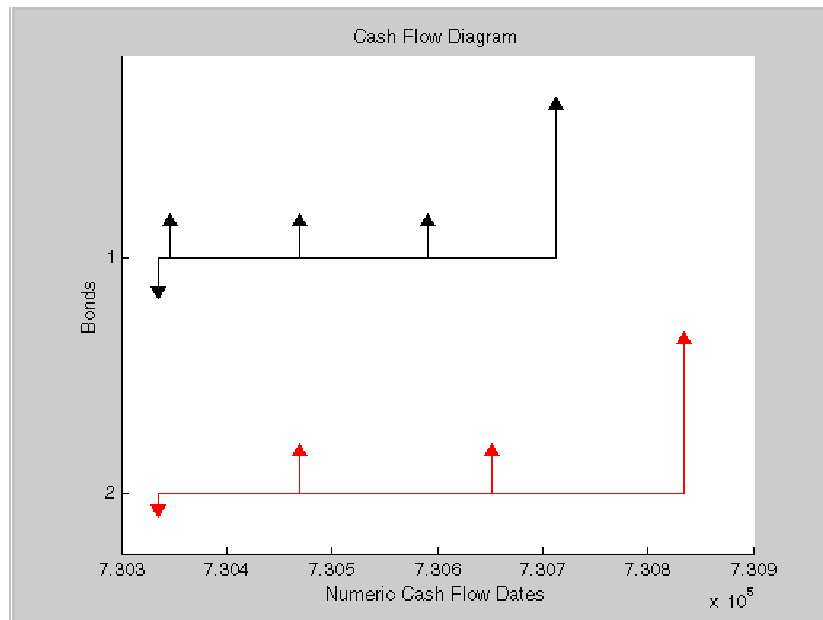
`AllTF` is a `NUMDATES`-by-1 list of time factors corresponding to the dates in `AllDates`. If `TFactors` is not entered, `AllTF` contains the number of days from the first date in `AllDates`.

`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 `NaN`s.

Examples

Use `cfamounts` to calculate the cash flow amounts, cash flow dates, and time factors for each of two bonds. Then use `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')
```



Finally, call `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

Purpose

Time factors corresponding to bond cash flow dates (SIA compliant)

Syntax

```
TFactors = cftimes(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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.

Description

`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. The time factor of a cash flow is the difference between the settlement date and the cash flow date in units of semiannual coupon periods. In computing time factors, we use SIA actual/actual day count conventions for all time factor calculations.

Examples

```
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
```

See Also

`accrfrac`, `cfdates`, `cfamounts`, `cpncount`, `cpndaten`, `cpndatenq`, `cpndatep`, `cpndatepq`, `cpndaysn`, `cpndaysp`, `cpnpersz`, `date2time`

chaikosc

Purpose 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 prices that must be included are the high (`highp`), low (`lowp`), and closing (`closep`) prices.

`chosc = chaikosc([highp lowp closep tvolume])` accepts a four-column matrix as input.

`choscts = chaikosc(tsobj)` calculates the Chaikin Oscillator, `choscts`, from the data contained in the financial time series object `tsobj`. `tsobj` 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 `tsobj` but only one series named `ChaikOsc`.

`choscts = chaikosc(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

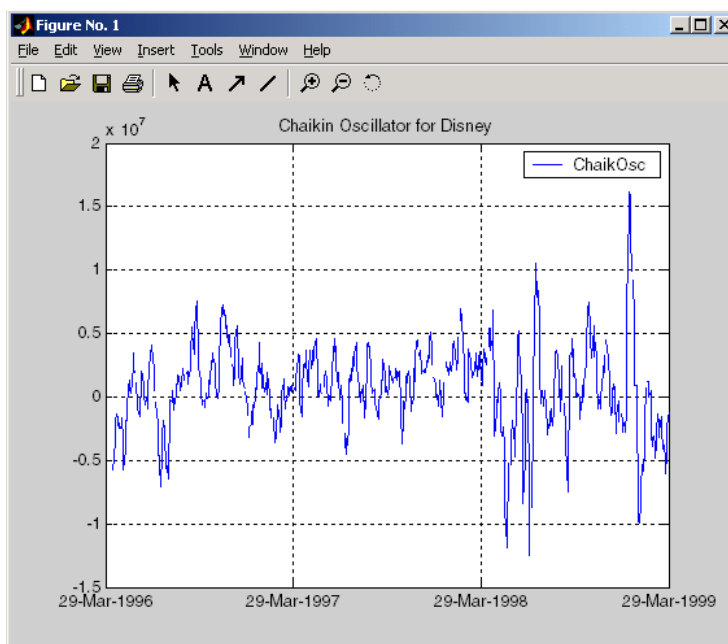
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

Parameter values are the strings that represent the valid parameter names.

Examples

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')
```



See Also

`adline`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 91 - 94.

chaikvolat

Purpose 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

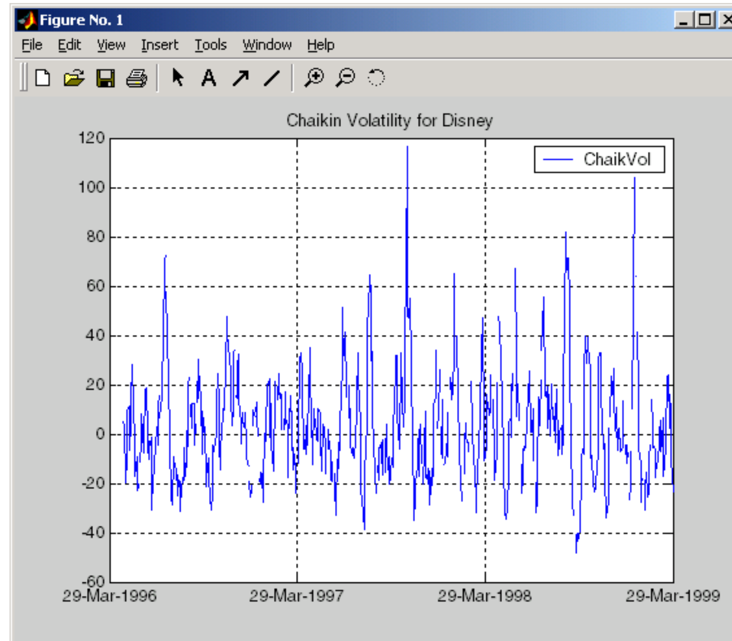
Parameter values are the strings that represent the valid parameter names.

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 for Disney stock and plot the results:

```
load disney.mat
dis_CHAIKvol = chaikvolat(dis)
plot(dis_CHAIKvol)
title('Chaikin Volatility for Disney')
```



See Also

`chaikosc`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 304 - 305.

Purpose

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

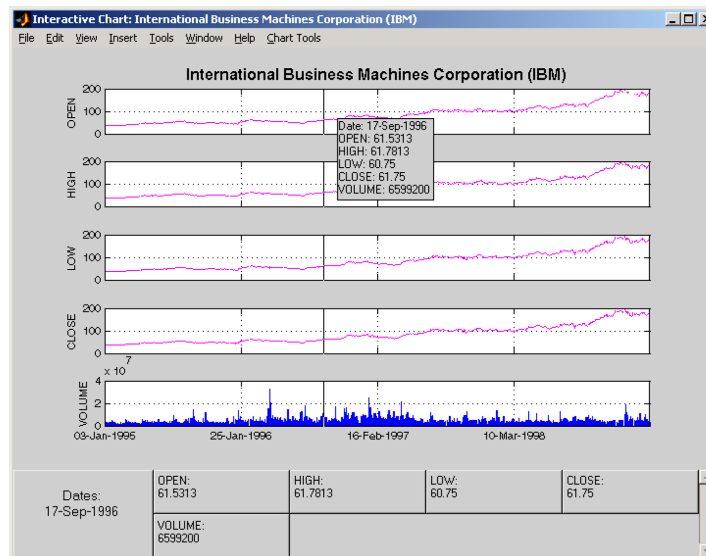
Create a financial time series object from the supplied data file `ibm9599.dat`:

```
ibmfts = ascii2fts('ibm9599.dat', 1, 3, 2);
```

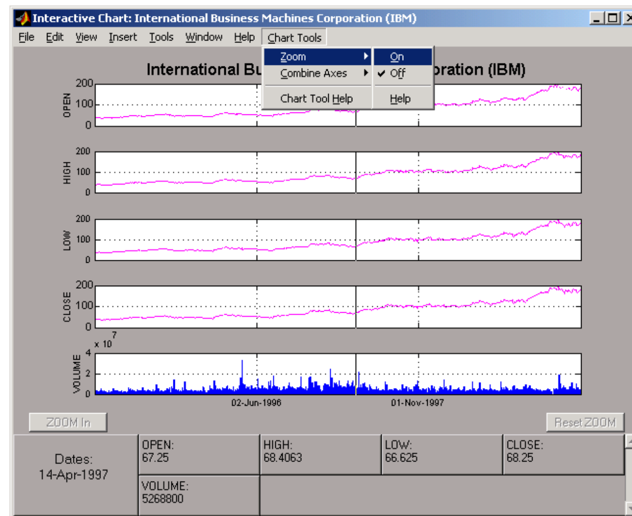
Chart the financial time series object `ibmfts`:

```
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 the section “Visualizing Financial Time Series Objects” on page 6-17. See “Zoom Tool” on page 6-20 for details on performing the zoom. Also see “Combine Axes Tool” on page 6-23 for information about combining axes for specified plots.

See Also

`candle`, `highlow`, `plot`

Purpose 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`

Purpose

Convert multivariate normal regression model to seemingly unrelated regression (SUR) model

Syntax

```
DesignSUR = convert2sur(Design, Group)
```

Arguments

`Design` — A matrix or a cell array that depends on the number of data series `NUMSERIES`.

- If `NUMSERIES = 1`, `convert2sur` simply returns the `Design` matrix.
- If `NUMSERIES > 1`, `Design` is a cell array with `NUMSAMPLES` cells, where each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

`Group` — Contains information about how data series are to be grouped, with separate parameters for each group. Specify groups either by series or by groups:

- To identify groups by series, construct an index vector that has `NUMSERIES` elements. Element `i = 1, ..., NUMSERIES` in the vector, and has the index `j = 1, ..., NUMGROUPS` of the group in which series *i* is a member.
- To identify groups by groups, construct a cell array with `NUMGROUPS` elements. Each cell contains a vector with the indexes of the series that populate a given group.

In either case, the number of series is `NUMSERIES` and the number of groups is `NUMGROUPS`, with `1 ≤ NUMGROUPS ≤ NUMSERIES`.

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 upon the value of `NUMSERIES`:

- If `NUMSERIES = 1`, `DesignSUR = Design`, 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.

Example

This example has ten series in three groups, and two model parameters. Suppose

Group 1 has series 1, 3, 4, 8

Group 2 has series 2, 6, 10

Group 3 has series 5, 7, 9

Either:

```
Group = [ 1, 2, 1, 1, 3, 2, 3, 1, 3, 2];
```

or

```
Group = cell(3,1);
Group{1} = [1, 3, 4, 8];
Group{2} = [2, 6, 10];
Group{3} = [5, 7, 9];
```

A regression with `DesignSUR` would have 3 x 2 = 6 model parameters.

convertto

Purpose Convert to specified frequency

Syntax

```
newfts = convertto(oldfts, newfreq)
```

Arguments

`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. It makes some assumptions regarding the dates in the resulting time series.

`newfts = convertto(oldfts, newfreq)` converts the object `oldfts` to the new time series object `newfts` with the frequency `newfreq`.

See Also

`toannual`, `todaily`, `tomonthly`, `toquarterly`, `tosemi`, `toweekly`

Purpose

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

```
ExpSigma = [0.5  2.0];

ExpCorrC = [1.0 -0.5
           -0.5  1.0];

ExpCovariance = corr2cov(ExpSigma, ExpCorrC)
```

Expected results:

```
ExpCovariance =

     0.2500   -0.5000
    -0.5000    4.0000
```

See Also

`corrcoef`, `cov`, `cov2corr`, `ewstats`, `std`

cov2corr

Purpose Convert covariance to standard deviation and correlation coefficient

Syntax

```
[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)
```

Arguments

`ExpCovariance` n-by-n covariance matrix, e.g., from `cov` or `ewstats`. n 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

```
ExpCovariance = [0.25 -0.5
                 -0.5   4.0];

[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)
```

Expected results:

```
  ExpSigma =

    0.5000    2.0000

ExpCorrC =

    1.0000   -0.5000
   -0.5000    1.0000
```

See Also

`corr2cov`, `corrcoef`, `cov`, `ewstats`, `std`

Purpose

Coupon payments remaining until maturity (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement 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.

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.

Examples

```
NumCouponsRemaining = cpncount('14 Mar 1997', '30 Nov 2000',...
2, 0, 0)

n =
     8
```

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

cpndaten

Purpose Next coupon date for fixed-income security (SIA compliant)

Syntax

```
NextCouponDate = cpndaten(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 or equal to `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. 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).
`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. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure.
LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow 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.

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

Purpose Next quasi coupon date for fixed income security (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow 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. 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

Given a pair of bonds with the 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`, `cpnpersz`

Purpose Previous coupon date for fixed-income security (SIA compliant)

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 or equal to `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. 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).
`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. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure.
LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow 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.

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, cpndatenq, cpndatepq, cpndaysn, cpndaysp, cpnpersz

cpndatepq

Purpose Previous quasi coupon date for fixed income security (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow 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. 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

Given a pair of bonds with the 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 prior to 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, cpndatenq, cpndatep, cpndaysn, cpndaysp, cpnpersz

cpndaysn

Purpose Number of days to next coupon date (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement 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.

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, cpndatenq, cpndatep, cpndatepq, cpndaysp, cpnpersz

Purpose

Number of days since previous coupon date (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.

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, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpnpersz

cpnpersz

Purpose Number of days in coupon period (SIA compliant)

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 or equal to `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. 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).
`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. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. 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 will be followed only by the bond's maturity cash flow date.
`StartDate`	(Future implementation; optional) Date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward-starting, specify this date as a future date. If `StartDate` is not explicitly specified, the effective start date is the settlement date.

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 = cpnpersz(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 = cpnpersz('14 Sep 2000', '30 Jun 2001', 2, 0, 0)

NumDaysPeriod =

   183

NumDaysPeriod = cpnpersz('14 Sep 2000', '30 Jun 2001', 2, 0, 1)

NumDaysPeriod =

   184
```

```
Maturity = ['30 Apr 2001'; '31 May 2001'; '30 Jun 2001'];

NumDaysPeriod = cpnpersz('14 Sep 2000', Maturity)

NumDaysPeriod =

   184
   183
   184
```

See Also accrfrac, cfamounts, cfdates, cpncount, cpndaten, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpndaysp

Purpose Cumulative sum

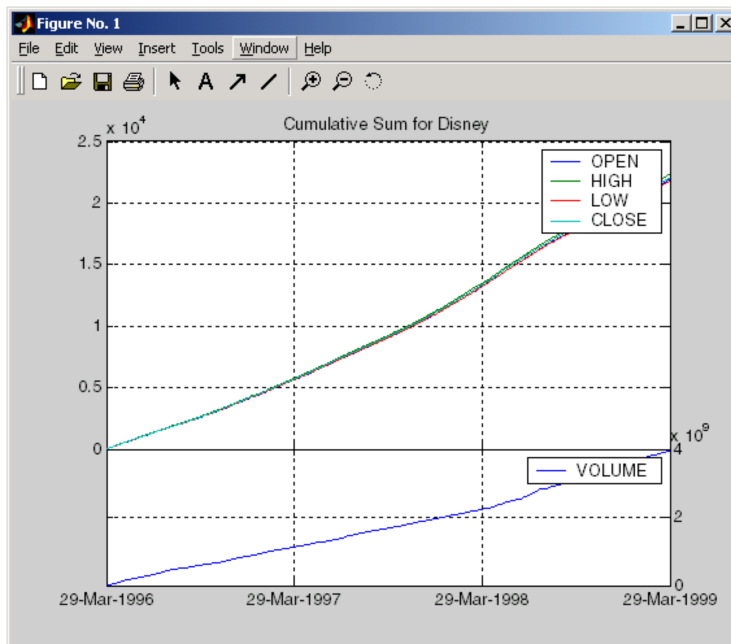
Syntax

```
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 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` in the MATLAB documentation

cur2frac

Purpose 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

```
Fraction = cur2frac(12.125, 8)
```

returns `Fraction = 12.1`, a string.

See Also `cur2str`, `frac2cur`

Purpose 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

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

returns `String = ($8264.00)`

See Also `cur2frac`, `frac2cur`

date2time

Purpose Time and frequency from dates

Syntax

```
[TFactors, F] = date2time(Settle, Dates, Compounding, Basis,
   EndMonthRule)
```

Arguments

`Settle` Settlement date. A vector of serial date numbers or date strings.

`Dates` A vector of dates corresponding to the compounding value.

`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` = 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, e.g. `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`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`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, Dates, Compounding, Basis, EndMonthRule)` computes time factors appropriate to compounded rate quotes beyond the settlement date.

`TFactors` is a vector of time factors.

`F` is a scalar of related compounding frequencies.

`date2time` is the inverse of `time2date`.

See Also

`cftimes`, `disc2rate`, `rate2disc`, `time2date`

dateaxis

Purpose Convert serial-date axis labels to calendar-date axis labels

Syntax

```
dateaxis(Aksis, DateForm, StartDate)
```

Arguments

`Aksis`	(Optional) Determines which axis tick labels—*x*, *y*, or *z*—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(Aksis, 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

DateForm	Format	Description
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

Purpose

Display date entries

Syntax

```
datedisp(NumMat, DateForm)
CharMat = datedisp(NumMat, DateForm)
```

Arguments

`NumMat`	Numeric matrix to display
`DateForm`	(Optional) Date format. See `datestr` 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

```
NumMat = [730730, 0.03, 1200 730100;
          730731, 0.05, 1000 NaN]

NumMat =

   1.0e+05 *

    7.3073    0.0000    0.0120    7.3010
    7.3073    0.0000    0.0100       NaN

datedisp(NumMat)

01-Sep-2000   0.03   1200   11-Dec-1998
02-Sep-2000   0.05   1000      NaN
```

See Also

`datestr`

datefind

Purpose

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

```
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`

Purpose

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`	(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. This flag has no effect if `EndMonthRule` is set to 1.
`Basis`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(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) 0 = rule off (default), meaning that the rule is not in effect.

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`

Purpose Create date number

Syntax

```
DateNumber = datenum(DateString)
DateNumber = datenum(DateString, Pivot)
DateNumber = datenum(Year, Month, Day)
DateNumber = datenum(Year, Month, Day, Hour, Minute, Second)
```

Description

`DateNumber = datenum(DateString)` returns a serial date number given a date string. Date numbers are the number of days that has passed since a base date. *In MATLAB, date number 1 is January 1, 0000 A.D.* If the input includes time components, the date number includes a fractional component.

The date string can be any of several forms.

`'19-may-1999'`
`'may 19, 1999'`
`'19-may-99'`
`'19-may'` (current year assumed)
`'5/19/99'`
`'5/19'` (current year assumed)
`'19-may-1999, 18:37'`
`'19-may-1999, 6:37 pm'`
`'5/19/99/18:37'`
`'5/19/99/6:37 pm'`

Certain formats may not contain enough information to compute a date number. In these cases, missing values default to 0 for hours, minutes, and seconds; January for the month; and 1 for the day of month. The year defaults to the current year. Unless you specify a pivot year, date strings with two-character years, e.g., `12-june-12`, are assumed to lie within the 100-year period centered about the current year.

`DateNumber = datenum(DateString, Pivot)` assumes that two-character years lie within the 100-year period beginning with the pivot year. The default pivot year is the current year minus 50 years.

`DateNumber = datenum(Year, Month, Day)` returns a serial date number given year, month, and day integers.

datenum

`DateNumber = datenum(Year, Month, Day, Hour, Minute, Second)` returns a serial date number given year, month, day, hour, minute, and second integers.

Note This function now ships with basic MATLAB. It originally shipped only with the Financial Toolbox. This description remains here for your convenience.

Examples

```
DateNumber = datenum('19-may-1999')
DateNumber = 730259

DateNumber = datenum('5/19/99')
DateNumber = 730259

DateNumber = datenum('19-may-1999, 6:37 pm')
DateNumber = 730259.78

DateNumber = datenum('5/19/99/18:37')
DateNumber = 730259.78

DateNumber = datenum(1999, 5, 19)
DateNumber = 730259

DateNumber = datenum(1999, 1:6, 19)
DateNumber = [730139  730170  730198  730229  730259  730290]

DateNumber = datenum(1999, 5, 19, 18, 37, 0)
DateNumber = 730259.78

DateNumber = datenum(730259)
DateNumber = 730259
```

The next example demonstrates the use of the pivot year in interpreting date strings with two-character years.

```
DateNumber = datenum('12-june-12 )
DateNumber =
       735032
```

```
datestr(735032)
ans =
12-Jun-2012

DateNumber = datenum('12-june-12 ,1900)
DateNumber =
       698507
datestr(698507)
ans =
12-Jun-1912
```

See Also

datedisp, datestr, datevec, daysact, now, today

datestr

Purpose Create date string

Syntax

```
DateString = datestr(Date, DateForm)
DateString = datestr(Date, DateForm, Pivot)
DateString = datestr(Date)
```

Description

`DateString = datestr(Date, DateForm)` converts a date number or a date string to a date string. `DateForm` specifies the format of `DateString`. Date strings with two-character years, e.g., `12-june-12`, are assumed to lie within the 100-year period centered about the current year.

`DateString = datestr(Date, DateForm, Pivot)` assumes that two-character years lie within the 100-year period beginning with the pivot year. The default pivot year is the current year minus 50 years.

Note MATLAB internal date handling and calculations generate no ambiguous values. However, whenever possible, programmers should use date strings containing four-digit years or serial date numbers.

`DateString = datestr(Date)` assumes `DateForm` is `1`, `16`, or `0` depending on whether the date number `Date` contains a date, time, or both, respectively. If `Date` is a date string, the function assumes `DateForm` is `1`.

DateForm	Format	Example
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01

DateForm	Format	Example
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1 01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd.yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd.yyyy'	Mar.01.2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY	Q1-2001
28	'mmmyyyy'	Mar2000

Note This function now ships with basic MATLAB. It originally shipped only with the Financial Toolbox. This description remains here for your convenience.

Examples

```
DateString = datestr(730123, 1)
DateString = 03-Jan-1999

DateString = datestr(730123, 2)
DateString = 01/03/99

DateString = datestr(730123, 12)
DateString = Jan99

DateString = datestr(730123.776, 0)
DateString = 03-Jan-1999 18:37:26

DateString = datestr('1/03', 1) (assuming the current year is 1999)
DateString = 03-Jan-1999

DateString = datestr(730123)
DateString = 03-Jan-1999

DateString = datestr([730123 730154 730182 730213 730243 730274])
DateString =
03-Jan-1999
03-Feb-1999
03-Mar-1999
03-Apr-1999
03-May-1999
03-Jun-1999

DateString = datestr('1/03')
DateString = 03-Jan-1999 (assuming the current year is 1999)
```

See Also

`dateaxis`, `datedisp`, `datenum`, `datevec`, `daysact`, `now`, `today`

Purpose Date components

Syntax

```
DateVector = datevec(Date)
DateVector = datevec(Date, Pivot)
[Year, Month, Day, Hour, Minute, Second] = datevec(Date)
```

Description

`DateVector = datevec(Date)` converts a date number or a date string to a date vector whose elements are `[Year Month Day Hour Minute Second]`. The first five elements are integers, the sixth is a floating-point number. Date strings with two-character years, e.g., `12-june-12`, are assumed to lie within the 100-year period centered about the current year.

`DateVector = datevec(Date, Pivot)` assumes that two-character years lie within the 100-year period beginning with the pivot year. The default pivot year is the current year minus 50 years.

Note MATLAB internal date handling and calculations generate no ambiguous values. However, whenever possible, programmers should use date strings containing four-digit years or serial date numbers.

`[Year, Month, Day, Hour, Minute, Second] = datevec(Date)` converts a date number or a date string to a date vector and returns the components of the date vector as individual variables.

Note This function now ships with basic MATLAB. It originally shipped only with the Financial Toolbox. This description remains here for your convenience.

Examples

```
DateVec = datevec('28-Jul-00')
DateVec =
        2000     7     28     0     0     0

DateVec = datevec(730695)
DateVec =
        2000     7     28     0     0     0
```

```
DateVec = datevec(730695.776)

DateVec =
        2000     7     28     18     37     26.4

[Year, Month, Day, Hour, Minute, Second] = datevec(730695.776)

Year =
  2000

Month =
     7

Day =
    28

Hour =
    18

Minute =
    37

Second =
    26.4

[Year, Month, Day] = datevec(730695:730697)

Year =
    2000    2000    2000

Month =
       7       7       7

Day =
      28      29      30
```

See Also

datenum, datestr, now, today

datewrkdy

Purpose 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

Purpose Day of month

Syntax `DayMonth = day(Date)`

Description `DayMonth = day(Date)` returns the day of the month given a serial date number or date string.

Examples

```
DayMonth = day(730544)
```

or

```
DayMonth = day('2/28/00')
```

returns `DayMonth = 28`

See Also `datevec`, `eomday`, `month`, `year`

days360

Purpose

Days between dates based on 360-day year (SIA compliant)

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 (i.e., 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
```

See Also

`days365`, `daysact`, `daysdif`, `wrkdydif`, `yearfrac`

References

Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures,* Vol. 2, Spring 1995.

Purpose

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 (i.e., 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

Purpose Days between dates based on 360-day year (ISDA)

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 (i.e., 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

Purpose Days between dates based on 360-day year (PSA)

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 (i.e., 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

Purpose

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

Purpose 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, 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

Purpose

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`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

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
```

See Also `daysdif`

References

Stigum, Marcia L. and Franklin Robinson, *Money Market and Bond Calculations*, Richard D. Irwin, 1996, ISBN 1-55623-476-7

daysdif

Purpose

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`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

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.

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
```

See Also

`datenum`, `days360`, `days365`, `daysact`, `daysadd`, `wrkdydif`, `yearfrac`

References

Stigum, Marcia L. and Franklin Robinson, *Money Market and Bond Calculations*, Richard D. Irwin, 1996, ISBN 1-55623-476-7

dec2thirtytwo

Purpose 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

Two bonds are quoted with decimal prices of 101.78 and 102.96. Convert these prices 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`

Purpose Fixed declining-balance depreciation schedule

Syntax

```
Depreciation = depfixdb(Cost, Salvage, Life, Period, Month)
```

Arguments

`Cost`	Initial value of the asset.
`Salvage`	Salvage value of the asset.
`Life`	Life of the asset in years.
`Period`	Number of years to calculate.
`Month`	(Optional) Number of months in the first year of asset life. Default = 12.

Description

`Depreciation = depfixdb(Cost, Salvage, Life, Period, Month)` calculates the fixed declining-balance depreciation for each period.

Examples

A car is purchased for $11,000 with a salvage value of $1500 and a lifetime of eight years. To calculate the depreciation for the first five years

```
Depreciation = depfixdb(11000, 1500, 8, 5)
```

returns

```
Depreciation =
      2425.08     1890.44     1473.67     1148.78     895.52
```

See Also

`depgendb`, `deprdv`, `depsoyd`, `depstln`

depgendb

Purpose 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 A car is purchased for $11,000 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.

```
Depreciation = depgendb(11000, 1000, 5, 2)
```

returns

```
Depreciation =
        4400.00     2640.00     1584.00     950.40     425.60
```

See Also `depfixdb`, `deprdv`, `depsoyd`, `depstln`

deprdv

Purpose 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 The cost of an asset is $13,000 with a life of 10 years. The salvage value is $1000. First find the accumulated depreciation with the straight-line depreciation function, `depstln`. Then find the remaining depreciable value after six years.

```
Accum = depstln(13000, 1000, 10) * 6

Accum =
       7200.00

Value = deprdv(13000, 1000, 7200)

Value =
       4800.00
```

See Also `depfixdb`, `depgendb`, `depsoyd`, `depstln`

depsoyd

Purpose 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

The cost of an asset is $13,000 with a life of 10 years. The salvage value of the asset is $1000.

```
Sum = depsoyd(13000, 1000, 10)'
```

returns

```
Sum =
       2181.82
       1963.64
       1745.45
       1527.27
       1309.09
       1090.91
        872.73
        654.55
        436.36
        218.18
```

See Also

`depfixdb`, `depgendb`, `deprdv`, `depstln`

Purpose 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 The cost of an asset is $13,000 with a life of 10 years. The salvage value of the asset is $1000.

```
Depreciation = depstln(13000, 1000, 10)
```

returns

```
Depreciation =
              1200
```

See Also `depfixdb`, `depgendb`, `deprdv`, `depsoyd`

diff

Purpose 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` in the MATLAB documentation

disc2zero

Purpose

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. 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).

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

Given 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 = [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');
```

Set daily compounding for the output zero curve, on an actual/365 basis.

```
Compounding = 365;
Basis = 3;
```

Execute the function

```
[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates,...
Settle, Compounding, Basis)
```

which returns the zero curve `ZeroRates` at the maturity dates `CurveDates`.

```
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 computed them at full precision. If you enter `DiscRates` as shown, `ZeroRates` may differ due to rounding.

See Also

`zero2disc` and other functions for Term Structure of Interest Rates

discrate

Purpose 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 or equal to `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. 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).

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 (e.g., U. S. Treasury Bills) and understates the true yield earned by investors.

Examples

```
DiscRate = discrate('12-jan-2000', '25-jun-2000', 100, 97.74, 0)
```

returns

```
DiscRate =

    0.0501
```

a discount rate of 5.01%.

See Also `acrudisc`, `fvdisc`, `prdisc`, `ylddisc`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 1.

Purpose

Least-squares regression with missing data

Syntax

```
[Parameters, Covariance, Resid, Info] = ecmlsrmle(Data, Design,
   MaxIterations, TolParam, TolObj, Param0, Covar0)
```

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 mvnrmle.)

`Design` — A matrix or a cell array that handles two model structures:

- If NUMSERIES = 1, Design is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series.
- If NUMSERIES ≥ 1, Design is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values.

If Design has a single cell, it is assumed to have the same Design matrix for each sample. If Design has more than one cell, each cell contains a Design matrix for each sample.

`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 sqrt(eps) which is about 1.0e-8 for double precision. The convergence test for changes in model parameters is

$$\|\texttt{Param}_k - \texttt{Param}_{k-1}\| < \texttt{TolParam} \times (1 + \|\texttt{Param}_k\|)$$

where Param represents the output Parameters, and iteration $k = 2, 3, \ldots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both TolParam ≤ 0 and TolObj ≤ 0, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.

`TolObj`	(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 $\left\| \mathtt{Obj}_k - \mathtt{Obj}_{k-1} \right\| < \mathtt{TolObj} \times (1 + \left\| \mathtt{Obj}_k \right\|)$ for iteration $k = 2, 3, \ldots$. Convergence is assumed when both the `TolParam` and `TolObj` conditions are satisfied. If both `TolParam ≤ 0` and `TolObj ≤ 0`, do the maximum number of iterations (`MaxIterations`), whatever the results of the convergence tests.
`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`	(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. 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.

Description

`[Parameters, Covariance, Resid, Info] = ecmlsrmle(Data, Design, MaxIterations, TolParam, TolObj, Param0, Covar0)` estimates a least-squares regression model with missing data. The model has the form

$$\mathtt{Data}_k \sim N(\mathtt{Design}_k \times \mathtt{Parameters}, \mathtt{Covariance})$$

for samples $k = 1, \ldots,$ `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.

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`.
- `ecmlsrmle` is more strict than `mvnrmle` about the presence of `NaN` values in the `Design` array.

Use the estimates in the optional output structure `Info` for diagnostic purposes.

See Also

`ecmlsrobj`, `ecmmvnrmle`, `mvnrmle`

References

[1] Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[2] Xiao-Li Meng and Donald B. Rubin, "Maximum Likelihood Estimation via the ECM Algorithm," *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

[3] Joe Sexton and Anders Rygh Swensen, "ECM Algorithms that Converge at the Rate of EM," *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

[4] 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.

Purpose 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 `NaN`s. Only samples that are entirely `NaN`s are ignored. (To ignore samples with at least one `NaN`, use `mvnrmle`.)
`Design`	A matrix or a cell array that handles two model structures: • If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series. • If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values. If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.
`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` $\geq$ `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.

See Also

`ecmlsrmle`, `mvnrmle`, `mvnrobj`

Purpose

Fisher information matrix for multivariate normal regression model

Syntax

```
Fisher = ecmmvnrfish(Data, Design, Covariance, Method, MatrixFormat)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `NaN`s. Only samples that are entirely `NaN`s are ignored. (To ignore samples with at least one `NaN`, use `mvnrfish`.)

`Design` — A matrix or a cell array that handles two model structures:

- If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series.
- If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.

`Covariance` — `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the residuals of the regression.

`Method`	(Optional) String that identifies method of calculation for the information matrix: • `hessian` - 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. • `fisher` - Use the Fisher information matrix.
`MatrixFormat`	(Optional) String that identifies parameters to be included in the Fisher information matrix: • `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.

Description

`Fisher = ecmmvnrfish(Data, Design, Covariance, Method, MatrixFormat)` 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'`,

```
NUMPARAMS = NUMSERIES * (NUMSERIES + 3)/2
```

If `MatrixFormat = 'paramonly'`,

```
NUMPARAMS = NUMSERIES
```

Note `ecmmvnrfish` operates slowly if you calculate the full Fisher information matrix.

See Also `ecmnstd`, `ecmnmle`

ecmmvnrmle

Purpose

Multivariate normal regression with missing data

Syntax

```
[Parameters, Covariance, Resid, Info] = ecmmvnrmle(Data, Design,
   MaxIterations, TolParam, TolObj, Param0, Covar0)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `NaN`s. Only samples that are entirely `NaN`s are ignored. (To ignore samples with at least one `NaN`, use `mvnrmle`.)

`Design` — A matrix or a cell array that handles two model structures:

- If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series.
- If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.

`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 `sqrt(eps)` which is about 1.0e-8 for double precision. The convergence test for changes in model parameters is

$$\|\mathtt{Param}_k - \mathtt{Param}_{k-1}\| < \mathtt{TolParam} \times (1 + \|\mathtt{Param}_k\|)$$

where `Param` represents the output `Parameters`, and iteration $k = 2, 3, \ldots$. Convergence is assumed when both the `TolParam` and `TolObj` conditions are satisfied. If both `TolParam ≤ 0` and `TolObj ≤ 0`, do the maximum number of iterations (`MaxIterations`), whatever the results of the convergence tests.

`TolObj`	(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 $\lvert \mathtt{Obj}_k - \mathtt{Obj}_{k-1} \rvert < \mathtt{TolObj} \times (1 + \lvert \mathtt{Obj}_k \rvert)$ for iteration $k = 2, 3, \ldots$. Convergence is assumed when both the `TolParam` and `TolObj` conditions are satisfied. If both `TolParam ≤ 0` and `TolObj ≤ 0`, do the maximum number of iterations (`MaxIterations`), whatever the results of the convergence tests.
`Param0`	(Optional) `NUMPARAMS`-by-`1` column vector that contains a user-supplied initial estimate for the parameters of the regression model.
`Covar0`	(Optional) `NUMSERIES`-by-`NUMSERIES` matrix that contains a user-supplied initial or known estimate for the covariance matrix of the regression residuals.

Description

[Parameters, Covariance, Resid, Info] = ecmmvnrmle(Data, Design, MaxIterations, TolParam, TolObj, Param0, Covar0) estimates a multivariate normal regression model with missing data. The model has the form

$$\texttt{Data}_k \sim N(\texttt{Design}_k \times \texttt{Parameters, Covariance})$$

for samples k = 1, ... , NUMSAMPLES.

ecmmvnrmle estimates a NUMPARAMS-by-1 column vector of model parameters called Parameters, and a NUMSERIES-by-NUMSERIES matrix of covariance parameters called Covariance.

ecmmvnrmle(Data, Design) with no output arguments plots the log-likelihood function for each iteration of the algorithm.

To summarize the outputs of ecmmvnrmle:

- 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. WARNING: 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

`ecmmvnrmle` 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`.
- `ecmmvnrmle` is more strict than `mvnrmle` about the presence of `NaN` values in the `Design` array.

Use the estimates in the optional output structure `Info` for diagnostic purposes.

See Also

`ecmmvnrobj`, `mvnrmle`

References

[1] Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[2] Xiao-Li Meng and Donald B. Rubin, "Maximum Likelihood Estimation via the ECM Algorithm," *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

[3] Joe Sexton and Anders Rygh Swensen, "ECM Algorithms that Converge at the Rate of EM," *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

[4] 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.

ecmmvnrobj

Purpose

Log-likelihood function for multivariate normal regression with missing data

Syntax

```
Objective = ecmmvnrobj(Data, Design, Parameters, Covariance)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `NaN`s. Only samples that are entirely `NaN`s are ignored. (To ignore samples with at least one `NaN`, use `mvnrmle`.)

`Design` — A matrix or a cell array that handles two model structures:

- If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series.
- If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.

`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.

Description

`Objective = ecmmvnrobj(Data, Design, Parameters, Covariance)` 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.

See Also

ecmmvnrmle, mvnrmle, mvnrobj

ecmmvnrstd

Purpose

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters, StdCovariance] = ecmmvnrstd(Data, Design,
   Covariance, Method)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `NaN`s. Only samples that are entirely `NaN`s are ignored. (To ignore samples with at least one `NaN`, use `mvnrstd`.)

`Design` — A matrix or a cell array that handles two model structures:

- If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series.
- If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.

`Covariance` — `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the regression residuals.

`Method` — (Optional) String that identifies method of calculation for the information matrix:

- `hessian` - 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.
- `fisher` - Use the Fisher information matrix.

Description

`[StdParameters, StdCovariance] = ecmmvnrstd(Data, Design, Covariance, Method)` evaluates standard errors for a multivariate normal regression model with missing data. The model has the form

$$\mathtt{Data}_k \sim N(\mathtt{Design}_k \times \mathtt{Parameters}, \mathtt{Covariance})$$

for samples k = 1, ... , `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.

See Also

`mvnrmle`, `ecmmvnrmle`, `ecmmvnrstd`

References

[1] Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

ecmnfish

Purpose 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: `inv(Covariance)`
`MatrixFormat`	(Optional) String that identifies parameters included in the Fisher information matrix. If `MatrixFormat = []` or `''`, the default method `full` is used. The parameter choices are • `full` — (Default) Compute full Fisher information matrix. • `meanonly` — 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

```
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 `NaN`s 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 with standard errors calculated with ecmnhess.

See Also

ecmnhess, ecmnmle

ecmnhess

Purpose Hessian of negative log-likelihood function

Syntax

```
Hessian = ecmnhess(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: `inv(Covariance)`
`MatrixFormat`	(Optional) String that identifies parameters included in the Hessian matrix. If `MatrixFormat = []` or `''`, the default method `full` is used. The parameter choices are • `full` — (Default) Compute full Hessian matrix. • `meanonly` — Compute only components of the Hessian matrix associated with the mean.

Description

`Hessian = ecmnhess(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 with standard errors calculated from ecmnfish.

See Also

ecmnfish, ecmnmle

ecmninit

Purpose

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 InitMethod = [] or '', the default method nanskip is used. The initialization methods are • nanskip — (Default) Skip all records with NaNs. • twostage — Estimate mean. Fill NaNs with the mean. Then estimate the covariance. • diagonal — Form a diagonal covariance.

Description

[Mean, Covariance] = ecmninit(Data, InitMethod) creates initial mean and covariance estimates for the function ecmnmle. 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.

Algorithm

Model

The general model is

$$Z \sim N(Mean, Covariance)$$

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 `NaN`s 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 `NaN`s. 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

`ecmnmle`

ecmnmle

Purpose Mean and covariance of incomplete multivariate normal data

Syntax

```
[Mean, Covariance] = ecmnmle(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 `NaN`s. A sample is also called an *observation* or a *record*.

`InitMethod` — (Optional) String that identifies one of three defined initialization methods to compute initial estimates for the mean and covariance of the data. If `InitMethod` = `[]` or `''`, the default method `nanskip` is used. The initialization methods are

- `nanskip` — (Default) Skip all records with `NaN`s.
- `twostage` — Estimate mean. Fill `NaN`s with mean. Then estimate covariance.
- `diagonal` — Form a diagonal covariance.

Note If you supply `Mean0` and `Covar0`, `InitMethod` is not executed.

`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 `Tolerance` $\leq$ 0, perform maximum iterations specified by `MaxIterations` and do not evaluate the objective function at each step unless in display mode, as described below.

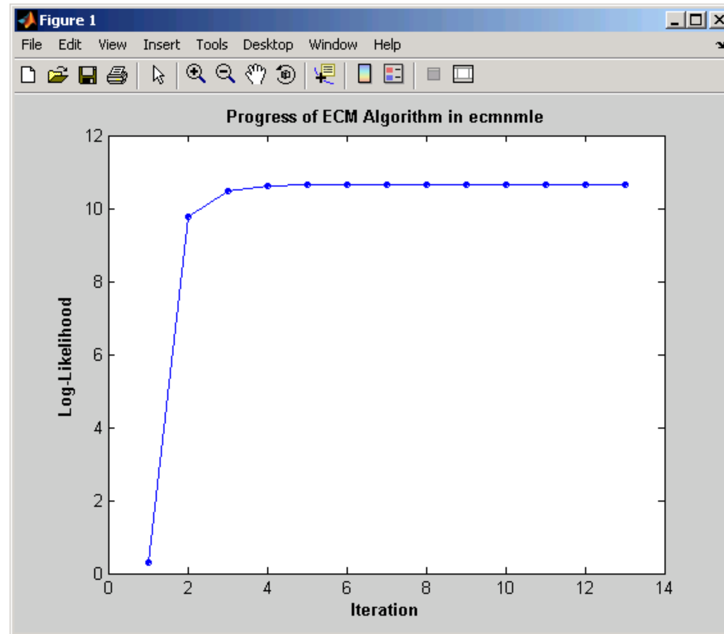
`Mean0`	(Optional) Initial `NUMSERIES`-by-`1` column vector estimate for the mean. If you leave `Mean0` unspecified (`[]`), the method specified by `InitMethod` is used. If you specify `Mean0`, you must also specify `Covar0`.
`Covar0`	(Optional) Initial `NUMSERIES`-by-`NUMSERIES` matrix estimate for 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] = ecmnmle(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 `ecmnmle`, run the demo program `ecmguidemo`.

Algorithm

Model

The general model is

$$Z \sim N(Mean, Covariance)$$

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 ≤ `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 `NaN`s 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 `NaN`s. 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.

See Also

`ecmnfish`, `ecmnhess`, `ecmninit`, `ecmnobj`, `ecmnstd`

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.

ecmnobj

Purpose

Multivariate normal negative log-likelihood function

Syntax

```
Objective = ecmnobj(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 = ecmnobj(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 `NaN`s 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`

Purpose

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 `NaN`s.
`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 • `hessian` — (Default) Hessian of the observed negative log-likelihood function. • `fisher` — 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 `ecmnmle`. 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

`ecmnmle`

Purpose 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^Rate-1)`.

Examples

Find the effective annual rate of return based on an annual percentage rate of 9% compounded monthly.

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

returns

```
Return =

        0.0938
```

or 9.38%

See Also

`nomrr`

emaxdrawdown

Purpose

Expected maximum drawdown

Syntax

```
ExpDrawdown = 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

`ExpDrawdown = emaxdrawdown(Mu, Sigma, T)` computes the expected maximum drawdown for a Brownian motion for each time period in `T` using the equation

$$dX(t) = \mu dt + \sigma dW(t).$$

If the Brownian motion is geometric with the stochastic differential equation

$$dS(t) = \mu_0 S(t)dt + \sigma_0 S(t)dW(t)$$

use Ito's lemma with $X(t) = \log(S(t))$ such that

$$\mu = \mu_0 - 0.5\sigma_0^2$$
$$\sigma = \sigma_0$$

to convert 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.

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.

See Also

`maxdrawdown`

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.

end

Purpose 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`, `subsref`

`end` in the MATLAB documentation

eomdate

Purpose Last date of month

Syntax

```
DayMonth = eomdate(Year, Month)
```

Description `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 to 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.

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`

eomday

Purpose Last day of month

Syntax

```
Day = eomday(Year, Month)
```

Description

`Day = eomday(Year, Month)` returns the last day of the month for the given year and month. Enter `Year` as a four-digit integer; enter `Month` as an integer from `1` to `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. `Day` is then a 1-by-n vector of days.

Note This function now ships with basic MATLAB. It originally shipped only with the Financial Toolbox. This description remains here for your convenience.

Examples

```
Day = eomday(2000, 2)

Day =

    29
```

See Also

`day`, `eomdate`, `month`

ewstats

Purpose 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 < DecayFactor <= 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.

`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-DecayFactor^WindowLength)/(1-DecayFactor)`.

A smaller `DecayFactor` or `WindowLength` emphasizes recent data more strongly but uses less of the available data set.

Examples

```
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

Purpose 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`

Purpose Data series extraction

Syntax

```
ftse = extfield(tsobj, fieldnames)
```

Arguments

`tsobj`	Financial time series object
`fieldnames`	Data series to be extracted. A cell array if a list of data series names (`fieldnames`) is supplied. A string if only one is wanted.

Description `ftse = extfield(tsobj, fieldnames)` extracts from `tsobj` the dates and data series specified by `fieldnames` into a new financial time series object `ftse`. `ftse` has all the dates in `tsobj` 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`

fbusdate

Purpose 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

Example 1:

```
Date = fbusdate(2001, 11); datestr(Date)
ans =
01-Nov-2001

Year = [2002 2003 2004];
Date = fbusdate(Year, 11); datestr(Date)

ans =
```

```
01-Nov-2002
03-Nov-2003
01-Nov-2004
```

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];
```

March 1, 2003, is a Saturday. Use `fbusdate` to check that this Saturday is actually the first business day of the month.

```
Date = datestr(fbusdate(2003, 3, [], Weekend))

Date =

01-Mar-2003
```

See Also

`busdate`, `eomdate`, `holidays`, `isbusday`, `lbusdate`

fetch

Purpose Data from financial time-series object

Syntax

```
newfts = fetch(oldfts, StartDate, StartTime, EndDate, EndTime,
   delta, dmy_specifier, time_ref)
```

Arguments

`oldfts`	Existing financial time series object
`StartDate`	First date in the range from which data is to be extracted.
`StartTime`	Beginning time on each day. If you do not require specific times or `oldfts` does not contain time information, use `[]`. If you specify `StartTime`, you must also specify `EndTime`.
`EndDate`	Last date in the range from which data is to be extracted.
`EndTime`	Ending time on each day. If you do not require specific times or `oldfts` does not contain time information, use `[]`. If you specify `EndTime`, you must also specify `StartTime`.
`delta`	Skip interval. Can be any positive integer. Units for the skip interval specified by `dmy_specifier`.
`dmy_specifier`	Specifies the units for `delta`. Can be `D, d` (Days) `M, m` (Months) `Y, y` (Years)
`time_ref`	Time reference intervals or specific times. Valid time reference intervals are 1, 5, 15, or 60 minutes. Enter specific times as `'hh:mm'`.

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 produces

```
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 one 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 five-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

Purpose Get names of fields

Syntax

```
fnames = fieldnames(tsobj)
fnames = fieldnames(tsobj, srsnameonly)
```

Arguments

`tsobj`	Financial time series object
`srsnameonly`	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(tsobj)` returns the field names associated with the financial time series object `tsobj` as a cell array of strings, including the common fields: `desc`, `freq`, `dates` (and `times` if present).

`fnames = fieldnames(tsobj, 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`

Purpose

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

`oldfts` Financial time series object

`fill_method` (Optional) Replaces missing values (`NaN`) in `oldfts` using an interpolation process, a constant, or a zero-order hold.

Valid fill methods (interpolation methods) are:
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 `interp1` 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`.

`newdates` (Optional) Column vector of serial dates, a date string, or a column cell array of date strings. If `oldfts` contains time of day information, `newdates` must be accompanied by a time vector (`newtimes`). Otherwise, `newdates` is assumed to have times of `'00:00'`.

`T1, T2, TS, TE`	First time, second time, start time, end time
`delta`	Time interval in minutes to span between the start time and end time
`sortmode`	(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

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

```
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 1. Fill the missing data in `OpenFts` using cubic interpolation.

```
FilledFts = fillts(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 2. Fill the missing data in `OpenFts` with a constant value.

```
FilledFts = fillts(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 3. You can use `fillts` 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 = fillts(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 4. 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` in the MATLAB documentation

filter

Purpose 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` in the MATLAB documentation

Purpose 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

`dates_and_data` — 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 `'dates'` containing the date information and another labeled `'times'` containing the time information.

You can use the function `today` to enter date information or the function `now` to enter date with time information.

`dates` — 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. Valid date and time string formats are

- `'ddmmmyy hh:mm'` or `'ddmmmyyyy 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'`

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 `fints`.

You can use the MATLAB functions `today` and `now` to assist in entering date and time information.

`data`	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.
`datanames`	Cell array of data series names. Overrides the default data series names. Default data series names are `series1`, `series2`, ...

`freq`	Frequency indicator. Allowed values 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`
	Default = `Unknown`.

`desc`	String providing descriptive name for financial time series object. Default = `''`.

Note The toolbox only supports hourly and minute time series. Seconds are disregarded when the object is created (e.g., 01-jan-2001 12:00:01 is considered to be 01-jan-2001 12:00). If there are duplicate dates and times, `fints` sorts the dates and times and chooses the first instance of the duplicate dates and times. The other duplicate dates and times are removed from the object along with their corresponding data.

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.

Examples

Example 1: Create a financial time series containing days and data only:

```
data = [1:6]'

data =

     1
     2
     3
     4
     5
     6

dates = [today:today+5]'
```

```
dates =

      731132
      731133
      731134
      731135
      731136
      731137

tsobjkt = fints(dates, data)

tsobjkt =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (6)'    'series1:  (6)'
    '08-Oct-2001'    [            1]
    '09-Oct-2001'    [            2]
    '10-Oct-2001'    [            3]
    '11-Oct-2001'    [            4]
    '12-Oct-2001'    [            5]
    '13-Oct-2001'    [            6]
```

Example 2. Expand the above example to include time-of-day information:

```
dates = [now:now+5]';

tsobjkt = fints(dates, data)

tsobjkt =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (6)'    'times:  (6)'    'series1:  (6)'
    '08-Oct-2001'    '14:51'          [            1]
    '09-Oct-2001'    '14:51'          [            2]
    '10-Oct-2001'    '14:51'          [            3]
    '11-Oct-2001'    '14:51'          [            4]
```

```
'12-Oct-2001'    '14:51'          [              5]
'13-Oct-2001'    '14:51'          [              6]
```

Example 3. Create a financial time series object when dates and times are located in separate vectors.

Step 1. Create a column vector of times in date number format:

```
times = datenum(datestr(now:1/24+1/24/60:now+6/24+1/24/60,15))

times =

   0.43750000000000
   0.47986111111111
   0.52222222222222
   0.56458333333333
   0.60694444444444
   0.64930555555556
```

Step 2. Create a column vector of dates:

```
dates = [today:today+5]'

dates =

      731133
      731134
      731135
      731136
      731137
      731138
```

Step 3. Concatenate dates and times into a single matrix:

```
dates_times = [dates, times]

dates_times =

  1.0e+005 *

   7.31133000000000   0.00000437500000
   7.31134000000000   0.00000479861111
```

```
7.31135000000000    0.00000522222222
7.31136000000000    0.00000564583333
7.31137000000000    0.00000606944444
7.31138000000000    0.00000649305556
```

Step 4. Create column vector of data:

```
data = [1:6]'
```

Step 5. Create the financial time series object:

```
tsobj = fints(dates_times, data)

tsobj =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (6)'    'times:  (6)'    'series1:  (6)'
    '09-Oct-2001'    '10:30'          [            1]
    '10-Oct-2001'    '11:31'          [            2]
    '11-Oct-2001'    '12:32'          [            3]
    '12-Oct-2001'    '13:33'          [            4]
    '13-Oct-2001'    '14:34'          [            5]
    '14-Oct-2001'    '15:35'          [            6]
```

See Also

`datenum`, `datestr`

fpctkd

Purpose

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

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). Valid parameter names are:

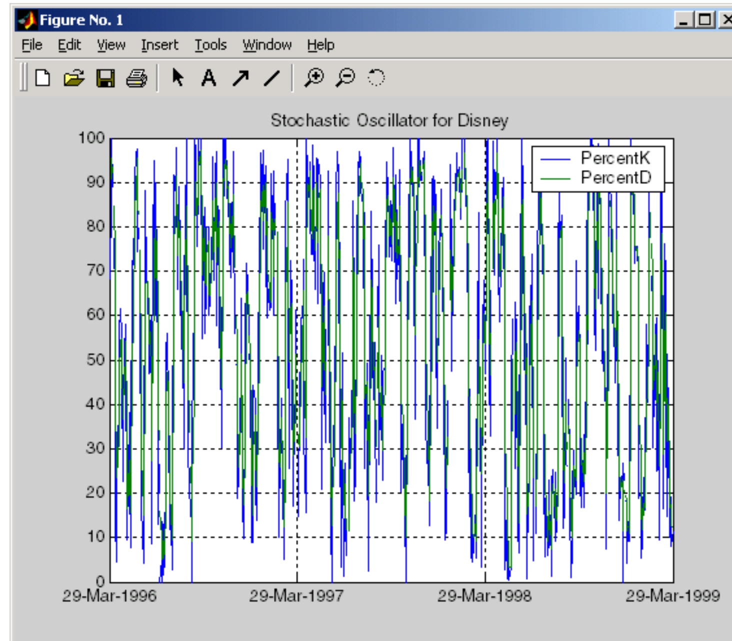
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the stochastic oscillator for Disney stock and plot the results:

```
load disney.mat
dis_FastStoc = fpctkd(dis)
plot(dis_FastStoc)
title('Stochastic Oscillator for Disney')
```



See Also

`spctkd`, `stochosc`, `tsmovavg`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 268 - 271.

Purpose 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

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

returns

```
Decimal =
         12.1250
```

See Also `cur2frac`, `cur2str`

freqnum

Purpose

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
`SEMIANNUAL`, `Semiannual`, `semiannual`, `S`, `s`	5
`ANNUAL`, `Annual`, `annual`, `A`, `a`	6

See Also

`freqstr`

Purpose 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`
5	`Semiannual`
6	`Annual`

See Also

`freqnum`

frontcon

Purpose Mean-variance efficient frontier

Syntax

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

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 `NumPorts` is empty (entered as `[]`, `frontcon` computes 10 equally spaced points. When entering a target rate of return (`PortReturn`), enter `NumPorts` as an empty matrix `[]`.
`PortReturn`	(Optional) Vector of length equal to the number of portfolios (`NPORTS`) containing the target return values on the frontier. If `PortReturn` is not entered or `[]`, `NumPorts` 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. `Groups(i,j) = 1` (*j*th asset belongs in the `i`th group). `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.

Description

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

Given three assets with expected returns of

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

and expected covariance of

```
ExpCovariance = [ 0.0100   -0.0061    0.0042
                 -0.0061    0.0400   -0.0252
                  0.0042   -0.0252    0.0225];
```

compute the mean-variance efficient frontier for four points.

```
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
```

See Also

ewstats, portopt, portstats

Purpose 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 non`NaN` 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.

See Also

`portcons`, `portopt`

Purpose

Write elements of time-series data into ASCII file

Syntax

```
stat = fts2ascii(filename, tsobj, exttext)
stat = fts2ascii(filename, dates, data, colheads, desc, exttext)
```

Arguments

`filename`	Name of an ASCII file
`tsobj`	Financial time series object
`exttext`	(Optional) Extra text. A string written after the description line (line 2 in the file).
`dates`	Column vector containing dates. Dates must be in serial date number format and can specify time of day.
`data`	Column-oriented matrix. Each column is a series.
`colheads`	(Optional) Cell array of column headers (names); first cell must always be the one for the dates column. `colheads` will be written to the file just before the data.
`desc`	(Optional) Description string, which will be the first line in the file.

Description

`stat = fts2ascii(filename, tsobj, exttext)` writes the financial time series object `tsobj` 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`).

See Also

`ascii2fts`

fts2mat

Purpose Convert to matrix

Syntax

```
tsmat = fts2mat(tsobj)
tsmat = fts2mat(tsobj, datesflag)
tsmat = fts2mat(tsobj, seriesnames)
tsmat = fts2mat(tsobj, datesflag, seriesnames)
```

Arguments

`tsobj`	Financial time series object
`datesflag`	(Optional) Specifies inclusion of dates vector: `datesflag = 0` (default) excludes dates. `datesflag = 1` includes dates vector.
`seriesnames`	(Optional) Specifies the data series to be included in the matrix. Can be a cell array of strings.

Description

`tsmat = fts2mat(tsobj)` takes the data series in the financial time series object `tsobj` 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 `tsobj`.

`tsmat = fts2mat(tsobj, 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(tsobj, 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(tsobj, 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

Purpose

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 `datestr` 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

Purpose Financial time series GUI

Syntax

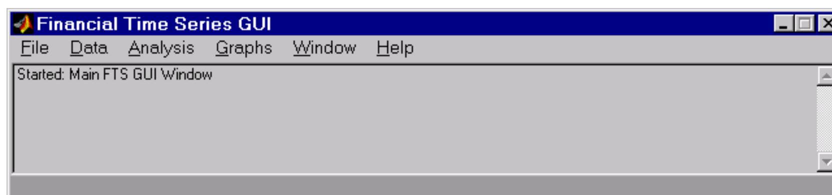
```
ftsgui
```

Description `ftsgui` displays the financial time series graphical user interface (GUI) main window.

The use of the financial time series GUI is described in Chapter 8, "Financial Time Series Graphical User Interface."

Example

```
ftsgui
```



Purpose

Financial time series object information

Syntax

```
ftsinfo(tsobj)
infofts = ftsinfo(tsobj)
```

Arguments

tsobj	Financial time series object

Description

`ftsinfo(tsobj)` displays information about the financial time series object `tsobj`.

`infofts = ftsinfo(tsobj)` stores information about the financial time series object `tsobj` in the structure `infofts`.

`infofts` has these fields.

Field	Contents
`version`	Financial time series object version
`desc`	Description of the time series object (`tsobj.desc`)
`freq`	Numeric representation of the time series data frequency (`tsobj.freq`). See `freqstr` for list of numeric frequencies and what they represent.
`startdate`	Earliest date in the time series
`enddate`	Latest date in the time series
`seriesnames`	Cell array containing the time series data column names
`ndata`	Number of data points in the time series
`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`

ftsuniq

Purpose

Determine uniqueness

Syntax

```
uniq = ftsuniq(dates_and_times)
[uniq, dup] = ftsuniq(dates_and_times)
```

Arguments

`dates_and_times`	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

Purpose 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 or equal to `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. 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).

Description `FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)` finds the amount received at maturity for a fully vested security.

Examples Using this data

```
Settle = '02/15/2001';
Maturity = '05/15/2001';
Price = 100;
Discount = 0.0575;
Basis = 2;

FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)
```

returns

```
FutureVal =
           101.44
```

See Also `acrudisc`, `discrate`, `prdisc`, `ylddisc`

References Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition.

Purpose

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

A savings account 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. Using this data

```
FutureVal = fvfix(0.09/12, 12*10, 200, 1500, 0)
```

returns

```
FutureVal =
          42379.89
```

See Also

`fvvar`, `pvfix`, `pvvar`

fvvar

Purpose Future value of varying cash flow

Syntax

```
FutureVal = fvvar(CashFlow, Rate, IrrCFDates)
```

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.
`IrrCFDates`	(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, IrrCFDates)` 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];

IrrCFDates = ['01/12/2000'
              '02/14/2001'
              '03/03/2001'
              '06/14/2001'
              '12/01/2001'];

FutureVal = fvvar(CashFlow, 0.09, IrrCFDates)
```

returns

```
FutureVal =

            170.66
```

See Also

fvfix, irr, payuni, pvfix, pvvar

fwd2zero

Purpose

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` — (Optional) Output day-count basis for annualizing the output zero rates. 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).

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

Given an implied forward rate curve over a set of maturity dates, a settlement date, and a compounding rate, compute the zero curve.

```
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;
```

Execute the function

```
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,...
Settle, Compounding)
```

which returns the zero curve `ZeroRates` at the maturity dates `CurveDates`.

```
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
```

For readability, `ForwardRates` and `ZeroRates` are shown here only to the basis point. However, MATLAB computed them at full precision. If you enter `ForwardRates` as shown, `ZeroRates` may differ due to rounding.

See Also

`zero2fwd` and other functions for Term Structure of Interest Rates

Purpose Content of specific field

Syntax

```
fieldval = getfield(tsobj, field)
fieldval = getfield(tsobj, field, {dates})
```

Arguments

tsobj	Financial time series object
field	Field name within tsobj
dates	Date range. Dates can be expanded to include time-of-day information.

Description

getfield treats the contents of a financial times series object tsobj as fields in a structure.

fieldval = getfield(tsobj, field) returns the contents of the specified field. This is equivalent to the syntax fieldval = tsobj.field.

fieldval = getfield(tsobj, 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`

getnameidx

Purpose Find name in list

Syntax

```
nameidx = getnameidx(list, name)
```

Arguments

`list`	A cell array of name strings
`name`	A 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

`findstr`, `strcmp`, `strfind`

hhigh

Purpose Highest high

Syntax

```
hhv = hhigh(data)
hhv = hhigh(data, nperiods, dim)
hhvts = hhigh(tsobj, nperiods)
hhvts = hhigh(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

`data`	Data series matrix
`nperiods`	(Optional) Number of periods. Default = 14.
`dim`	(Optional) Dimension
`tsobj`	Financial time series object

Description

`hhv = hhigh(data)` generates a vector of highest high values the past 14 periods from the matrix `data`.

`hhv = hhigh(data, nperiods, dim)` generates a vector of highest high values 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:

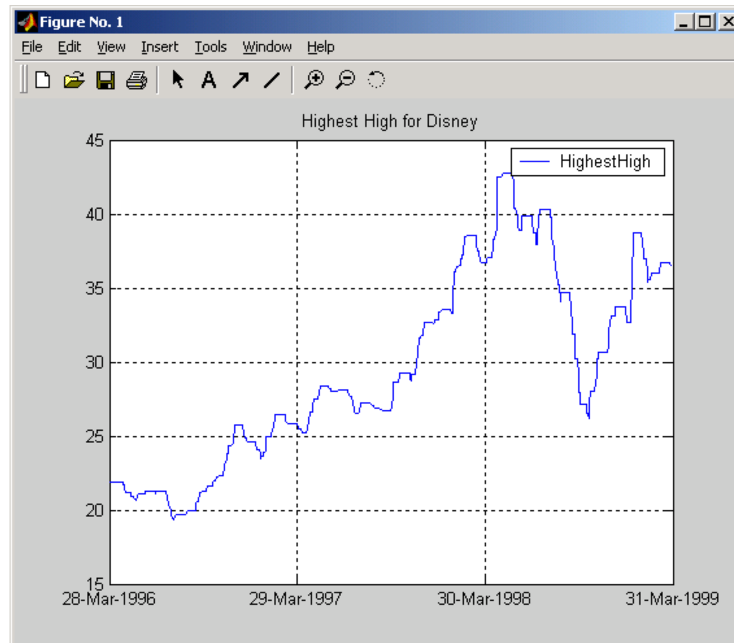
- `HighName`: high prices series name

The parameter value is a string that represents the valid parameter name.

Example

Compute the highest high prices 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

Purpose Time series High-Low plot

Syntax

```
highlow(tsobj)
highlow(tsobj, color)
highlow(tsobj, color, dateform)
highlow(tsobj, color, dateform, ParameterName, ParameterValue, ...)
hhll = 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 `plot` in the MATLAB documentation.)
`dateform`	(Optional) Date string format used as the *x*-axis tick labels. (See `datetick` in the MATLAB documentation.) You can specify a `dateform` only when `tsobj` does not contain time-of-day data. If `tsobj` contains time-of-day data, `dateform` is restricted to `'dd-mmm-yyyy HH:MM'`.

Description

`highlow(tsobj)` generates a High-Low 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).

`highlow(tsobj, color)` additionally specifies the color of the plot.

`highlow(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.

`highlow(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

You can specify open prices as optional by providing the parameter name `'OpenName'` and the parameter value `''` (empty string).

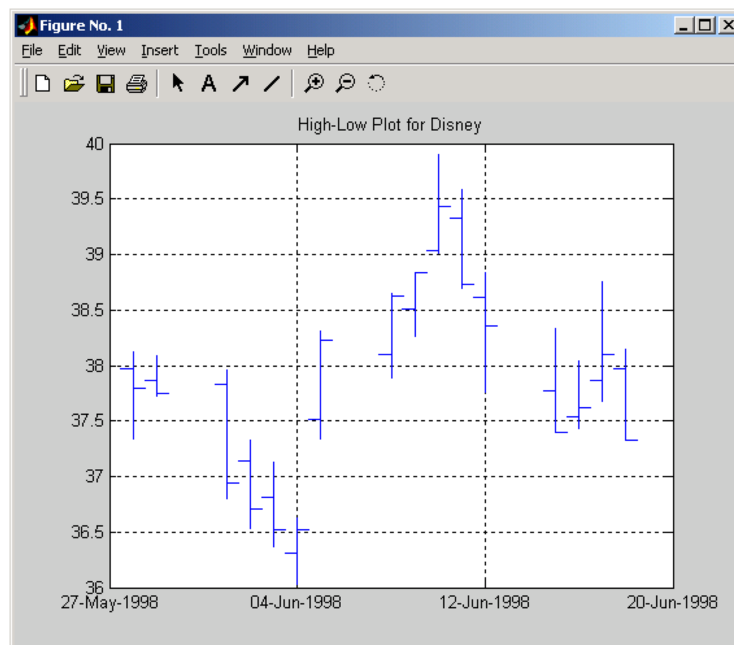
```
highlow(tsobj, color, dateform, 'OpenName', '')
```

`hhll = highlow(tsobj, color, dateform, ParameterName, ParameterValue, ...)` returns the handle to the line object that makes up the High-Low plot.

Examples

Generate a High-Low plot for Disney stock for the dates from 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

Purpose

High, low, open, close chart

Syntax

```
highlow(High, Low, Close, Open, Color)
Handles = highlow(High, Low, Close, Open, Color)
```

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 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.

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.

`Handles = highlow(High, Low, Close, Open, Color)` plots the figure and returns the handles of the lines.

Examples

The high, low, and closing prices for an asset are stored in equal-length vectors `AssetHi`, `AssetLo`, and `AssetCl` respectively

```
highlow(AssetHi, AssetLo, AssetCl, [], 'cyan')
```

plots the price data using cyan lines.

See Also

`bolling`, `candle`, `dateaxis`, `movavg`, `pointfig`

Purpose Histogram

Syntax

```
hist(tsobj, numbins)
ftshist = hist(tsobj, numbins)
[ftshist, binpos] = hist(tsobj, numbins)
```

Arguments

`tsobj`	Financial time series object
`numbins`	(Optional) Number of histogram bins. Default = 10.

Description

`hist(tsobj, numbins)` calculates and displays the histogram of the data series contained in the financial time series object `tsobj`.

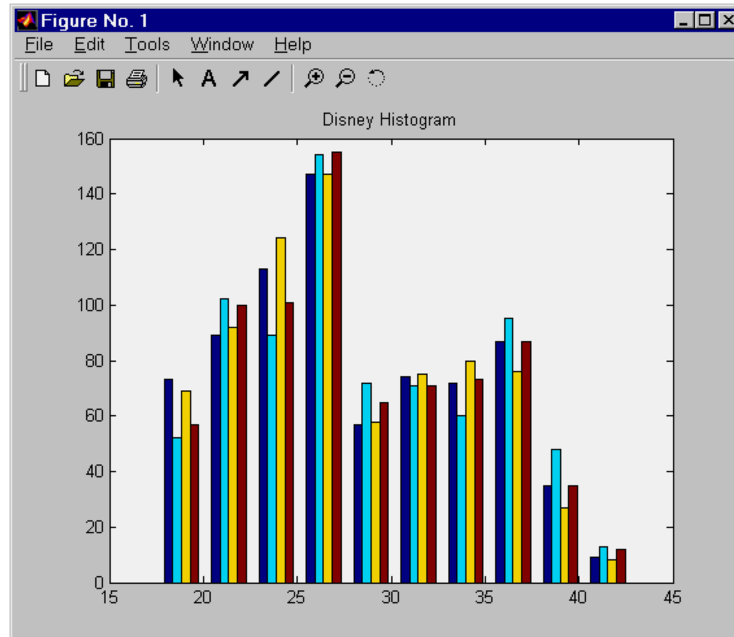
`ftshist = hist(tsobj, numbins)` calculates, but does not display, the histogram of the data series contained in the financial time series object `tsobj`. The output `ftshist` is a structure with field names similar to the data series names of `tsobj`.

`[ftshist, binpos] = hist(tsobj, numbins)` additionally returns the bin positions `binpos`. The positions are the centers of each bin. `binpos` is a column vector.

Example

Create 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` in the MATLAB documentation

Purpose 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 1. `Holdings` may be negative to indicate a short position, but the overall portfolio weights must satisfy a nonzero budget constraint.

2. The weights in each portfolio sum to the `Budget` value (which is 1 if `Budget` is unspecified.)

See Also `holdings2weights`

holidays

Purpose Holidays and nontrading days

Syntax

```
Holidays = holidays(StartDate, EndDate)
```

Arguments

`StartDate`	Start date vector. Enter as serial date numbers or date strings.
`EndDate`	End date vector. Enter as serial date numbers or date strings.

Description

`Holidays = holidays(StartDate, EndDate)` returns a vector of serial date numbers corresponding to the holidays and nontrading days between `StartDate` and `EndDate`, inclusive.

`Holidays = holidays` returns a vector of serial date numbers corresponding to all holidays and nontrading days.

As shipped, this function contains all holidays and special nontrading days for the New York Stock Exchange between 1950 and 2050. You can edit the `holidays.m` file to contain your own holidays and nontrading days. By definition, holidays and nontrading days are those that occur on weekdays.

Examples

```
Holidays = holidays('jan 1 2001', 'jun 23 2001')
```

returns

```
Holidays =

       730852
       730901
       730954
       730999
```

which are the serial date numbers for

01-Jan-2001 (New Year's Day)
19-Feb-2001 (President's Day)
13-Apr-2001 (Good Friday)
28-May-2001 (Memorial Day)

See Also `busdate`, `fbusdate`, `isbusday`, `lbusdate`

Purpose

Concatenate financial time series objects horizontally

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

Purpose 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`

irr

Purpose Internal rate of return

Syntax

```
Return = irr(CashFlow)
```

Description `Return = irr(CashFlow)` calculates the internal rate of return for a series of periodic cash flows. `CashFlow` is the cash flow vector. The first entry in `CashFlow` is the initial investment. If the initial investment is negative, `irr` generates a unique result only if all subsequent cash flows are positive. If some future cash flows are negative, `irr` generates nonunique solutions (multiple solutions that are each valid).

If the cash flow payments are monthly, multiply the resulting rate of return by 12 for the annual rate of return. This function calculates only positive rates of return; for nonpositive rates of return, `Return = NaN`.

Examples This cash flow represents the yearly income from an initial investment of $100,000:

Year 1	$10,000
Year 2	$20,000
Year 3	$30,000
Year 4	$40,000
Year 5	$50,000

To calculate the internal rate of return on the investment

```
Return = irr([-100000  10000  20000  30000  40000  50000])
```

returns

```
Return =

          0.1201
```

(12.01%)

See Also `effrr`, `mirr`, `nomrr`, `taxedrr`, `xirr`

References Brealey and Myers, *Principles of Corporate Finance*, Chapter 5

Purpose True for dates that are business days

Syntax

```
Busday = isbusday(Date, Holiday, Weekend)
```

Arguments

`Date` Date(s) being checked. Enter as a serial date number or date string. `Date` can contain multiple dates, but they must all be in the same format.

`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
```

See Also busdate, fbusdate, holidays, lbusdate

Purpose Structural equality

Syntax

```
iscomp = iscompatible(tsobj_1, tsobj_2)
```

Arguments

`tsobj_1`, `tsobj_2` 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 as well as 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

Purpose Multiple object equality

Syntax

```
iseq = isequal(tsobj_1, tsobj_2, ...)
```

Arguments

`tsobj_1 ...`	A list of financial time series objects

Description

`iseq = isequal(tsobj_1, tsobj_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

`iscompatible`

Purpose Check whether string is field name

Syntax

```
F = isfield(tsobj, name)
```

Description `F = isfield(tsobj, name)` returns true (`1`) if `name` is the name of a data series in `tsobj`. Otherwise, `isfield` returns false (`0`).

See Also `fieldnames`, `getfield`, `setfield`

issorted

Purpose 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`

Purpose

Lag time series object

Syntax

```
newfts = lagts(oldfts)
newfts = lagts(oldfts, lagperiod)
newfts = lagts(oldfts, lagperiod, padmode)
```

Arguments

`oldfts`	Financial time series object
`lagperiod`	Number of lag periods expressed in the frequency of the time series object
`padmode`	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

Purpose

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 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 = 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

c
ans =

31-May-2001
31-May-2002
30-May-2003
```

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`

length

Purpose Get number of dates (rows)

Syntax

```
lenfts = length(tsobj)
```

Description `lenfts = length(tsobj)` returns the number of dates (rows) in the financial time series object `tsobj`. This is the same as issuing `lenfts = size(tsobj, 1)`.

See Also `size`

`length` in the MATLAB documentation

leadts

Purpose

Lead time series object

Syntax

```
newfts = leadts(oldfts)
newfts = leadts(oldfts, leadperiod)
newfts = leadts(oldfts, leadperiod, padmode)
```

Arguments

`oldfts`	Financial time series object
`leadperiod`	Number of lead periods expressed in the frequency of the time series object
`padmode`	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`

llow

Purpose Lowest low

Syntax

```
llv = llow(data)
llv = llow(data, nperiods, dim)
llvts = llow(tsobj, nperiods)
llvts = llow(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

`data`	Data series matrix
`nperiods`	(Optional) Number of periods. Default = 14.
`dim`	Dimension
`tsobj`	Financial time series object

Description

`llv = llow(data)` generates a vector of lowest low values for the past 14 periods from the matrix `data`.

`llv = llow(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 = llow(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, `llow` generates a financial time series object of lowest low values for the past `nperiods` periods.

`llvts = llow(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

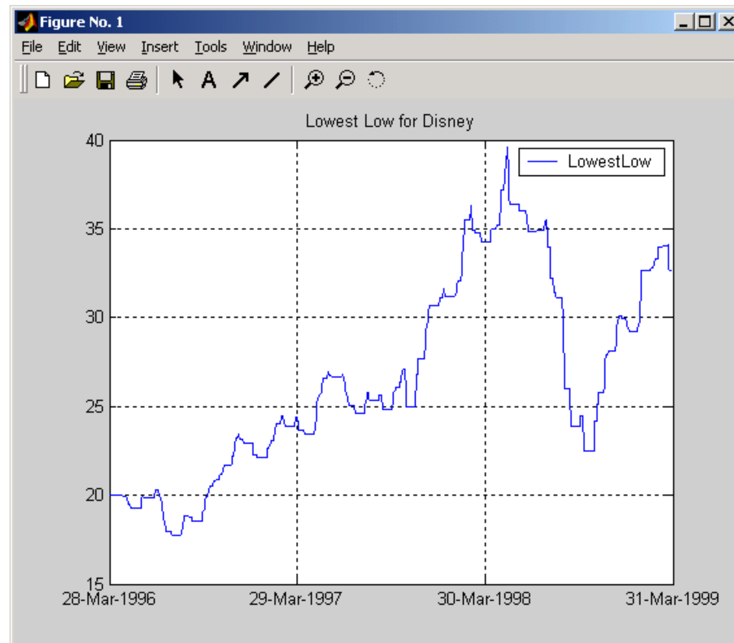
- `LowName`: low prices series name

The parameter value is a string that represents the valid parameter name.

Examples

Compute the lowest low prices for Disney stock and plot the results.

```
load disney.mat
dis_LLow = llow(dis)
plot(dis_LLow)
title('Lowest Low for Disney')
```



See Also

`hhigh`

log

Purpose

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`

log2

Purpose

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`

log10

Purpose

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`

Purpose

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 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

To find the last Monday in June 2001

```
LastDate = lweekdate(2, 2001, 6); datestr(LastDate)

ans =

25-Jun-2001
```

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
```

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

Purpose

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`	(Optional) Excel date system. A vector or scalar. When `Convention = 0` (default), the Excel 1900 date system is in effect. When `Convention = 1`, the Excel 1904 date system in used. 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.

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 693961 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with MATLAB Excel Link.

Examples

Given MATLAB date numbers for Christmas 2001 through 2004

```
DateNum = datenum(2001:2004, 12, 25)

DateNum =

      731210      731575      731940      732306
```

convert them to Excel date numbers in the 1904 system

```
ExDate = m2xdate(DateNum, 1)

ExDate =

       35788       36153       36518       36884
```

or the 1900 system

```
ExDate = m2xdate(DateNum)

ExDate =

      37250       37615       37980       38346
```

See Also

`datenum`, `datestr`, `x2mdate`

Purpose Moving Average Convergence/Divergence (MACD)

Syntax

```
[macdvec, nineperma] = macd(data)
[macdvec, nineperma] = macd(data, dim)
macdts = macd(tsobj, series_name)
```

Arguments

`data`	Data matrix
`dim`	Dimension. Default = 1 (column orientation).
`tsobj`	Financial time series object
`series_name`	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).

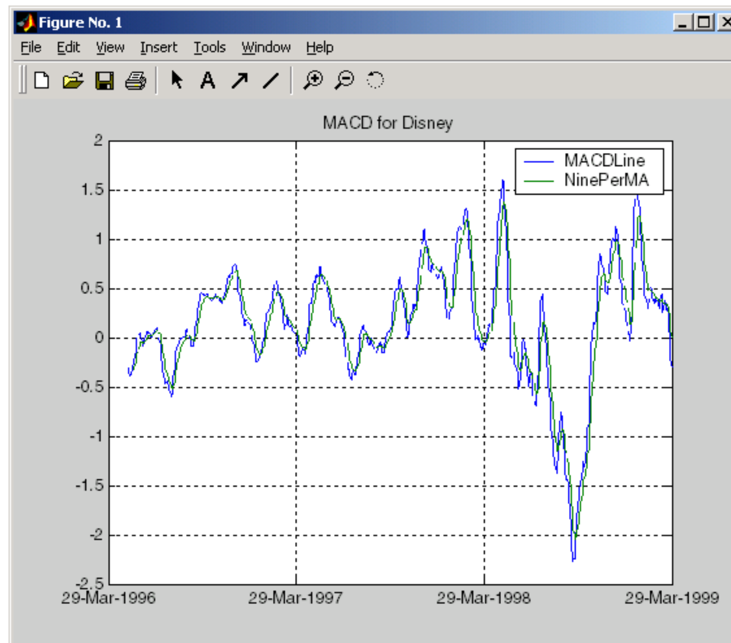
`macdts = macd(tsobj, series_name)` calculates the MACD line from the financial time series `tsobj`, as well as the nine-period exponential moving average from the MACD line. The MACD is calculated for the closing price series in `tsobj`, presumed to have been named `Close`. The result is stored in the financial time series object `macdts`. The `macdts` object has the same dates as the input object `tsobj` 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 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')
```



See Also

`adline`, `willad`

max

Purpose Maximum value

Syntax

```
tsmax = max(tsobj)
```

Description `tsmax = max(tsobj)` finds the maximum value in each data series in the financial time series object `tsobj` and returns it in a structure `tsmax`. The `tsmax` structure contains field name(s) identical to the data series name(s).

Note `tsmax` 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

Purpose

Maximum drawdown

Syntax

```
MaxDrawdown = 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 using the equation $dX(t) = \mu dt + \sigma dW(t)$ `'geometric'`: Maximum drawdown of a geometric Brownian motion with drift using the equation $dS(t) = \mu_0 S(t)dt + \sigma_0 S(t)dW(t)$

Description

`MaxDrawdown = maxdrawdown(Data, Format)` computes maximum drawdown for one or more total return price series contained in a `T`-by- `N` matrix of data, where the rows contain `T` samples of total equity over time and the columns contain `N` total equity time series.

`MaxDrawdown` is a `1`-by-`N` vector of maximum drawdown values in percentage terms for each total equity time series.

Notes 1. 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 negative 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.

2. Maximum drawdown is sensitive to quantization error.

See Also `emaxdrawdown`

Purpose

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`

Purpose 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

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). Valid parameter names are

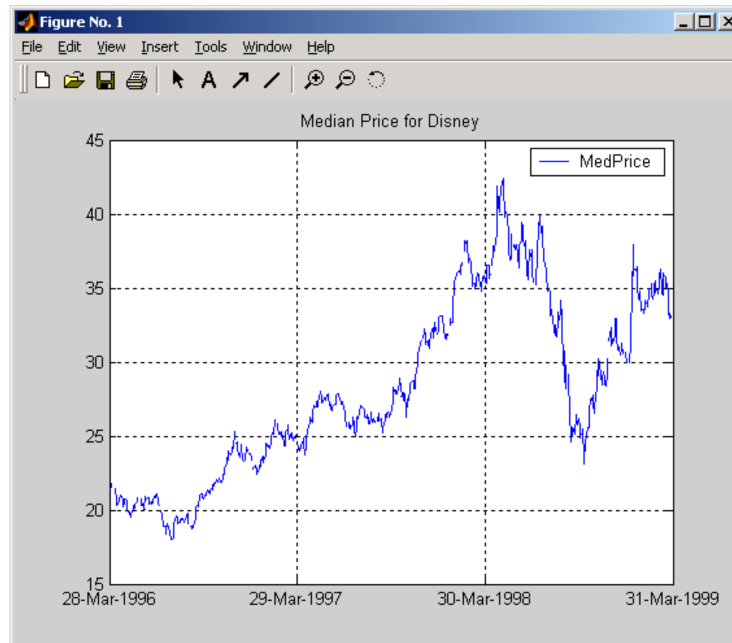
- `HighName`: high prices series name
- `LowName`: low prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

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')
```



Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 177 -178.

merge

Purpose Merge multiple financial time-series objects

Syntax

```
newfts = merge(fts1, fts2, ..., Method, RefObj)
```

Arguments

`fts1, fts2, ...` Comma-separated list of financial time series objects to merge.

> **Note** The order of the inputs is significant. Because duplicate dates and times are not supported, the first instance of a duplicate date is used. Subsequent duplicate dates (and associated data) are discarded.

`Method` (Optional) Merge method. Valid merge methods are:

`'union'` or `'u'`: (Default) Returns the combined values of all merged objects.

`'intersection'` or `'i'`: Returns the values common to all merged objects.

`'reftime'` or `'r'`: Maps all values to a reference time contained in `RefObj`.

`RefObj` (Optional) Financial time series object whose time vector is used as the reference time.

Description

`newfts = merge(fts1, fts2, ..., Method, RefObj)` merges multiple financial time series objects. The optional *`Method`* argument specifies the values contained in the output financial time series object `newfts`.

Examples

Create three financial time series objects and merge them into a single object.

```
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)

ans =

    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]%
```

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.

See Also

`horzcat`, `vertcat`

min

Purpose Minimum value

Syntax

```
tsmin = min(tsobj)
```

Description

`tsmin = min(tsobj)` finds the minimum value in each data series in the financial time series object `tsobj` and returns it in the structure `tsmin`. The `tsmin` structure contains field name(s) identical to the data series name(s).

Note `tsmin` 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

Purpose

Financial time series subtraction

Syntax

```
newfts = tsobj_1 - tsobj_2
newfts = tsobj - array
newfts = array - tsobj
```

Arguments

`tsobj_1`, `tsobj_2`	A pair of financial time series objects
`array`	A scalar value or array with the number of rows equal to the number of dates in `tsobj` and the number of columns equal to the number of data series in `tsobj`

Description

`minus` is an element by element subtraction of the components.

`newfts = tsobj_1 - tsobj_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 = tsobj - array` subtracts an array element by element from a financial time series object.

`newfts = array - tsobj` subtracts a financial time series object element by element from an array.

See Also

`rdivide`, `plus`, `times`

minute

Purpose

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`

Purpose 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 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%)
```

See Also

`annurate`, `effrr`, `irr`, `nomrr`, `pvvar`, `xirr`

References

Brealey and Myers, *Principles of Corporate Finance*, Chapter 5

month

Purpose Month of date

Syntax

```
[MonthNum, MonthString] = month(Date)
```

Description `[MonthNum, MonthString] = month(Date)` returns the month in numeric and string form given a serial date number or a date string.

Examples

```
[MonthNum, MonthString] = month(730368)
```

or

```
[MonthNum, MonthString] = month('05-Sep-1999')
```

returns

```
MonthNum =

      9

MonthString =

Sep
```

See Also `datevec`, `day`, `year`

months

Purpose

Number of whole months between dates

Syntax

```
Months = 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

`Months = months(StartDate, EndDate, EndMonthFlag)` returns the number of whole months between `StartDate` and `EndDate`. If `EndDate` is earlier than `StartDate`, `Months` 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. `Months` is then an n-by-1 vector of numbers.

Examples

```
Months = months('may 31 2000', 'jun 30 2000', 1)
Months =
        1

Months = months('may 31 2000','jun 30 2000', 0)
Months =
        0

Dates = ['mar 31 2002'; 'apr 30 2002'; 'may 31 2002'];
Months = months(Dates, 'jun 30 2002')
Months =
        3
        2
        1
```

See Also

`yearfrac`

Purpose

Leading and lagging moving averages chart

Syntax

```
movavg(Asset, Lead, Lag, Alpha)
[Short, Long] = movavg(Asset, Lead, Lag, Alpha)
```

Arguments

`Asset`	Security data, usually 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`	(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, etc. To calculate the exponential moving average, set `Alpha ='e'`.

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.

Notes 1. Moving averages smooth data via a sliding window to engender a better view of trends. Starting at the first element, the function slides along the input vector (`Asset`), capturing all data within the size of the window. The first window - 1 elements contain invalid information because there is insufficient data to compute an average until the `N`th element is reached, where `N` = the size of the window. The simple moving average (`Alpha = 0`) retains the window - 1 invalid points of data in its computation, while the exponential moving average (`Alpha = e`) removes them.

2. The size of the `Short` and `Long` output matrices depends upon the type of moving average you are computing, as determined by the `Alpha` input argument. For all forms of moving average, except exponential, the size of the output matrics is number of observations (`NUMOBS`) by number of assets

(NUMASSETS). For the exponential moving average only, the size of the Short and Long output matrices is NUMOBS - (N - 1) by NUMASSETS.

Examples

If Asset is a vector of stock price data

```
movavg(Asset, 3, 20, 1)
```

plots linear three-sample leading and 20-sample lagging moving averages.

See Also

bolling, candle, dateaxis, highlow, pointfig

mrdivide

Purpose Financial time series matrix division

Syntax

```
newfts = tsobj_1 / tsobj_2
newfts = tsobj / array
newfts = array / tsobj
```

Arguments

`tsobj_1`, `tsobj_2`	A pair of financial time series objects
`array`	A scalar value or array with number of rows equal to the number of dates in `tsobj` and number of columns equal to the number of data series in `tsobj`.

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

Purpose Financial time series matrix multiplication

Syntax

```
newfts = tsobj_1 * tsobj_2
newfts = tsobj * array
newfts = array * tsobj
```

Arguments

`tsobj_1`, `tsobj_2`	A pair of financial time series objects
`array`	A scalar value or array with number of rows equal to the number of dates in `tsobj` and number of columns equal to the number of data series in `tsobj`.

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 = tsobj_1 * tsobj_2` multiplies financial time series objects element by element.

`newfts = tsobj * array` multiplies a financial time series object element by element by an array.

`newfts = array * tsobj` `newfts = array / tsobj` 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 `mrdivide`, `minus`, `plus`, `times`

Purpose Fisher information matrix for multivariate normal or least-squares regression

Syntax

```
Fisher = mvnrfish(Data, Design, Covariance, MatrixFormat)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `NaN`s, the sample is ignored.

`Design` — A matrix or a cell array that handles two model structures:

- If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series.
- If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.

`Covariance` — `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the residuals of the regression.

`MatrixFormat` — (Optional) String that identifies parameters to be included in the Fisher information matrix:

- `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.

mvnrfish

Description

`Fisher = mvnrfish(Data, Design, Covariance, MatrixFormat)` 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'`,

```
TOTALPARAMS = NUMPARAMS + NUMSERIES * (NUMSERIES + 1)/2
```

If `MatrixFormat = 'paramonly'`,

```
TOTALPARAMS = NUMPARAMS
```

Note `mvnrfish` operates slowly if you calculate the full Fisher information matrix.

See Also

`mvnrstd`, `mvnrmle`

Purpose

Multivariate normal regression (ignore missing data)

Syntax

```
[Parameters, Covariance, Resid, Info] = mvnrmle(Data, Design,
   MaxIterations, TolParam, TolObj, Covar0)
```

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 ecmmvnrmle to handle missing data.)

`Design` — A matrix or a cell array that handles two model structures:

- If NUMSERIES = 1, Design is a NUMSAMPLES-by-NUMPARAMS matrix with known values. This structure is the standard form for regression on a single series.
- If NUMSERIES ≥ 1, Design is a cell array. The cell array contains either one or NUMSAMPLES cells. Each cell contains a NUMSERIES-by-NUMPARAMS matrix of known values.

If Design has a single cell, it is assumed to have the same Design matrix for each sample. If Design has more than one cell, each cell contains a Design matrix for each sample.

`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 sqrt(eps) which is about 1.0e-8 for double precision. The convergence test for changes in model parameters is

$$\|\mathtt{Param}_k - \mathtt{Param}_{k-1}\| < \mathtt{TolParam} \times (1 + \|\mathtt{Param}_k\|)$$

where Param represents the output Parameters, and iteration $k = 2, 3, \ldots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both TolParam ≤ 0 and TolObj ≤ 0, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.

TolObj	(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 $$\lvert \mathtt{Obj}_k - \mathtt{Obj}_{k-1} \rvert < \mathtt{TolObj} \times (1 + \lvert \mathtt{Obj}_k \rvert)$$ for iteration $k = 2, 3, \ldots$. Convergence is assumed when both the TolParam and TolObj conditions are satisfied. If both TolParam ≤ 0 and TolObj ≤ 0, do the maximum number of iterations (MaxIterations), whatever the results of the convergence tests.
Covar0	(Optional) NUMSERIES-by-NUMSERIES matrix that contains a user-supplied initial or known estimate for the covariance matrix of the regression residuals.

Description

[Parameters, Covariance, Resid, Info] = mvnrmle(Data, Design, MaxIterations, TolParam, TolObj, Covar0) estimates a multivariate normal regression model without missing data. The model has the form

$$\mathtt{Data}_k \sim N(\mathtt{Design}_k \times \mathtt{Parameters}, \mathtt{Covariance})$$

for samples k = 1, ... , NUMSAMPLES.

mvnrmle estimates a NUMPARAMS-by-1 column vector of model parameters called Parameters, and a NUMSERIES-by-NUMSERIES matrix of covariance parameters called Covariance.

mvnrmle(Data, Design) with no output arguments plots the log-likelihood function for each iteration of the algorithm.

To summarize the outputs of mvnrmle:

- 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 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

mvnrmle 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.
- Two functions for handling missing data, ecmmvnrmle and ecmlsrmle, are stricter about the presence of NaN values in Design.

Use the estimates in the optional output structure `Info` for diagnostic purposes.

See Also

`mvnrstd`, `mvnrobj`, `ecmmvnrmle`

References

[1] Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[2] Xiao-Li Meng and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

Purpose

Log-likelihood function for multivariate normal regression without missing data

Syntax

```
Objective = mvnrobj(Data, Design, Parameters, Covariance)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `NaN`s, the sample is ignored. (Use `ecmmvnrmle` to handle missing data.)
`Design`	A matrix or a cell array that handles two model structures: • If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series. • If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values. If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.
`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.

Description

`Objective = mvnrobj(Data, Design, Parameters, Covariance)` 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.

See Also

`mvnrmle`, `ecmmvnrmle`, `ecmmvnrobj`

Purpose Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters, StdCovariance] = mvnrstd(Data, Design, Covariance)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `NaN`s, the sample is ignored. (Use `ecmmvnrmle` to handle missing data.)

`Design` — A matrix or a cell array that handles two model structures:

- If `NUMSERIES = 1`, `Design` is a `NUMSAMPLES`-by-`NUMPARAMS` matrix with known values. This structure is the standard form for regression on a single series.
- If `NUMSERIES ≥ 1`, `Design` is a cell array. The cell array contains either one or `NUMSAMPLES` cells. Each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix of known values.

If `Design` has a single cell, it is assumed to have the same `Design` matrix for each sample. If `Design` has more than one cell, each cell contains a `Design` matrix for each sample.

`Covariance` — `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the regression residuals.

mvnrstd

Description

`[StdParameters, StdCovariance] = mvnrstd(Data, Design, Covariance)` evaluates standard errors for a multivariate normal regression model without missing data. The model has the form

$$\mathtt{Data}_k \sim N(\mathtt{Design}_k \times \mathtt{Parameters},\ \mathtt{Covariance})$$

for samples k = 1, ... , `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.

See Also

`mvnrmle`, `ecmmvnrmle`, `ecmmvnrstd`

References

[1] Roderick J. A. Little and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

Purpose Negative volume index

Syntax

```
nvi = negvolidx(closep, tvolume, initnvi)
nvi = negvolidx([closep tvolume], initnvi)
nvits = negvolidx(tsobj)
nvits = negvolidx(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

Description

`nvi = negvolidx(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(tsobj)` calculates the negative volume index from the financial time series object `tsobj`. The object must contain, at least, the series `Close` and `Volume`. The `nvits` output is a financial time series object with dates similar to `tsobj` and a data series named `NVI`. The initial value for the negative volume index is arbitrarily set to 100.

`nvits = negvolidx(tsobj, 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). Valid parameter names are

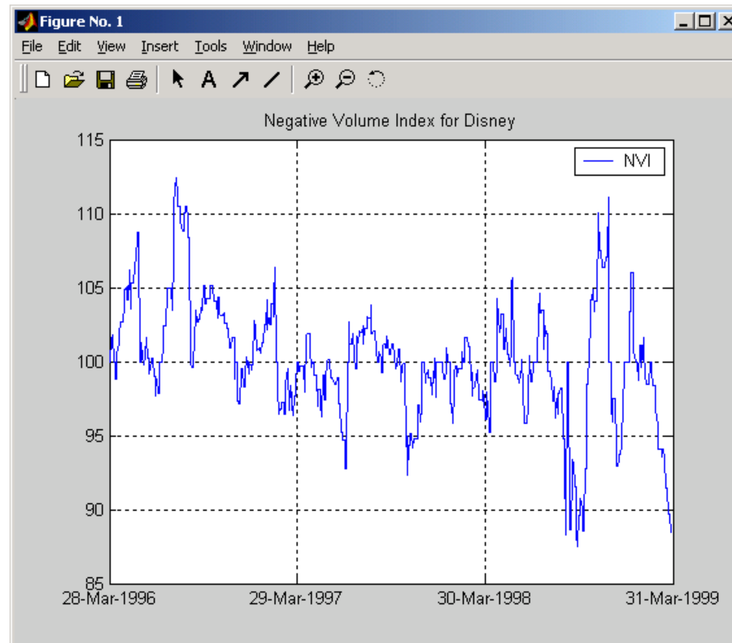
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the 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')
```



See Also

`onbalvol`, `posvolidx`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 193 - 194.

Purpose

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

To find the nominal annual rate of return based on an effective annual percentage rate of 9.38% compounded monthly

```
Return = nomrr(0.0938, 12)
```

returns

```
Return =
          0.0900 (9.0%)
```

See Also

`effrr`, `irr`, `mirr`, `taxedrr`, `xirr`

now

Purpose Current date and time

Syntax `Datenum = now`

Description `Datenum = now` returns the current date and time as a serial date number.

Note This function now ships with basic MATLAB. It originally shipped only with the Financial Toolbox. This description remains here for your convenience.

Examples

```
Datenum = now

Datenum =

     730695.5942469908
```

(on July 28, 2000 at 2:15 PM)

See Also `date`, `datenum`, `today`

Purpose 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

Purpose On-Balance Volume (OBV)

Syntax

```
obv = onbalvol(closep, tvolume)
obv = onbalvol([closep tvolume])
obvts = onbalvol(tsobj)
obvts = onbalvol(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

`closep`	Closing price (vector)
`tvolume`	Volume traded
`tsobj`	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(tsobj)` calculates the OBV from the stock data in the financial time series object `tsobj`. The object must minimally contain series names `Close` and `Volume`. The `obvts` output is a financial time series object with the same dates as `tsobj` and a series named `OnBalVol`.

`obvts = onbalvol(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

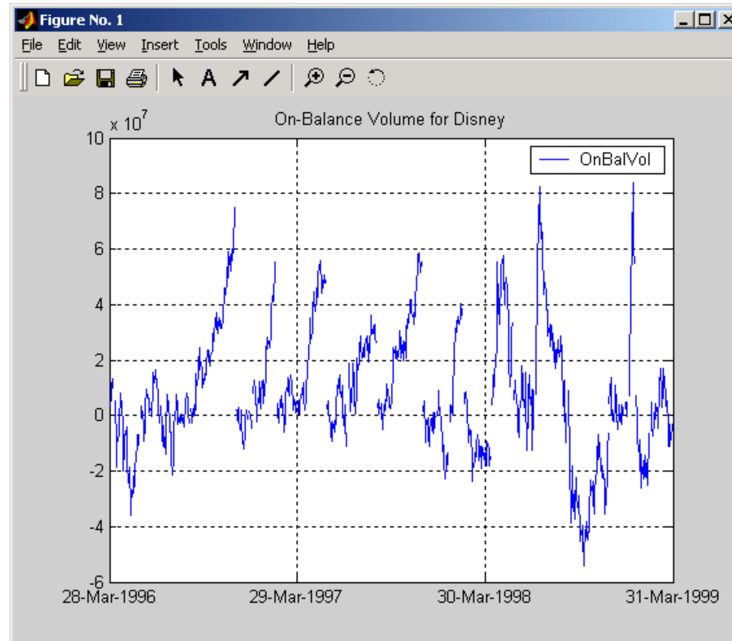
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute 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')
```



See Also

`negvolidx`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 207 - 209.

opprofit

Purpose

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

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)
```

returns

```
Profit =
            6.00
```

a profit of $6 if the option is exercised under these conditions.

See Also

`binprice`, `blsprice`

payadv

Purpose 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

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. Using this data

```
Payment = payadv(0.1/12, 12, 1000, 0, 3)
```

returns

```
Payment =

          85.94
```

for the periodic payment.

See Also

`amortize`, `payodd`, `payper`

Purpose Payment of loan or annuity with odd first period

Syntax

```
Payment = payodd(Rate, NumPeriods, PresentValue, FutureValue, Days)
```

Arguments

`rate`	Interest rate per period. Enter as a decimal fraction.
`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.
`Days`	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 A two-year loan for $4000 has an annual interest rate of 11%. The first payment will be made in 36 days. To find the monthly payment

```
Payment = payodd(0.11/12, 24, 4000, 0, 36)
```

returns

```
Payment =

           186.77
```

See Also `amortize`, `payadv`, `payper`

payper

Purpose 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

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)
```

returns

```
Payment =

          297.86
```

See Also

`amortize`, `fvfix`, `payadv`, `payodd`, `pvfix`

Purpose

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

Purpose Linear inequalities for individual asset allocation

Syntax

```
[A,b] = pcalims(AssetMin, AssetMax, NumAssets)
```

Arguments

AssetMin	Scalar or NASSETS vector of minimum allocations in each asset. NaN indicates no constraint.
AssetMax	Scalar or NASSETS vector of maximum allocations in each asset. NaN indicates no constraint.
NumAssets	(Optional) Number of assets. Default = length of AssetMin or AssetMax.

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 a vector such that A*PortWts' <= 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 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.

See Also

`pcgcomp`, `pcglims`, `pcpval`, `portcons`, `portopt`

Purpose

Linear inequalities for asset group comparison constraints

Syntax

```
[A,b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)
```

Arguments

`GroupA` `GroupB`	Number of groups (`NGROUPS`) by number of assets (`NASSETS`) specifications of groups to compare. Each row specifies a group. For a specific group, `Group(i,j) = 1` if the group contains asset `j`; otherwise, `Group(i,j) = 0`.
`AtoBmin` `AtoBmax`	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 >= `AtoBmin` and <= `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*PortWts' <= 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.

See Also

`pcalims`, `pcglims`, `pcpval`, `portcons`, `portopt`

Purpose

Linear inequalities for asset group minimum and maximum allocation

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, `Group(i,j) = 1` if the group contains asset `j`; otherwise, `Group(i,j) = 0`.
`GroupMin` `GroupMax`	Scalar or `NGROUPS`-long vectors of minimum and maximum 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*PortWts' <= 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.

```
%           INTC  XOM  RD
Groups = [   1    1   0  ;  % 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.

See Also

`pcalims`, `pcgcomp`, `pcpval`, `portcons`, `portopt`

pcpval

Purpose Linear inequalities for fixing total portfolio value

Syntax `[A,b] = pcpval(PortValue, NumAssets)`

Arguments

`PortValue`	Scalar total value of asset portfolio (sum of the allocations in all assets). `PortValue = 1` 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*PortWts' <= 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.

See Also

`pcalims`, `pcgcomp`, `pcglims`, `portcons`, `portopt`

peravg

Purpose Periodic average

Syntax

```
avgfts = peravg(tsobj, numperiod)
avgfts = peravg(tsobj, daterange)
```

Arguments

`tsobj`	Financial time series object
`numperiod`	Integer specifying the number of data points over which each periodic average should be averaged
`daterange`	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`.

See Also

`mean`, `tsmovavg`

`mean` in the MATLAB documentation

Purpose Periodic total returns from total return prices

Syntax

```
TotalReturn = periodicreturns(TotalReturnPrices, Period)
```

Arguments

`TotalReturnPrices`	Number of observations (`NUMOBS`) by number of assets (`NASSETS`) 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 `'m'` = monthly values

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` 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.

See Also `totalreturnprice`

plot

Purpose

Plot data series

Syntax

```
plot(tsobj)
hp = plot(tsobj)
plot(tsobj, linefmt)
hp = plot(tsobj, linefmt)
plot(..., volumename, bar)
hp = plot(..., volumename, bar)
```

Arguments

`tsobj`	Financial time series object
`linefmt`	(Optional) Line format
`volumename`	(Optional) Specifies which data series is the volume series. `volumename` must be the exact data series name for the volume column (case sensitive).
`bar`	(Optional) `bar = 0` (default). Plot volume as a line. `bar = 1`. Plot volume as a bar chart. The width of each bar is the same as the default in `bar`.

Description

`plot(tsobj)` plots the data series contained in the object `tsobj`. Each data series will be a line. `plot` automatically generates a legend as well as 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's 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`, `chartfts`, `highlow`

`grid`, `legend`, and `plot` in the MATLAB documentation

plus

Purpose Financial time series addition

Syntax

```
newfts = tsobj_1 + tsobj_2
newfts = tsobj + array
newfts = array + tsobj
```

Arguments

`tsobj_1`, `tsobj_2`	A pair of financial time series objects
`array`	A scalar value or array with the number of rows equal to the number of dates in `tsobj` and the number of columns equal to the number of data series in `tsobj`

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`

Purpose 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.

See Also `bolling`, `candle`, `dateaxis`, `highlow`, `movavg`

portalloc

Purpose

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).

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

Generate the efficient frontier from the asset data.

```
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);
```

Find the optimal risky portfolio and allocate capital. The risk free investment return is 8%, and the borrowing rate is 12%.

```
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
```

See Also `frontcon`, `portrand`, `portstats`

References Bodie, Kane, and Marcus, *Investments*, Second Edition, Chapters 6 and 7.

Purpose Portfolio constraints

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 `ConSet = [A b]`. `A` is a matrix and `b` a vector such that `A*PortWts' <= 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, ..., Data1N,'ConstType2', 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.	`NumAssets` (required). Scalar representing number of assets in portfolio.
`PortValue`	Fix total value of portfolio to `PVal`.	`PVal` (required). Scalar representing total value of portfolio. `NumAssets` (required). Scalar representing number of assets in portfolio. See `pcpval`.

Constraint Type	Description	Values
AssetLims	Minimum and maximum allocation per asset.	AssetMin (required). Scalar or vector of length NASSETS, specifying minimum allocation per asset. AssetMax (required). Scalar or vector of length NASSETS, specifying maximum allocation per asset. NumAssets (optional). See pcalims.
GroupLims	Minimum and maximum allocations to asset group.	Groups (required). NGROUPS-by-NASSETS matrix specifying which assets belong to each group. GroupMin (required). Scalar or a vector of length NGROUPS, specifying minimum combined allocations in each group. GroupMax (required). Scalar or a vector of length NGROUPS, specifying maximum combined allocations in each group. See pcglims.

Constraint Type	Description	Values
GroupComparison	Group-to-group comparison constraints.	GroupA (required). NGROUPS-by-NASSETS matrix specifying first group in the comparison. AtoBmin (required). Scalar or vector of length NGROUPS specifying minimum ratios of allocations in GroupA to allocations in GroupB. AtoBmax (required). Scalar or vector of length NGROUPS specifying maximum ratios of allocations in GroupA to allocations in GroupB. GroupB (required). NGROUPS-by-NASSETS matrix specifying second group in the comparison. See pcgcomp .
Custom	Custom linear inequality constraints A*PortWts' <= b.	A (required). NCONSTRAINTS-by-NASSETS matrix, specifying weights for each asset in each inequality equation. b (required). Vector of length NCONSTRAINTS specifying the right hand sides of the inequalities.

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
```

Portfolio weights of 30% in IBM, 30% in HPQ, and 40% in XOM satisfy the constraints.

See Also

`pcalims`, `pcgcomp`, `pcglims`, `pcpval`, `portopt`

portopt

Purpose Portfolios on constrained efficient frontier

Syntax

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

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.

Description

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

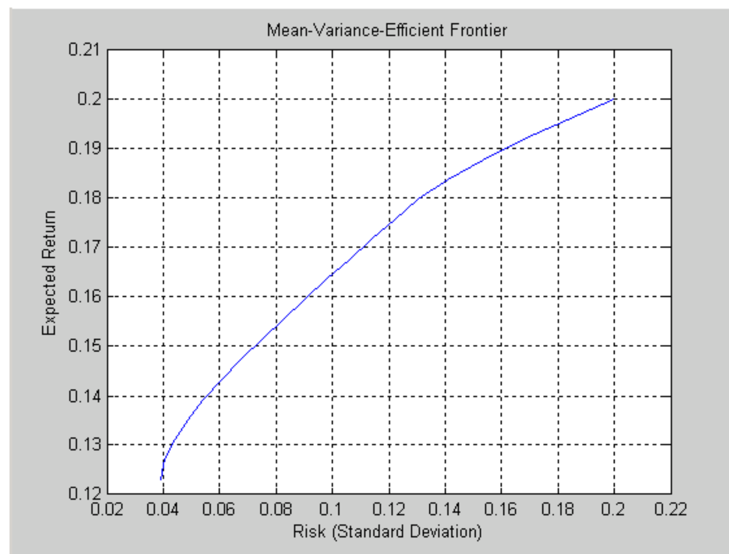
`PortWts` is an `NPORTS`-by-`NASSETS` matrix of weights allocated to each asset. Each row represents a portfolio. The total of all weights in a portfolio is 1.

If `portopt` is invoked without output arguments, it returns a plot of the efficient frontier.

Examples

Plot the risk-return efficient frontier 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
```

See Also

ewstats, frontcon, portcons, portstats

portrand

Purpose Randomized portfolio risks, returns, and weights

Syntax

```
[PortRisk, PortReturn, PortWts] = portrand(Asset, Return, Points)
portrand(Asset, Return, Points)
```

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.

Description

`[PortRisk, PortReturn, PortWts] = portrand(Asset, Return, Points)` 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)` plots the points representing each portfolio configuration. It does not return any data to the MATLAB workspace.

See Also `frontcon`

References Bodie, Kane, and Marcus, *Investments*, Chapter 7.

Purpose

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: `ExpSigma = sqrt(diag(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 mean (`ExpReturn`) and covariance (`ExpCovariance`) specifications. `'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 (`ExpReturn`) and covariance (`ExpCovariance`) specifications.) For either method the sample mean and covariance returned are appropriately scaled by `RetIntervals`.

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 1. 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.

2. 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

Example 1. Distinction Between Simulation Methods

This example highlights 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.

```
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-003 *

    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 on a daily basis, 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.

```
randn('state',0);
RetExact = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Exact');
randn('state',0);
RetExpected = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected');
```

If you compare the mean and covariance of `RetExact` with the inputs (`ExpReturn` and `ExpCovariance`), you will observe that they are almost identical.

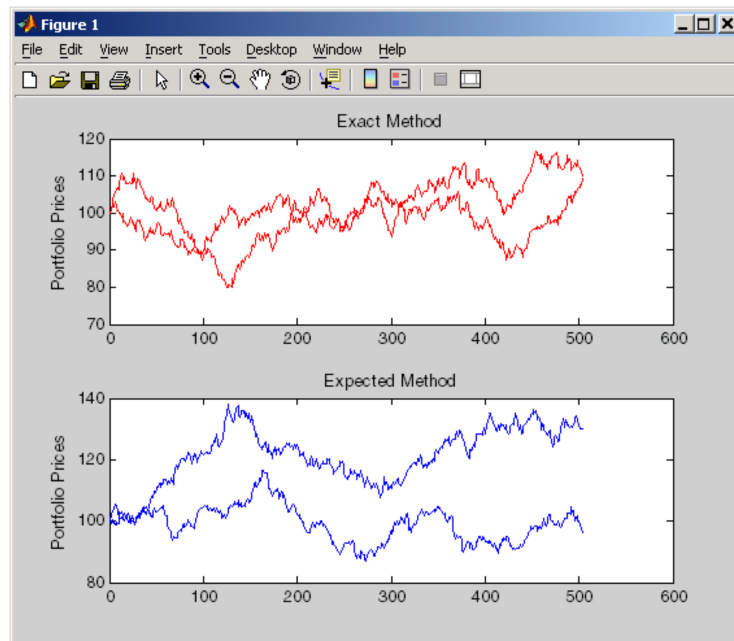
At this point, `RetExact` and `RetExpected` are both 504-by-5-by-2 arrays. Now assume an equally-weighted portfolio formed from the five assets and create arrays of portfolio returns in which each column represents the portfolio return of the corresponding sample path of the simulated returns of the 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')
```



Example 2. Interaction between `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 on a daily basis, 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.

```
randn('state',0);
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.

Example 3. Univariate Geometric Brownian Motion

This example simulates 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.

First, 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).

Now set the random number generator state, and simulate 10,000 trials (realizations) of stock returns over a full calendar year of 252 trading days.

```
randn('state',10);
RetSeries = squeeze(portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected'));
```

The `squeeze` function simply 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,:))

SampMean =

   24.4587
```

```
SampVar = var(StockPrices(end,:))

SampVar =

  104.2016
```

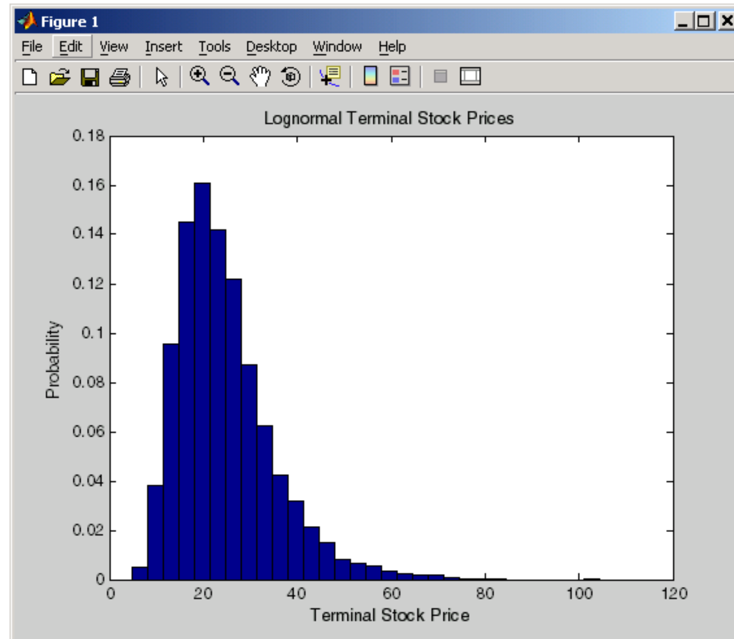
Compare these values with the values you obtain by using Hull's equations.

```
ExpValue = StartPrice*exp(ExpReturn)

ExpValue =

   24.4281

ExpVar = ...
StartPrice*StartPrice*exp(2*ExpReturn)*(exp((ExpCovariance)) - 1)

ExpVar =

  103.5391
```

These results are very close to the results shown in Hull's example 12.2.

Next, 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')
```



See Also `ewstats`, `portopt`, `portstats`, `randn`, `ret2tick`

References

Hull, John, C., *Options, Futures, and Other Derivatives*, Upper Saddle River, New Jersey: Prentice-Hall. 5th ed., 2003, ISBN 0-13-009056-5.

Purpose 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

```
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
```

See Also

`frontcon`

Purpose Portfolio value at risk

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- `RiskThreshold`.

If `PortValue` is not given, `ValueAtRisk` is presented on a per-unit basis. A value of 0 indicates no losses.

Examples

This example computes `ValueAtRisk` 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
```

This example computes ValueAtRisk 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+007 *
    3.6572
    1.8684
```

See Also frontcon, portopt

Purpose

Positive volume index

Syntax

```
pvi = posvolidx(closep, tvolume, initpvi)
pvi = posvolidx([closep tvolume], initpvi)
pvits = posvolidx(tsobj)
pvits = posvolidx(tsobj, initpvi, ParameterName, ParameterValue, ...)
```

Arguments

`closep`	Closing price (vector)
`tvolume`	Volume traded (vector)
`initpvi`	(Optional) Initial value for positive volume index Default = 100.
`tsobj`	Financial time series object

Description

`pvi = posvolidx(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, `posvolidx` uses that value instead of the default (100).

`pvi = posvolidx([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, `posvolidx` uses that value instead of the default (100).

`pvits = posvolidx(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 = posvolidx(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

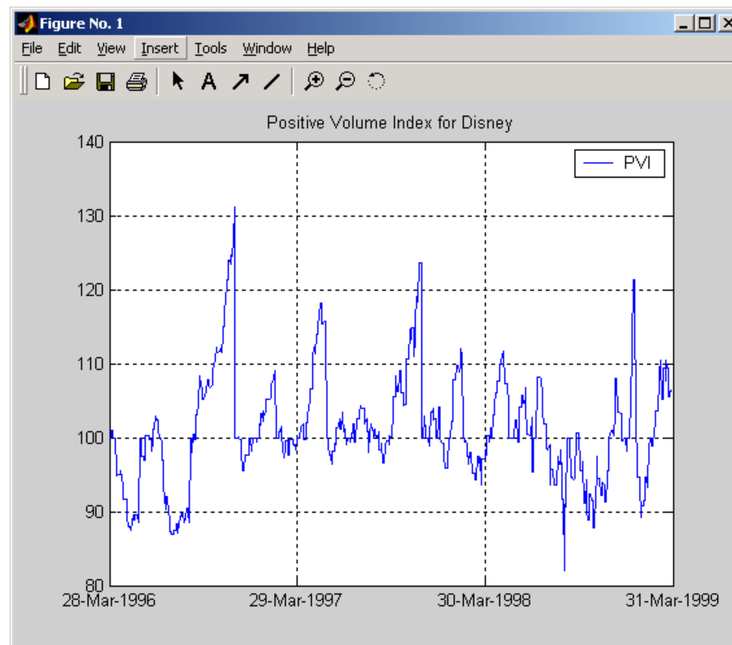
posvolidx

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the positive volume index for Disney stock and plot the results:

```
load disney.mat
dis_PosVol = posvolidx(dis)
plot(dis_PosVol)
title('Positive Volume Index for Disney')
```



See Also

`onbalvol`, `negvolidx`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 236 - 238.

Purpose

Financial time series power

Syntax

```
newfts = tsobj .^ array
newfts = array .^tsobj
newfts = tsobj_1 .^ tsobj_2
```

Arguments

`tsobj`	Financial time series object
`array`	A scalar value or array with the number of rows equal to the number of dates in `tsobj` and the number of columns equal to the number of data series in `tsobj`.
`tsobj_1`, `tsobj_2`	A 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

Purpose

Price bonds in portfolio by set of zero curves

Syntax

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

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. 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).
`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

Description

`BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)` 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`.

Examples

This example uses `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 `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. Derive the zero curve within 50 iterations.

```
OutputCompounding = 2;
OutputBasis = 3;
MaxIterations = 50;
```

Execute zbtprice

```
[ZeroRates, ZeroDates] = zbtprice(Bonds, Prices, Settle,...
OutputCompounding, OutputBasis, MaxIterations)
```

which returns the zero curve at the maturity dates.

```
ZeroRates =

            0.0616
            0.0609
            0.0658
            0.0590
            0.0648
            0.0655
            0.0606
            0.0601
            0.0642
            0.0621
            0.0627

ZeroDates =

             729907
             730364
             730439
             730500
             730667
             730668
             730971
```

```
              731032
              731033
              731321
              731336
```

Now execute `prbyzero`

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

which returns

```
BondPrices =

               99.38
               98.80
              106.83
               96.88
              103.62
              101.13
              103.12
               99.36
              101.00
              101.25
               96.37
              102.74
```

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

Purpose

Price rate of change

Syntax

```
proc = prcroc(closep, nperiods)
procts = prcroc(tsobj, nperiods)
procts = prcroc(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

`closep`	Closing price
`nperiods`	(Optional) Period difference. Default = 12.
`tsobj`	Financial time series object

Description

`proc = prcroc(closep, nperiods)` calculates the price rate of change `proc` from the closing price `closep`. If `nperiods` periods is specified, the price rate of change is calculated between the current closing price and the closing price `nperiods` ago.

`procts = prcroc(tsobj, nperiods)` 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 `nperiods` is specified, the price rate of change is calculated between the current closing price and the closing price `nperiods` ago.

`procts = prcroc(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

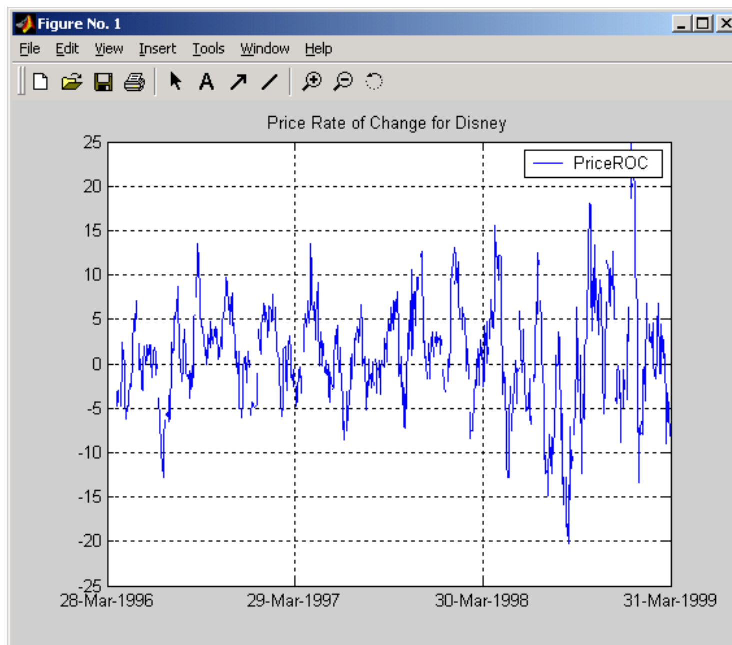
- `CloseName`: closing price series name

The parameter value is a string that represents the valid parameter name.

Examples

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')
```



See Also

`volroc`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 243 - 245.

prdisc

Purpose

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 or equal to `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`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`Price = prdisc(Settle, Maturity, Face, Discount, Basis)` returns the price of a security whose yield is quoted as a bank discount rate (e.g., U. S. Treasury Bills).

Examples

Using this data

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Discount = 0.087;
Basis = 2;

Price = prdisc(Settle, Maturity, Face, Discount, Basis)
```

returns

```
Price =

        96.2783
```

See Also

`acrudisc`, `bndprice`, `discrate`, `prmat`, `ylddisc`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 2.

prmat

Purpose 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 or equal to `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. 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).

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 Using this data

```
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)
```

returns

```
Price =

   99.9784

AccruInterest =

    1.9591
```

See Also

`acrubond`, `acrudisc`, `bndprice`, `prdisc`, `yldmat`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 4.

prtbill

Purpose 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 or equal to `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

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. Using this data

```
Price = prtbill('2/10/2002', '8/6/2002', 1000, 0.0377)
```

returns

```
Price =
        981.4642
```

See Also `beytbill`, `yldtbill`

References Bodie, Kane, and Marcus, *Investments*, pages 41-43.

Purpose 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

$200 is paid monthly into a savings account earning 6%. The payments are made at the end of the month for five years. To find the present value of these payments

```
PresentVal = pvfix(0.06/12, 5*12, 200, 0, 0)
```

returns

```
PresentVal =

          10345.11
```

See Also

`fvfix`, `fvvar`, `payper`, `pvvar`

pvtrend

Purpose 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

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). Valid parameter names are

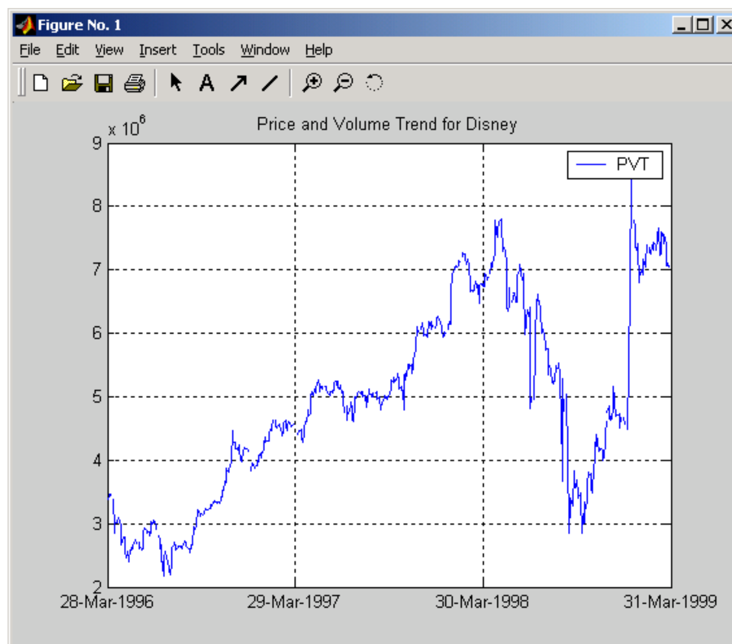
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the 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')
```



Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 239 - 240.

Purpose Present value of varying cash flow

Syntax

```
PresentVal = pvvar(CashFlow, Rate, IrrCFDates)
```

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.
`IrrCFDates`	(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

`PresentVal = pvvar(CashFlow, Rate, IrrCFDates)` returns the net present 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 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];

IrrCFDates = ['01/12/1987'
              '02/14/1988'
              '03/03/1988'
              '06/14/1988'
              '12/01/1988'];

PresentVal = pvvar(CashFlow, 0.09, IrrCFDates)
```

returns

```
PresentVal =

             142.16
```

See Also

`fvfix`, `fvvar`, `irr`, `payuni`, `pvfix`

pyld2zero

Purpose Zero curve given par yield curve

Syntax

```
[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates, Settle,
   Compounding, Basis, OutputCompounding)
```

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`	A serial date number that is the common settlement date for the par rates.
`Compounding`	(Optional) A scalar that sets the rate at which the 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 365 daily compounding
`Basis`	(Optional) Day-count basis used to annualize the zero rates. 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).
`OutputCompounding`	(Optional) Value representing the rate at which the zero rates are compounded. Default = `Compounding`.

Description

`[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates, Settle, Compounding, Basis, OutputCompounding)` returns a zero curve given a par yield 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`.
`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

Given

- A par yield curve over a set of maturity dates
- A settlement date
- Annual compounding for the input par rates and monthly compounding for the output zero curve

compute a zero yield curve.

```
ParRates = [0.0479
            0.0522
            0.0540
            0.0540
            0.0536
            0.0532
            0.0532
            0.0539
            0.0558
            0.0543];

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;
OutputCompounding = 12;

[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates,...
Settle, Compounding, [], OutputCompounding)

ZeroRates =

    0.0484
    0.0529
    0.0549
    0.0550
    0.0547
    0.0544
    0.0545
    0.0551
    0.0572
    0.0557

CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `ParRates` and `ZeroRates` are shown only to the basis point. However, MATLAB computes them at full precision. If you enter `ParRates` as shown, `ZeroRates` may differ due to rounding.

See Also

`zero2pyld` and other functions for Term Structure of Interest Rates

rdivide

Purpose Financial time series division

Syntax

```
newfts = tsobj_1 ./ tsobj_2
newfts = tsobj ./ array
newfts = array ./ tsobj
```

Arguments

`tsobj_1`, `tsobj_2`	A pair of financial time series objects
`array`	A scalar value or array with the number of rows equal to the number of dates in `tsobj` and the number of columns equal to the number of data series in `tsobj`

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`

resamplets

Purpose 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

Purpose

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*'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.
`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

```
TickSeries(i+1) = TickSeries(i)*[1 + RetSeries(i)]
```

If *Method* is 'Continuous', the prices are

```
TickSeries(i+1) = TickSeries(i)*exp[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

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)

TickSeries =

    1.0000    1.0000
    1.1000    1.1200
    1.1550    1.1648
    1.0973    1.2230

TickTimes =

      730838
      731020
      731111
      731203

datestr(TickTimes)
```

```
ans =

18-Dec-2000
18-Jun-2001
17-Sep-2001
18-Dec-2001
```

See Also

portsim, tick2ret

Purpose Remove data series

Syntax

```
fts = rmfield(tsobj, fieldname)
```

Arguments

`tsobj`	Financial time series object
`fieldname`	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(tsobj, fieldname)` removes the data series `fieldname` and its contents from the financial time series object `tsobj`.

See Also

`chfield`, `extfield`, `fieldnames`, `getfield`, `isfield`

rsindex

Purpose Relative Strength Index (RSI)

Syntax

```
rsi = rsindex(closep, nperiods)
rsits = rsindex(tsobj, nperiods)
rsits = rsindex(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

`closep`	Vector of closing prices
`nperiods`	(Optional) Number of periods. Default = 14.
`tsobj`	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

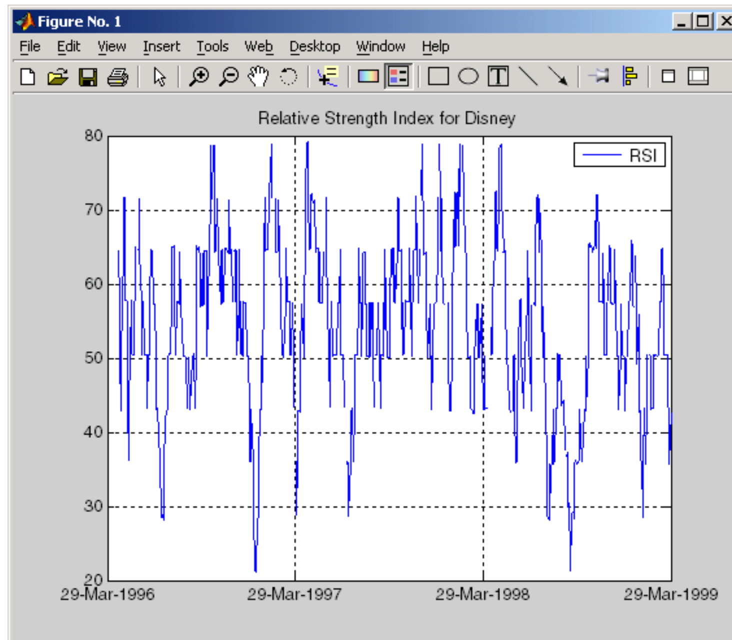
The parameter value is the string that represents the valid parameter name.

Note The relative strength index is calculated by dividing the sum of the closing values for the up days by the sum of the closing values for the down days: `RSI = sum(CLOSEP_up)/sum(CLOSEP_down)`.
Also, the first value of `RSI`, `RISI(1)`, is set as `NaN` to preserve the dimensions of `CLOSEP`.

Examples

Compute 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')
```



See Also

`negvolidx`, `posvolidx`

Reference

Murphy, John J., *Technical Analysis of the Futures Market*, New York Institute of Finance, 1986, pp. 295 - 302.

second

Purpose

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

Purpose 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`

setfield

Purpose 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, e.g., `'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);
```

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`

size

Purpose

Number of dates and data series

Syntax

```
szfts = size(tsobj)
szfts = size(tsobj, dim)
```

Arguments

`tsobj`	Financial time series object
`dim`	Dimension: `dim = 1` returns number of dates (rows). `dim = 2` 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.

See Also

`length`

`size` in the MATLAB documentation

Purpose Smooth data

Syntax

```
output = smoothts(input)
output = smoothts(input, 'b', wsize)
output = smoothts(input, 'g', wsize, stdev)
output = smoothts(input, 'e', n)
```

Arguments

`input`	A 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. `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`

Purpose

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` and `sortrows` in the MATLAB documentation

spctkd

Purpose

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)
`fastpctk`	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 `fastpctk` 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)` allows you to 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

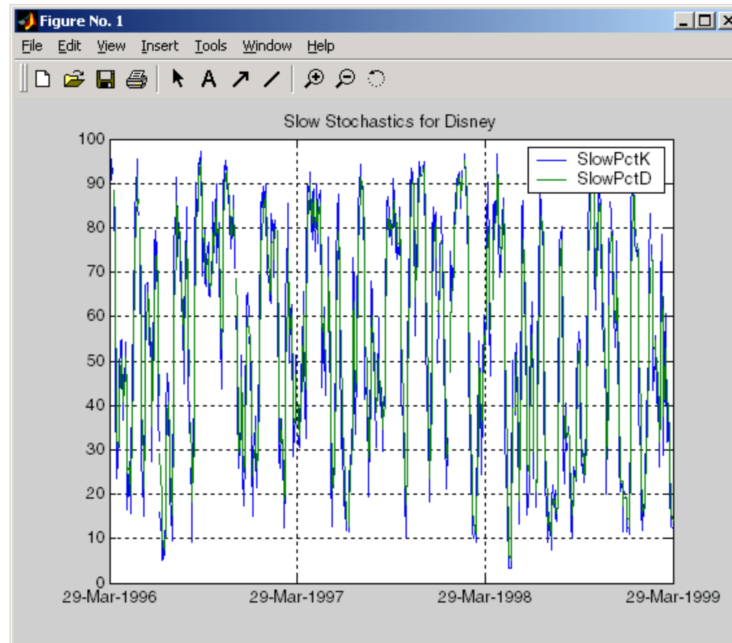
- `KName`: F%K series name
- `DName`: F%D series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute 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')
```



See Also

`fpctkd`, `stochosc`, `tsmovavg`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 268 - 271.

Purpose

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

Purpose

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. `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.

`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

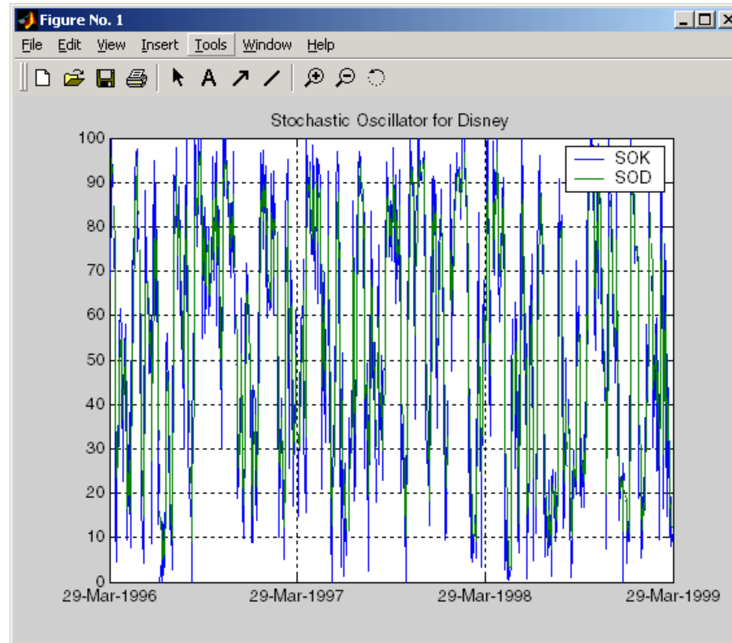
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the stochastic oscillator for Disney stock and plot the results:

```
load disney.mat
dis_StochOsc = stochosc(dis)
plot(dis_StochOsc)
title('Stochastic Oscillator for Disney')
```



See Also

`fpctkd`, `spctkd`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 268 - 271.

Purpose

Content assignment

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

Purpose

Subscripted reference

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

Purpose

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

Purpose 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 An investment has a 12% nominal rate of return and is taxed at a 30% rate. The after-tax rate of return is

```
Return = taxedrr(0.12, 0.30)

Return =
         0.0840
```

or 8.4%

See Also `effrr`, `irr`, `mirr`, `nomrr`, `xirr`

tbl2bond

Purpose

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 `datenum` 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 `DaysMaturity + 1`.
`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

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];
```

Execute the function.

```
TBond = tbl2bond(TBill)

TBond =
            0  729760       99.854       99.855     0.053
            0  729790       99.344       99.349     0.0544
            0  729820       98.942       98.946     0.054
```

(Example output has been formatted for readability.)

See Also

`tr2bonds` and other functions for Term Structure of Interest Rates

thirdwednesday

Purpose

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`.

Note
1. All dates are returned as serial date numbers. Convert to strings using `datestr`.
2. The function returns duplicates if you supply identical months and years.
3. 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

Purpose

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

Two bonds are quoted as 101-25 and 102-31. Convert these prices to decimal.

```
InNumber  = [101; 102];
InFraction = [25; 31]

OutNumber = thirtytwo2dec(InNumber, InFraction)

OutNumber =

    101.7813
    102.9688
```

See Also

`dec2thirtytwo`

Purpose

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 (e.g., 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 (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

Compute the 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

times

Purpose

Financial time series multiplication

Syntax

```
newfts = tsobj_1 .* tsobj_2
newfts = tsobj .* array
newfts = array .* tsobj
```

Arguments

`tsobj_1`, `tsobj_2`	A pair of financial time series objects
`array`	A scalar value or array with the number of rows equal to the number of dates in `tsobj` and the number of columns equal to the number of data series in `tsobj`

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 = tsobj_1 .* tsobj_2` multiplies financial time series objects element by element.

`newfts = tsobj .* array` multiplies a financial time series object element by element by an array.

`newfts = array .* tsobj` `newfts = array / tsobj` 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`

time2date

Purpose 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` (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:

`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, e.g. `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	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
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

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`

Purpose 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.

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`.

`newfts = toannual(oldfts, ParameterName, ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

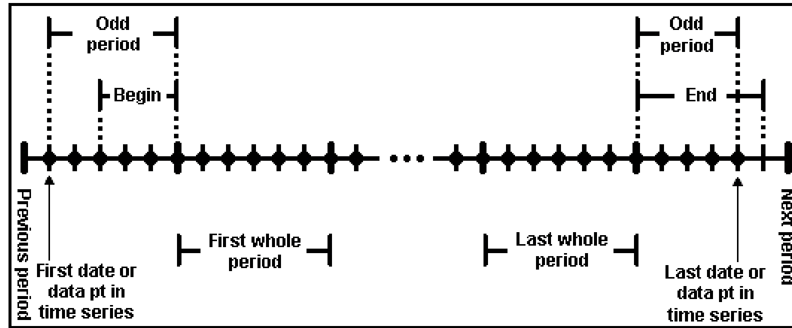
Parameter Name	Parameter Value	Description
`'CalcMethod'`	`'CumSum'`	Returns the cumulative sum of the values within each year. Data for missing dates are given the value `0`.
	`'Exact'`	Returns the exact value at the end-of-year date. No data manipulation occurs.
	`'Nearest'`	(Default) Returns the values located at the end-of-year dates. If there is missing data, `'Nearest'` returns the nearest data point preceding the end-of-year date.

Parameter Name	Parameter Value	Description
	'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.

Parameter Name	Parameter Value	Description
'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) Returns a financial time series that includes only NYSE business days.
'DateFilter'	'Absolute'	(Default) Returns all annual dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1.
	'Relative'	Returns only the annual dates that exist in oldfts. Some dates may be disregarded if BusDays = 1.
'ED'	0	Annual period ends on the last day or last business day of the month.

Parameter Name	Parameter Value	Description
	1 - 31	Specifies a particular annual day. Months that do not contain the specified day return the last day (or last business day) of the month (e.g., ED = 31 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).
'EndPtTol'	[Begin, End]	Denotes the minimum number of days that constitute an odd annual period at the end points of the time series (before the first time series date and after the last end-of-year date). Begin and End must be -1 or any positive integer greater than or equal to 0. A single value input for 'EndPtTol' is the same as specifying that single value for Begin and End. -1 Exclude odd annual period dates and data from calculations. 0 (Default) Include odd annual period dates and data in calculations. n Number of days (any positive integer) that constitute an odd annual period. If there are insufficient days for a complete year, the end point data is ignored.

Parameter Name	Parameter Value	Description
The following diagram is a general depiction of the factors involved in the determination of end points for this function. 		
'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.

See Also

`convertto`, `todaily`, `tomonthly`, `toquarterly`, `tosemi`, `toweekly`

Purpose

Convert to daily

Syntax

```
newfts = todaily(oldfts)
newfts = todaily(oldfts, ParameterName, ParameterValue, ...)
```

Arguments

`oldfts`	Financial time series object

Description

`newfts = todaily(oldfts)` converts a financial time series of any frequency to a daily frequency.

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`.

`newfts = todaily(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.

Note If you set `'CalcMethod'` to `'v21x'`, settings for all of the following parameter name/parameter value pairs are not supported.

Parameter Name	Parameter Value	Description
'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 includes only NYSE business days.
'DateFilter'	'Absolute'	(Default) Displays all daily dates between the start and end dates of `oldfts`. Some dates may be disregarded if `BusDays = 1`.
	'Relative'	Displays only dates that exist in `oldfts`. Some dates may be disregarded if `BusDays = 1`.
'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.

See Also

`convertto`, `toannual`, `tomonthly`, `toquarterly`, `tosemi`, `toweekly`

Purpose

Current date

Syntax

```
Datenum = today
```

Description

`Datenum = today` returns the current date as a serial date number.

Examples

```
Datenum = today
```

returns

```
Datenum =

        730695
```

on July 28, 2000.

See Also

`datenum`, `datestr`, `now`

todecimal

Purpose 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 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 Microsoft Excel.

See Also

`toquoted`

Purpose

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.

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`.

`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.
	`'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.

Parameter Name	Parameter Value	Description
	'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 tomonthly 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.

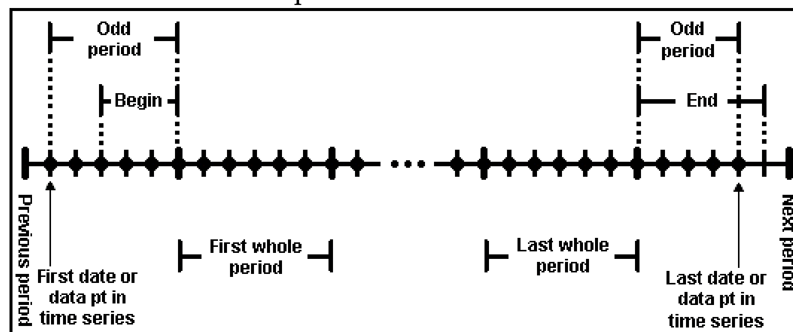
Note If you set 'CalcMethod' to 'v21x', settings for all of the following parameter name/parameter value pairs are not supported.

Parameter Name	Parameter Value	Description
'BusDays'	0	Generates a monthly financial time series that ranges from 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). If an end-of-month date falls on a nonbusiness day or NYSE holiday, returns the last business day of the month.
'DateFilter'	'Absolute'	(Default) Returns all monthly dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1.

Parameter Name	Parameter Value	Description
	'Relative'	Returns only monthly dates that exist in oldfts. Some dates may be disregarded if BusDays = 1.
'ED'	0	(Default) The end-of-month date is the last day (or last business day) of the month.
	1 - 31	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 (e.g., ED = 31 does not exist for February). If end-of-month falls on a NYSE non-business day or holiday, the previous business day is returned if BusDays = 1.

Parameter Name	Parameter Value	Description
'EndPtTol'	[Begin, End]	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). Begin and End must be -1 or any positive integer greater than or equal to 0. A single value input for 'EndPtTol' is the same as specifying that single value for Begin and End. -1 Do not include odd month dates and data in calculations. 0 (Default) Include all odd month dates and data in calculations. 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.

The following diagram is a general depiction of the factors involved in the determination of end points for this function.



Parameter Name	Parameter Value	Description
'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.

See Also

convertto, toannual, todaily, toquarterly, tosemi, toweekly

toquarterly

Purpose 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.

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`.

`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`.
	`'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.

Parameter Name	Parameter Value	Description
	'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`.

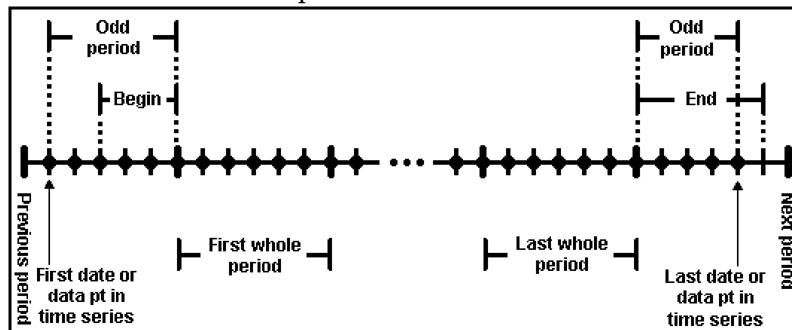
Note If you set `'CalcMethod'` to `'v21x'`, settings for all of the following parameter name/parameter value pairs are not supported.

Parameter Name	Parameter Value	Description
'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). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter.
'DateFilter'	'Absolute'	(Default) Returns all quarterly dates between the start and end dates of `oldfts`. Some dates may be disregarded if `BusDays = 1`.

Parameter Name	Parameter Value	Description
	'Relative'	Returns only quarterly dates that exist in oldfts. Some dates may be disregarded if BusDays = 1.
'ED'	0	(Default) The end-of-quarter date is the last day (or last business day) of the quarter.
	1 - 31	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 (e.g., ED = 31 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
'EndPtTol'	[Begin, End]	Denotes the minimum number of days that constitute a odd quarter at the end points 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 'EndPtTol' is the same as specifying that single value for Begin and End. -1 Do not include odd quarter dates and data in calculations. 0 (Default) Include all odd quarter dates and data in calculations. 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.

The following diagram is a general depiction of the factors involved in the determination of end points for this function.



Parameter Name	Parameter Value	Description
'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.

See Also

convertto, toannual, todaily, tomonthly, tosemi, toweekly

toquoted

Purpose 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 Microsoft Excel.

See Also `todecimal`

tosemi

Purpose

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.

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`.

`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.
	`'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.

Parameter Name	Parameter Value	Description
	`'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`.

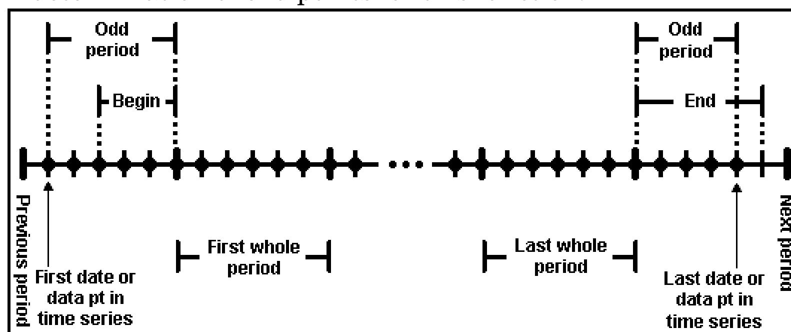
Note If you set `'CalcMethod'` to `'v21x'`, settings for all of the following parameter name/parameter value pairs are not supported.

Parameter Name	Parameter Value	Description
`'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). If an end-of-period date falls on a nonbusiness day or NYSE holiday, returns the last business day of the period.

Parameter Name	Parameter Value	Description
'DateFilter'	'Absolute'	(Default) Returns all semiannual dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1.
	'Relative'	Returns only semiannual dates that exist in oldfts. Some dates may be disregarded if BusDays = 1.
'ED'	0	(Default) The end-of-period date is the last day (or last business day) of the semiannual period.
	1 - 31	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 (e.g., ED = 31 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]	Denotes the minimum number of days that constitute an odd semiannual period at the end points 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 'EndPtTol' is the same as specifying that single value for Begin and End. -1 Do not include odd period dates and data in calculations. 0 (Default) Include all odd period dates and data in calculations. 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.

The following diagram is a general depiction of the factors involved in the determination of end points for this function.



Parameter Name	Parameter Value	Description
'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.

See Also

convertto, toannual, todaily, tomonthly, toquarterly, toweekly

totalreturnprice

Purpose 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`

toweekly

Purpose

Convert to weekly

Syntax

```
newfts = toweekly(oldfts)
newfts = toweekly(oldfts, ParameterName, ParameterValue, ...)
```

Arguments

`oldfts`	Financial time series object

Description

`newfts = toweekly(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.

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`.

`newfts = toweekly(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.
	`'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.

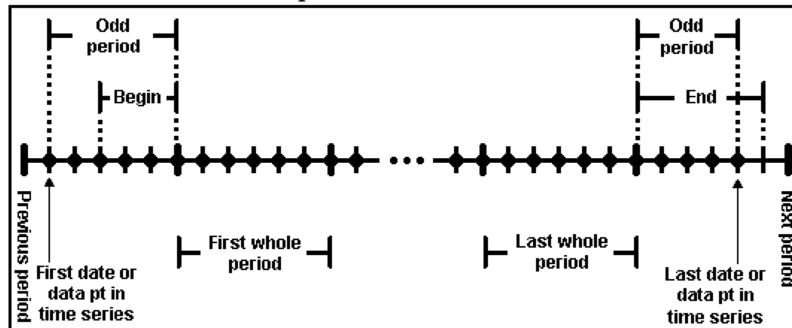
Parameter Name	Parameter Value	Description
	'SimpAvg'	Returns an averaged weekly value that only takes into account dates with data (nonNaN) within each week.
	'v21x'	This mode is compatible with previous versions of this function (Version 2.1.x and earlier). It returns an averaged end-of-weekly value 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.

Note If you set 'CalcMethod' to 'v21x', settings for all of the following parameter name/parameter value pairs are not supported.

Parameter Name	Parameter Value	Description
'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). If an end-of-week date falls on a nonbusiness day or NYSE holiday, returns the previous business day.
'DateFilter'	'Absolute'	(Default) Returns all weekly dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1.

Parameter Name	Parameter Value	Description
	'Relative'	Returns only end-of-week dates that exist in oldfts. Some dates may be disregarded if BusDays = 1.
'EndPtTol'	[Begin, End]	Denotes the minimum number of days that constitute a odd week at the end points 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 'EndPtTol' is the same as specifying that single value for Begin and End. -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 end points for this function.		
'EOW'	0 - 6	Specifies the end-of-week day: 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.



See Also

`convertto`, `toannual`, `todaily`, `tomonthly`, `toquarterly`, `tosemi`

tr2bonds

Purpose

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 `datenum` 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 `datestr` 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, always 0 (actual/actual).
`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. A 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. A 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

Given published 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];
```

Execute the function.

```
[Bonds, Prices, Yields] = tr2bonds(Matrix)
```

```
Bonds =

  729840    0.06125    100    2    0    1
  729966    0.0625     100    2    0    1
  730106    0.05125    100    2    0    1
  730225    0.065      100    2    0    1

Prices =

         100.1563
         100.3750
          99.5000
         101.0938

Yields =

         0.0546
         0.056
         0.0563
         0.0564
```

(Example output has been formatted for readability.)

See Also

`tbl2bond`, `zbtprice`, `zbtyield`, and other functions for Term Structure of Interest Rates

tsaccel

Purpose Acceleration between periods

Syntax

```
acc = tsaccel(data, nperiods, datatype)
accts = tsaccel(tsobj, nperiods, datatype)
```

Arguments

`data`	Data series
`nperiods`	(Optional) Number of periods. Default = 12.
`datatype`	(Optional) Indicates whether `data` 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.
`tsobj`	Name of an existing financial time series object

Description Acceleration is the difference of two momentums separated by some number of periods.

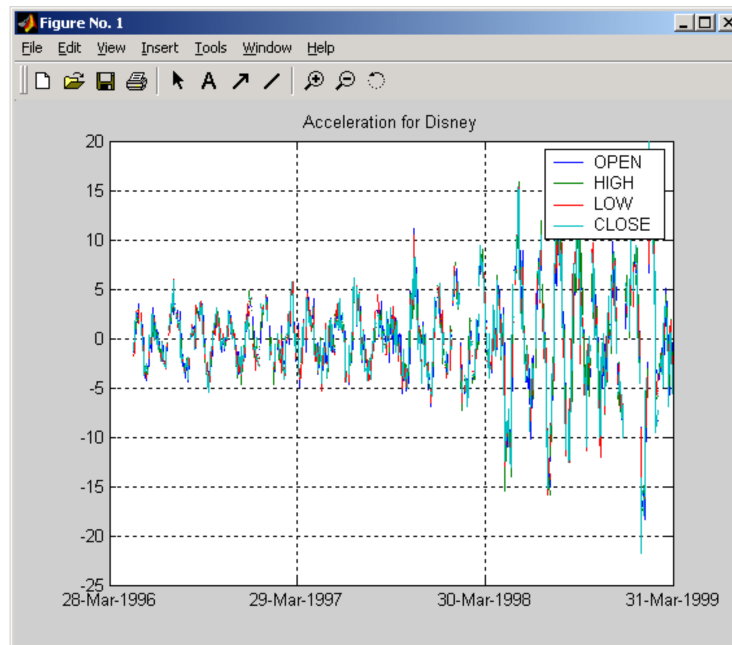
`acc = tsaccel(data, nperiods, datatype)` calculates the acceleration of a data series, essentially the difference of the current momentum with the momentum some number of periods ago. If `nperiods` is specified, `tsaccel` calculates the acceleration of a data series `data` with time distance of `nperiods` periods.

`accts = tsaccel(tsobj, nperiods, 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`.

Examples

Compute the acceleration 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')
```



See Also

`tsmom`

Reference

Kaufman, P. J., *The New Commodity Trading Systems and Methods*, New York: John Wiley & Sons, 1987.

Purpose Momentum between periods

Syntax

```
mom = tsmom(data, nperiods)
momts = tsmom(tsobj, nperiods)
```

Arguments

`data`	Data series. Column-oriented vector or matrix.
`nperiods`	(Optional) Number of periods. Default = 12.
`tsobj`	Financial time series object

Description Momentum is the difference between two prices (data points) separated by a number of periods.

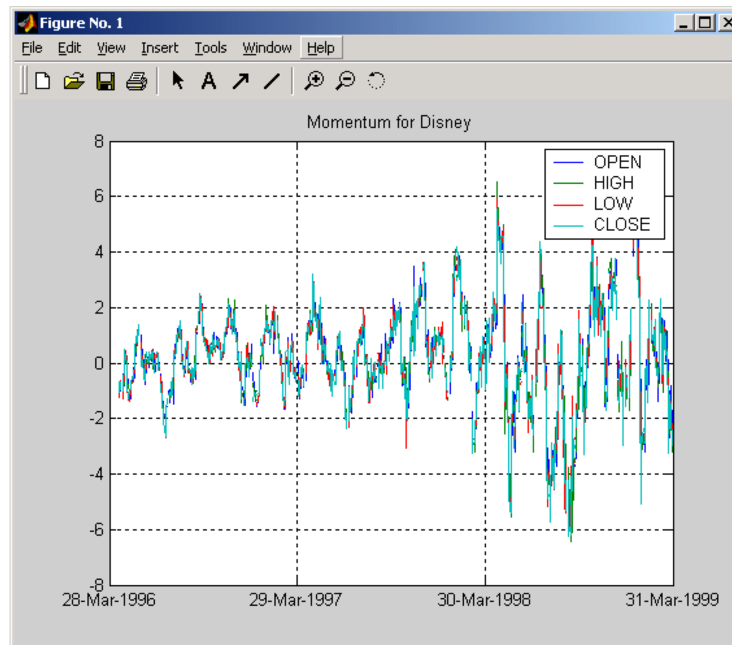
`mom = tsmom(data, nperiods)` calculates the momentum of a data series data. If `nperiods` is specified, `tsmom` uses that value instead of the default 12.

`momts = tsmom(tsobj, nperiods)` 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 `nperiods` is specified, `tsmom` uses that value instead of the default 12.

Examples

Compute the momentum 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')
```



See Also

`tsaccel`

Purpose

Moving average

Syntax

```
output = tsmovavg(tsobj, 's', lag)                    (Simple)
output = tsmovavg(vector, 's', lag, dim)
output = tsmovavg(tsobj, 'e', timeperiod)             (Exponential)
output = tsmovavg(vector, 'e', timeperiod, dim)
output = tsmovavg(tsobj, 't', numperiod)              (Triangular)
output = tsmovavg(vector, 't', numperiod, dim)
output = tsmovavg(tsobj, 'w', weights)                (Weighted)
output = tsmovavg(vector, 'w', weights, dim)
output = tsmovavg(tsobj, 'm', numperiod)              (Modified)
output = tsmovavg(vector, 'm', numperiod, dim)
```

Arguments

`tsobj`	Financial time series object
`lag`	Number of previous data points
`vector`	Row vector or row-oriented matrix. Each row is a set of observations.
`dim`	(Optional) Specifies dimension when `input` is a vector or matrix. Default = 2 (Row-oriented matrix: each row is a variable, and each column is an observation.). If `dim = 1`, input is assumed to be a column vector or column-oriented matrix (each column is a variable and each row an observation). `output` is identical in format to `input`.
`timeperiod`	Length of time period
`numperiod`	Number of periods considered
`weights`	Weights for each element in the window

Description

`output = tsmovavg(tsobj, 's', lag)` and
`output = tsmovavg(vector, 's', lag, dim)` compute the simple moving average. `lag` indicates the number of previous data points used in conjunction with the current data point when calculating the moving average.

`output = tsmovavg(tsobj, 'e', timeperiod)` and
`output = tsmovavg(vector, 'e', timeperiod, dim)` compute the exponential weighted moving average. 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/(timeperiod + 1)`).

`output = tsmovavg(tsobj, 't', numperiod)` and
`output = tsmovavg(vector, 't', numperiod, dim)` compute the triangular moving average. The triangular moving average double-smooths the data. `tsmovavg` calculates the first simple moving average with window width of `ceil(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)` and
`output = tsmovavg(vector, 'w', weights, dim)` calculate the weighted moving average 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)` and
`output = tsmovavg(vector, 'm', numperiod, dim)` calculate the modified moving average. 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.

See Also

`mean`, `peravg`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 184-192.

Purpose 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

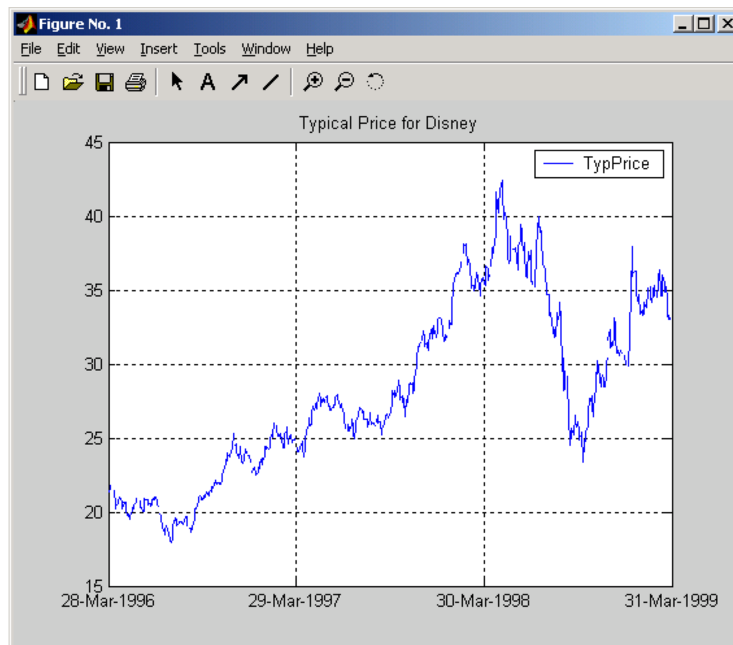
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the typical price for Disney stock and plot the results:

```
load disney.mat
dis_Typ = typprice(dis);
plot(dis_Typ)
title('Typical Price for Disney')
```



See Also

`medprice`, `wclose`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 291 - 292.

Purpose

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, i.e., the residuals or innovations (ε_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.

`P` Non-negative, 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 (κ) 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

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

where α represents the argument `Alpha`, β represents `Beta`, and the GARCH(P, Q) coefficients $\{\kappa, \alpha, \beta\}$ are subject to the following constraints.

$$\sum_{i=1}^{P} a_i + \sum_{j=1}^{Q} \beta_j < 1$$

$$\kappa > 0$$

$$a_i \geq 0 \qquad i = 1, 2, \ldots, P$$

$$\beta_j \geq 0 \qquad j = 1, 2, \ldots, Q$$

Note that `U` is a vector of residuals or innovations (ε_t) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

Note `ugarch` corresponds generally to the GARCH Toolbox function `garchfit`. The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information, see the *GARCH Toolbox User's Guide* 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.

See Also

`ugarchpred`, `ugarchsim`, and the GARCH Toolbox function `garchfit`

References

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

Purpose

Log-likelihood objective function of univariate GARCH(P,Q) processes with Gaussian innovations

Syntax

```
LogLikelihood = ugarchllf(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 κ 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, i.e., the residuals or innovations (ε_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.
`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 = ugarchllf(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.

`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

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

where α represents the argument `Alpha`, and β represents `Beta`.

`U` is a vector of residuals or innovations (ε_t) representing a mean-zero, discrete time stochastic process. Although σ_t^2 is generated via the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

Since `ugarchllf` 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 GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod/`.

See Also `ugarch`, `ugarchpred`, `ugarchsim`

Purpose 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, i.e., the residuals or innovations (ε_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.
`Kappa`	Scalar constant term κ 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` (i.e., ε_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 $\{\varepsilon_t\}$ 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

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

where α represents the argument `Alpha`, β represents `Beta`, and the GARCH(P,Q) coefficients $\{\kappa, \alpha, \beta\}$ are subject to the following constraints.

$$\sum_{i=1}^{P} a_i + \sum_{j=1}^{Q} \beta_j < 1$$

$$\begin{aligned} \kappa &> 0 & & \\ a_i &\geq 0 & \quad i &= 1, 2, \ldots, P \\ \beta_j &\geq 0 & \quad j &= 1, 2, \ldots, Q \end{aligned}$$

Note that `U` is a vector of residuals or innovations (ε_t) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

Note `ugarchpred` corresponds generally to the GARCH Toolbox function `garchpred`. The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see the *GARCH Toolbox User's Guide* 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`, and the GARCH Toolbox function `garchpred`

ugarchsim

Purpose Simulate univariate GARCH(P,Q) process with Gaussian innovations

Syntax

```
[U, H] = ugarchsim(Kappa, Alpha, Beta, NumSamples)
```

Arguments

`Kappa`	Scalar constant term κ 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 (ε_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 (i.e., `H(t)`) represents the time series inferred from the innovations time series $\{\varepsilon_t\}$ (i.e., `U`).

The time-conditional variance, σ_t^2, of a GARCH(P,Q) process is modeled as

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

where α represents the argument `Alpha`, β represents `Beta`, and the GARCH(P,Q) coefficients $\{\kappa, \alpha, \beta\}$ are subject to the following constraints.

$$\sum_{i=1}^{P} a_i + \sum_{j=1}^{Q} \beta_j < 1$$

$$\kappa > 0$$
$$a_i \geq 0 \qquad i = 1, 2, \ldots, P$$
$$\beta_j \geq 0 \qquad j = 1, 2, \ldots, Q$$

Note that `U` is a vector of residuals or innovations (ε_t) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

The output vectors `U` and `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 `U` and `H` such that the sum of the GARCH coefficients, excluding `Kappa`, raised to the `N`th power, does not exceed 0.01.

```
0.01 = (sum(Alpha) + sum(Beta))^N
```

Thus

```
N = log(0.01)/log((sum(Alpha) + sum(Beta)))
```

Note `ugarchsim` corresponds generally to the GARCH Toolbox function `garchsim`. The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see the *GARCH Toolbox User's Guide* 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 reproducability.

randn('seed', 10)

% 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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
End diagnostic information
                                  max                       Directional
 Iter    F-count       f(x)     constraint   Step-size       derivative     Procedure
    1       5       699.185       -0.125            1       -2.97e+006
    2      22       658.224      -0.1249     0.000488            -64.6
    3      28       610.181            0            1            -49.4
    4      35       590.888            0          0.5            -38.9
    5      42       583.961     -0.03317          0.5            -29.8
    6      49       583.224     -0.02756          0.5            -31.8
    7      57       582.947     -0.02067         0.25            -7.28
    8      63       578.182            0            1            -2.43
    9      71       578.138     -0.09145         0.25            -0.55
   10      77       577.898     -0.04452            1           -0.148
   11      84       577.882     -0.06128          0.5          -0.0488
   12      90       577.859     -0.07117            1        -0.000758
   13      96       577.858     -0.07033            1        -0.000305    Hessian modified
   14     102       577.858     -0.07042            1       -3.32e-005    Hessian modified
   15     108       577.858      -0.0707            1       -1.29e-006    Hessian modified
   16     114       577.858     -0.07077            1       -1.29e-007    Hessian modified
   17     120       577.858     -0.07081            1       -1.97e-007    Hessian modified
```

```
Optimization Converged Successfully
Magnitude of directional derivative in search direction
  less than 2*options.TolFun and maximum constraint violation
  is less than options.TolCon
No Active Constraints

Estimated Coefficients:
----------------------
0.2520
0.0708
0.1623
0.4000

Variance Forecasts:
------------------
1.3243
0.9594
0.9186
0.8402
0.7966
0.7634
0.7407
0.7246
0.7133
0.7054
```

See Also

`ugarch`, `ugarchpred`, and the GARCH Toolbox function `garchsim`

References

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

Purpose Unary minus of financial time series object

Syntax `uminus`

Description `uminus` implements unary minus for a financial time series object.

See Also `uplus`

uplus

Purpose Unary plus of financial time series object

Syntax `uplus`

Description `uplus` implements unary plus for a financial time series object.

See Also `uminus`

Purpose Concatenate financial time series objects vertically

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`

Purpose Volume rate of change

Syntax

```
vroc = volroc(tvolume nperiods)
vrocts = volroc(tsobj, nperiods)
vrocts = volroc(tsobj, nperiods, ParameterName, ParameterValue)
```

Arguments

`tvolume`	Volume traded
`nperiods`	(Optional) Period difference. Default = 12.
`tsobj`	Financial time series object

Description

`vroc = volroc(tvolume nperiods)` calculates the volume rate of change, `vroc`, from the volume traded data `tvolume`. If `nperiods` is specified, the volume rate of change is calculated between the current volume and the volume `nperiods` ago.

`vrocts = volroc(tsobj, nperiods)` 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 `nperiods` is specified, the volume rate of change is calculated between the current volume and the volume `nperiods` ago.

`vrocts = volroc(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

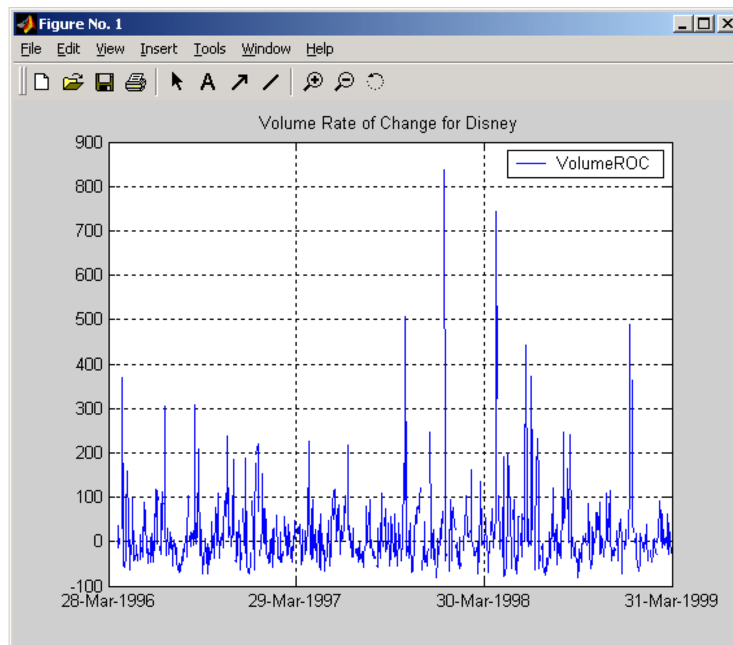
- `VolumeName`: volume traded series name

The parameter value is a string that represents the valid parameter name.

Examples

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')
```



See Also

`prcroc`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 310 - 311.

Purpose Weighted close

Syntax

```
wcls = wclose(highp, lowp, closep)
wcls = wclose([highp lowp closep])
wclsts = wclose(tsobj)
wclsts = wclose(tsobj, ParameterName, ParameterValue, ...)
```

Arguments

`highp`	High price (vector)
`lowp`	Low price (vector)
`closep`	Closing price (vector)
`tsobj`	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(tsobj)` computes the weighted close prices 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. `wclsts` is a financial time series object of the same dates as `tsobj` and contains the data series named `WClose`.

`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

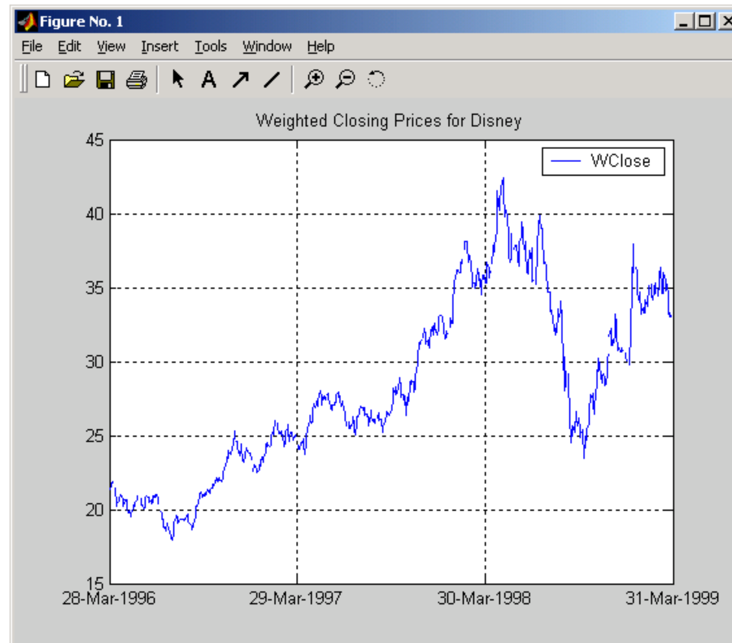
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the weighted closing prices for Disney stock and plot the results:

```
load disney.mat
dis_Wclose = wclose(dis)
plot(dis_Wclose)
title('Weighted Closing Prices for Disney')
```



See Also

`medprice`, `typprice`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 312 - 313.

Purpose Day of week

Syntax

```
[DayNum, DayString] = weekday(Date)
```

Description

`[DayNum, DayString] = weekday(Date)` returns the day of the week in numeric and string form given the date as a serial date number or date string. The days of the week have these values.

DayNum	DayString
1	Sun
2	Mon
3	Tue
4	Wed
5	Thu
6	Fri
7	Sat

Note This function now ships with basic MATLAB. It originally shipped only with the Financial Toolbox. This description remains here for your convenience.

Examples

```
[DayNum, DayString] = weekday(730845)
```

or

```
[DayNum, DayString] = weekday('25-Dec-2000')
```

returns

```
DayNum =

     2

DayString =

Mon
```

See Also

datenum, datestr, datevec, day

Purpose

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 `holdings2weights` 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

Purpose

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

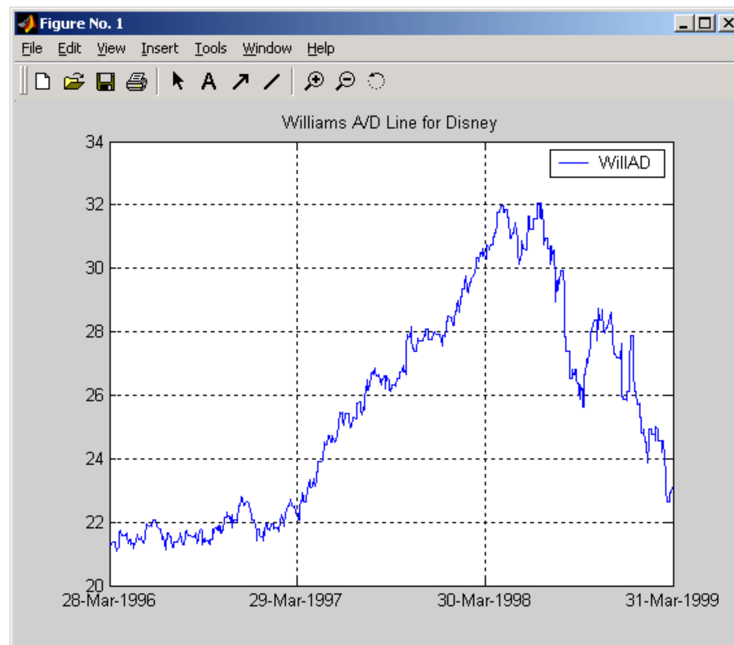
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Williams A/D line for Disney stock and plot the results:

```
load disney.mat
dis_Willad = willad(dis)
plot(dis_Willad)
title('Williams A/D Line for Disney')
```



See Also

`adline`, `adosc`, `willpctr`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 314 - 315.

willpctr

Purpose

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

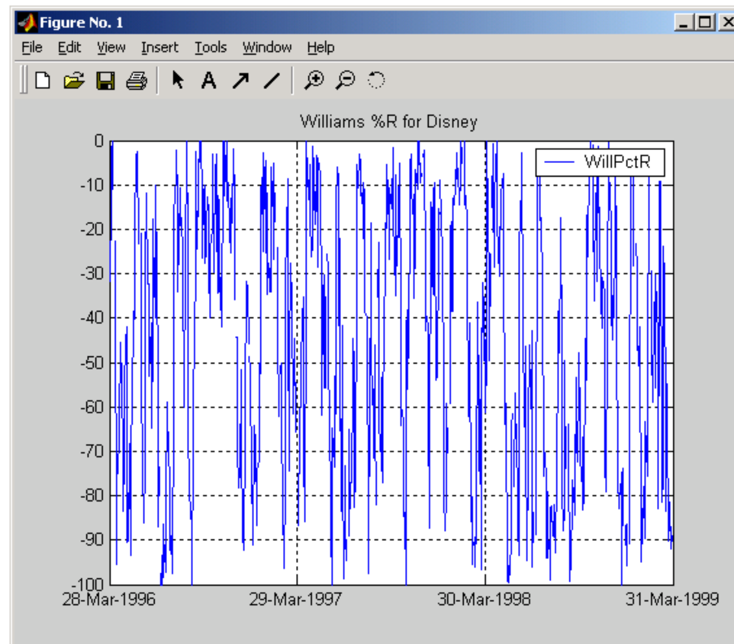
- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

Parameter values are the strings that represent the valid parameter names.

Examples

Compute the Williams %R values for Disney stock and plot the results:

```
load disney.mat
dis_Wpctr = willpctr(dis)
plot(dis_Wpctr)
title('Williams %R for Disney')
```



See Also

`stochosc`, `willad`

Reference

Achelis, Steven B., *Technical Analysis from A To Z*, Second printing, McGraw-Hill, 1995, pp. 316 - 317.

Purpose Number of working days between dates

Syntax

```
Days = wrkdydif(StartDate, EndDate, Holidays)
```

Description `Days = wrkdydif(StartDate, EndDate, Holidays)` returns the number of working days between dates `StartDate` and `EndDate`. `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

Purpose 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`	(Optional) Excel date system. A vector or scalar. When `Convention = 0` (default), the Excel 1900 date system is in effect. When `Convention = 1`, the Excel 1904 date system in used. 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.

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 693961 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with MATLAB Excel Link.

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

Purpose Internal rate of return for nonperiodic cash flow

Syntax

```
Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations)
```

Arguments

`CashFlow`	A vector of nonperiodic cash flows. Include the initial investment as the initial cash flow value (a negative number).
`CashFlowDates`	A vector of dates on which the cash flows occur. Enter dates as serial date numbers or date strings.
`Guess`	(Optional) Initial estimate of the expected return. Default = 0.1 (10%).
`MaxIterations`	(Optional) Number of iterations used by Newton's method to solve for `Return`. Default = 50.

Description `Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations)` returns the internal rate of return for a schedule of nonperiodic cash flows.

Examples An investment of $10,000 returns this nonperiodic cash flow. The original investment and its date are included.

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 internal rate of return for this nonperiodic cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
CashFlowDates = ['01/12/2000'
                 '02/14/2001'
                 '03/03/2001'
                 '06/14/2001'
                 '12/01/2001'];
```

```
Return = xirr(CashFlow, CashFlowDates)
```

returns

```
Return =
        0.1009 (or 10.09%)
```

See Also

`fvvar`, `irr`, `mirr`, `pvvar`

References

Sharpe and Alexander, *Investments*, 4th edition, page 463.

year

Purpose

Year of date

Syntax

```
Year = year(Date)
```

Description

`Year = year(Date)` returns the year of a serial date number or a date string.

Examples

```
Year = year(731798.776)
```

or

```
Year = year('05-Aug-2003')
```

returns

```
Year =

       2003
```

See Also

`datevec`, `day`, `month`, `yeardays`

yeardays

Purpose 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. 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).

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

Purpose Fraction of year between dates

Syntax

```
Fraction = yearfrac(StartDate, EndDate, Basis)
```

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`EndDate`	Enter as serial date numbers or date strings.
`Basis`	(Optional) 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 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

All specified arguments must be number of instruments (`NUMINST`) by 1 or 1-by-`NUMINST` conforming vectors or scalar arguments.

Description `Fraction = yearfrac(StartDate, EndDate, Basis)` returns a fraction based on the number of days between dates `StartDate` and `EndDate` using the given day-count basis. If `EndDate` is earlier than `StartDate`, `Fraction` is negative.

Examples

```
Fraction = yearfrac('14 mar 01', '14 sep 01', 0)

Fraction =

    0.5041

Fraction = yearfrac('14 mar 01', '14 sep 01', 1)

Fraction =

    0.5000
```

See Also `days360`, `days365`, `daysact`, `daysdif`, `months`, `wrkdydif`, `yeardays`

Purpose 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 or equal to `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. 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).

Description `Yield = ylddisc(Settle, Maturity, Face, Price, Basis)` finds the yield of a discounted security.

Examples Using the data

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Price = 96.28;
Basis = 2;

Yield = ylddisc(Settle, Maturity, Face, Price, Basis)
```

returns

```
Yield =

        0.0903  (or 9.03%)
```

See Also `acrudisc`, `bndprice`, `bndyield`, `prdisc`, `yldmat`, `yldtbill`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 1.

yldmat

Purpose 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 or equal to `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. 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).

Description `Yield = yldmat(Settle, Maturity, Issue, Face, Price, CouponRate, Basis)` returns the yield of a security paying interest at maturity.

Examples Using the data

```
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)
```

returns

```
Yield =
```

`0.0607` (or 6.07%)

See Also

`acrubond`, `bndprice`, `bndyield`, `prmat`, `ylddisc`, `yldtbill`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 3.

yldtbill

Purpose 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 or equal to `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 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. Using this data

```
Yield = yldtbill('2/10/2000', '8/6/2000', 1000, 981.36)
```

returns

```
Yield =

        0.0384  (or 3.84%)
```

See Also `beytbill`, `bndyield`, `prtbill`, `yldmat`

References Bodie, Kane, and Marcus, *Investments*, pages 41-43.

Purpose

Zero curve bootstrapping from coupon bond data given price

Syntax

```
[ZeroRates, CurveDates] = zbtprice(Bonds, Prices, 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 `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`	(Optional) Day-count basis of the bond: 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).

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	A 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) A 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. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

`ZeroRates` 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.

If more than one bond has the same maturity date, `zbtprice` returns the mean zero rate for that maturity.

`CurveDates` 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.

Examples

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

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

which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date*.

```
ZeroRates =

               0.0616
               0.0609
               0.0658
               0.0590
               0.0648
               0.0655*
               0.0606
               0.0601
               0.0642
               0.0621
               0.0627

CurveDates =
```

729907 (serial date number for 01-Jun-1998)
730364 (01-Sep-1999)
730439 (15-Nov-1999)
730500 (15-Jan-2000)

```
730667  (30-Jun-2000)
730668  (01-Jul-2000)*
730971  (30-Apr-2001)
731032  (30-Jun-2001)
731033  (01-Jul-2001)
731321  (15-Apr-2002)
731336  (30-Apr-2002)
```

See Also

`zbtyield` and other functions for Term Structure of Interest Rates

References

Fabozzi, Frank J. "The Structure of Interest Rates." Ch. 6 in Fabozzi, Frank J. and T. Dessa 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.

zbtyield

Purpose

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 `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`	(Optional) Day-count basis of the bond. 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).

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	A 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) A 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. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

`ZeroRates` 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.

If more than one bond has the same maturity date, `zbtyield` returns the mean zero rate for that maturity.

`CurveDates` 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 `datestr` to convert serial date numbers to date strings.

Examples

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

```
[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle,...
OutputCompounding)
```

which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date*.

```
ZeroRates =

    0.0616
    0.0575
    0.0692
    0.0613
    0.0616
    0.0596*
    0.0606
    0.0659
    0.0650
    0.0607
    0.0628

CurveDates =
```

729907	(serial date number for 01-Jun-1998)
730364	(01-Sep-1999)
730439	(15-Nov-1999)

```
730500
730667
730668
730971
731032
731033
731321
731336
```

730500	(15-Jan-2000)
730667	(30-Jun-2000)
730668	(01-Jul-2000)*
730971	(30-Apr-2001)
731032	(30-Jun-2001)
731033	(01-Jul-2001)
731321	(15-Apr-2002)
731336	(30-Apr-2002)

See Also

`zbtprice` and other functions for Term Structure of Interest Rates

References

Fabozzi, Frank J. "The Structure of Interest Rates." Ch. 6 in Fabozzi, Frank J. and T. Dessa 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.

Purpose Discount curve given zero curve

Syntax

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates, Settle,
   Compounding, Basis)
```

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; i.e., the settlement date for the bonds from which the zero curve was bootstrapped.
`Compounding`	(Optional) A 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. 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).

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

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 was compounded daily on an actual/365 basis.

```
InputCompounding = 365;
InputBasis = 3;
```

Execute the function

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates,...
Settle, Compounding, Basis)
```

which returns the discount curve `DiscRates` at the maturity dates `CurveDates`.

```
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 computed them at full precision. If you enter `ZeroRates` as shown, `DiscRates` may differ due to rounding.

See Also

`disc2zero` and other functions for Term Structure of Interest Rates

zero2fwd

Purpose

Forward curve given zero curve

Syntax

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,
   Settle, Compounding, Basis
```

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`. 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 zero rates.
`Settle`	A serial date number that is the common settlement date for the zero rates.
`Compounding`	(Optional) A 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. 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).

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

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

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,...
Settle, Compounding)
```

which returns the forward rate curve `ForwardRates` at the maturity dates `CurveDates`.

```
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 computed them at full precision. If you enter `ZeroRates` as shown, `ForwardRates` may differ due to rounding.

See Also

`fwd2zero` and other functions for Term Structure of Interest Rates

Purpose

Par yield curve given zero curve

Syntax

```
[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle,
     Compounding, Basis, OutputCompounding)
```

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) A scalar that sets the rate at which the implied 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 implied zero rates. 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).
`OutputCompounding`	(Optional) Value representing the rate at which the par rates are compounded. Default = `Compounding`.

zero2pyld

Description

`[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle, Compounding, Basis, OutputCompounding)` 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');
Compounding = 1;
OutputCompounding = 12;

[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates,...
Settle, Compounding, [] , OutputCompounding)

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 computed them at full precision. If you enter `ZeroRates` as shown, `ParRates` may differ due to rounding.

See Also

`pyld2zero` and other functions for Term Structure of Interest Rates

A

Bibliography

"Bond Pricing and Yields" on page A-2

"Term Structure of Interest Rates" on page A-2

"Derivatives Pricing and Yields" on page A-2

"Portfolio Analysis" on page A-3

"Other References" on page A-3

For the well-known algorithms and formulas used in the Financial Toolbox (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:

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.

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, the Financial Toolbox 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:

Fabozzi, Frank J. "The Structure of Interest Rates." Ch. 6 in Fabozzi, Frank J. and T. Dessa Fabozzi, eds. *The Handbook of Fixed Income Securities.* 4th ed. New York: Irwin Professional Publishing, 1995, ISBN 0-7863-0001-9.

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, ISBN 1-55738-542-4.

Derivatives Pricing and Yields

The pricing and yield formulas for derivative securities come from:

Chriss, Neil A. "Black-Scholes and Beyond: Option Pricing Models," Chicago: Irwin Professional Publishing, 1997, ISBN 0-7863-1025-1.

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 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:

Bodie, Zvi, Alex Kane, and Alan J. Marcus, *Investments*, Burr Ridge, IL: Irwin. 2nd. ed., 1993, ISBN 0-256-08342-8.

To solve the quadratic minimization problem associated with finding the efficient frontier, the toolbox uses the `fmincon` function (finds the constrained minimum of a function of several variables) in the Optimization Toolbox. See that toolbox documentation for more details.

Financial Statistics

The discussion of computing statistical values for portfolios containing missing data elements derives from the following references:

Little, Roderick J. A. and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

Meng, Xiao-Li and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

Sexton, Joe and Anders Rygh Swensen, “ECM Algorithms That Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

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.

Other References

Other references include:

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.

Brealey, Richard A., and Stewart C. Myers. *Principles of Corporate Finance,* New York: McGraw-Hill. 4th ed., 1991, ISBN 0-07-007405-4.

Daigler, Robert T. *Advanced Options Trading.* Chicago: Probus Publishing Co., 1994, ISBN 1-55738-552-1.

A Dictionary of Finance. Oxford: Oxford University Press., 1993, ISBN 0-19-285279-5.

Fabozzi, Frank J., and T. Dessa Fabozzi, eds. *The Handbook of Fixed-Income Securities.* Burr Ridge, IL: Irwin. 4th ed., 1995, ISBN 0-7863-0001-9.

Fitch, Thomas P. *Dictionary of Banking Terms.* Hauppauge, NY: Barron's. 2nd ed., 1993, ISBN 0-8120-1530-4.

Hill, Richard O., Jr. *Elementary Linear Algebra.* Orlando, FL: Academic Press. 1986, ISBN 0-12-348460-X

Luenberger, David G., *Investment Science*, Oxford University Press, 1998. ISBN 0195108094

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.

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`.

Sharpe, William F., and Gordon J. Alexander. *Investments.* Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990, ISBN 0-13-504382-4.

Stigum, Marcia, with Franklin Robinson. *Money Market and Bond Calculations.* Richard D. Irwin., 1996, ISBN 1-55623-476-7.

Glossary

Active return	Amount of return achieved in excess of the return produced by an appropriate benchmark (e.g., 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, semi-annually, or annually.
Arbitrage	The purchase of securities on one market for immediate resale on another market in order to profit from a price or currency discrepancy.
Basis point	One hundredth of one percentage point, or 0.0001.
Beta	The price volatility of a financial instrument relative to the price volatility of a market or index as a whole. Beta is most 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.
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 Binomial model.

Call — **a.** An option to buy a certain quantity of a stock or commodity for a specified price within a specified time. See 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.

Callable bond — 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; i.e., the holder has sold a call option to the issuer. See Puttable bond.

Candlestick chart — 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.

Cap — Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain level.

Cash flow — Cash received and paid over time.

Collar — 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.

Convexity — 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.

Correlation — The simultaneous change in value of two random numeric variables.

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 semi-annually.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 semi-annually.

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.
Delta	The rate of change of the price of a derivative security relative to the price of the underlying asset; i.e., 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 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.
Discount curve	The curve of discount rates vs. maturity dates for bonds.
Drawdown	The peak to trough decline during a specific record period of an investment or fund.
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. See Macaulay duration.
Efficient frontier	A graph representing a set of portfolios that maximizes expected return at each level of portfolio risk. See Markowitz model.
Elasticity	See Lambda.
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 uncertaintity 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 vs. 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; i.e., the second derivative of the option price relative to the security price.
Greeks	Collectively, “greeks” refer to the financial measures delta, gamma, lambda, rho, theta, and vega, which are sensitivity measures used in evaluating derivatives.
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.
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 Yield to maturity.
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 listed in the “Bibliography”.
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.

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 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; i.e., divided by (1+rate/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 pro-rated according to how long the bond is held during that period.

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.; i.e., the future value at a known period in time discounted by the interest rate over that time period.

Principal value — See Par value.

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 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; i.e., the holder has bought a put option. See 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.

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; i.e., when a buyer actually pays for a security. It need not coincide with the issue date.

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 Long position.

Spot curve, spot yield curve See Zero 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.

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.

Strike Exercise a put or call option.

Strike price See Exercise price.

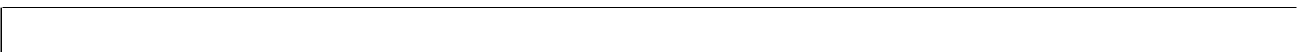
Swap A contract between two parties to exchange cash flows in the future according to some formula.

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 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 semi-annually 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.
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 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 for loanable funds across a continuum of durations and maturities. Also known as spot curve or spot yield curve.

Zero-coupon bond, or Zero 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.



Index

Numerics

1900 date system 10-329, 10-552
1904 date system 10-329, 10-552
360-day year 10-183
365-day year 10-190

A

abs2active 10-20
acceleration 10-515
accrued interest 2-22, 10-25, 10-27
  computing fractional period 10-23
acrubond 10-25
acrudisc 10-27
active return 3-20
active risk 3-20
active2abs 10-28
actual days
  between dates 10-191
adding a scalar and a matrix 1-8
adding matrices 1-7
adline 10-30
adosc 10-33
advance payments, periodic payment given 10-372
after-tax rate of return 10-471
algebra, linear 1-8, 1-13
American options 2-3, 2-37
amortization 1-21, 2-19, 2-20, 10-35
amortize 10-35
analysis models for equity derivatives 2-35
analysis, technical 9-2
analyzing
  and computing cash flows 2-17
  equity derivatives 2-34
  portfolios 2-38
annuity 2-19
  payment of with odd first period 10-373
  periodic interest rate of 10-38
  periodic payment of loan or 10-374
annurate 10-38
annuterm 10-39
apostrophe or prime character (') 1-6
arguments
  function return 1-20
  interest rate 1-21
  matrices as, limitations 1-21
  vectors as, limitations 1-21
arithmetic 7-15
array operations 1-16
ASCII character 1-19
ascii2fts 10-40
  creating object with 6-13
asset covariance matrix with exponential weighting 10-246
asset life 1-21
axes
  combining 6-23
axis labels, converting 10-166

B

bank format 10-163
bar 10-44
bar3 10-47
bar3h 10-47
barh 10-44
base date 10-173

basis 2-22
basis, day-count 10-194
beytbill 10-50
binomial
  functions 2-3
  model 2-36
  put and call pricing 10-51
  tree, building 2-37
binprice 10-51
Black's option pricing 10-55
Black-Scholes
  elasticity 10-62
  functions 2-3
  implied volatility 10-60
  model 2-35
  options 5-21, 5-23
  put and call pricing 10-64
  sensitivity to
    interest rate change 10-66
    time-until-maturity change 10-68
    underlying delta change 10-59
    underlying price change 10-57
    underlying price volatility 10-70
blkimpv 10-53
blkprice 10-55
blsdelta 10-57
blsgamma 10-59
blsimpv 10-60
blslambda 10-62
blsprice 10-64
blsrho 10-66
blstheta 10-68
blsvega 10-70
bndconvp 10-71
bndconvy 10-74
bnddurp 10-77
bnddury 10-80
bndprice 10-83
bndspread 10-86
bndyield 10-91
bolling 10-94
bollinger 10-96
Bollinger band chart 2-15
bond
  convexity 5-3
  duration 5-3
  equivalent yield for Treasury bill 10-50
  portfolio
    constructing to hedge against duration and convexity 5-6
    visualizing sensitivity of price to parallel shifts in the yield curve 5-8
  sensitivity of prices to changes in interest rates 5-3
  zero-coupon 10-564
bootstrapping 2-32, 10-512, 10-563, 10-568
boxcox 10-98
  example 7-20
building a binomial tree 2-37
busdate 10-100
busdays 10-102
business date
  last of month 10-318
business day
  next 2-10, 10-100
  previous 10-100
business days 10-311

C

call and put pricing
  Black-Scholes 10-64
candle 10-104
candle (time series) 10-105

candlestick chart 10-104
capital allocation line 3-3
cash flow
  analyzing and computing 2-17
  convexity 10-112
  dates 2-11, 10-113
  duration 10-116
  future value of varying 10-290
  internal rate of return 10-310
  internal rate of return for nonperiodic 10-554
  irregular 10-290
  modified internal rate of return 10-345
  negative 2-17
  portfolio form of amounts 10-117
  present value of varying 10-434
  sensitivity of 2-19
  uniform payment equal to varying 10-375
cell array 5-16
`cfamounts` 10-107
`cfconv` 10-112
`cfdates` 10-113
`cfdur` 10-116
`cfport` 10-117
`cftimes` 10-120
`chaikosc` 10-122
`chaikvolat` 10-124
character array
  strings stored as 1-19
character, ASCII 1-19
chart
  Bollinger band 2-15
  candlestick 10-104
  high, low, open, close 10-302
  leading and lagging moving averages 10-349
  point and figure 10-391
`chartfts` 10-127
  combine axes feature 6-23
  purpose 6-17
  using 6-17
`chartfts` zoom feature 6-20
charting 9-2
charting financial data 2-12
`chfield` 10-129
colon (`:`) 1-6
Combine Axes tool 6-23
commutative law 1-8, 1-13
compatible time series 7-15
component 7-3
computing
  cash flows 2-17
  dot products of vectors 1-10
  yields for fixed-income securities 2-21
constraint functions 3-14
constraint matrix 3-17
constructing
  a bond portfolio to hedge against duration and convexity 5-6
  greek-neutral portfolios of European stock options 5-12
conventions
  SIA 2-21
conversions
  currency 2-12
  date input 2-5
  date output 2-7
`convert2sur` 10-130
converting
  and handling dates 2-4
  axis labels 10-166
`convertto` 10-132
convexity 5-3
  cash flow 10-112
  constructing a bond portfolio to hedge against 5-6

portfolio 5-4, 5-6
corr2cov 10-133
coupon bond
prices to zero curve 10-563
yields to zero curve 10-568
coupon date
after settlement date 10-138
days between 10-152, 10-155
coupon dates 2-28
coupon payments remaining until maturity 10-135
coupon period
containing settlement date 10-158
fraction of 10-22
coupons payable between dates 10-135
cov2corr 10-134
covariance matrix 3-5
covariance matrix with exponential weighting 10-246
cpncount 10-135
cpndaten 10-138
cpndatenq 10-141
cpndatep 10-145
cpndatepq 10-148
cpndaysn 10-152
cpndaysp 10-155
cpnpersz 10-158
cumsum 10-161
cur2frac 10-162
cur2str 10-163
currency
converting 2-12
decimal 10-273
formatting 2-12
fractional 10-162, 10-273
values 10-162
current date 10-489
and time 2-8, 10-366

D

data extraction 7-3
data series vector 7-3
data transformation 7-19
date
base 10-173
components 10-179
conversions 2-5
current 2-8, 10-366, 10-489
end of month 10-244
first business, of month 10-250
formats 2-4
hour of 10-309
input conversions 2-5
last date of month 10-244
last weekday in month 10-327
maturity 2-22
minute of 10-344
number 2-4, 10-173
displaying as string 10-168
Excel to MATLAB 10-552
indices of in matrix 10-169
MATLAB to Excel 10-329
of day in future or past month 10-170
of future or past workday 10-181
output conversions 2-7
seconds of 10-448
starting, add month to 10-170
string 2-4, 10-176
vector 10-179
year of 10-556
date 2-8
date of specific weekday in month 10-367
date string 7-7

indexing 7-8
range 7-9
date system
1900 10-329, 10-552
1904 10-329, 10-552
date vector 7-3
`date2time` 10-164
`dateaxis` 10-166
`datedisp` 10-168
`datefind` 10-169
`datemnth` 10-170
`datenum` 10-173
dates
actual days between 10-191
business days 10-311
cash-flow 2-11, 10-113
coupon 2-28
days between 10-183, 10-190, 10-191, 10-192, 10-194
determining 2-9
first coupon 2-21
fraction of year between 10-558
handling and converting 2-4
investment horizon 2-32
issue 2-21
last coupon 2-21
number of months between 10-348
quasi-coupon 2-21
settlement 2-21
vector of 1-20
working days between 10-551
`datestr` 7-7, 10-176
`datevec` 10-179
`datewrkdy` 10-181
day
date of specific weekday in month 10-367
of month 10-182
of month, last 10-245
of the week 10-543
`day` 10-182
day-count basis 10-194
day-count convention 2-22
days
between
coupon date and settlement date 10-155
dates 10-183, 10-190, 10-191, 10-192, 10-194, 10-551
settlement date and next coupon date 10-152
business 10-311
holidays 10-306
in coupon period containing settlement date 10-158
last business date of month 10-318
last weekday in month 10-327
nontrading 10-306
number of, in year 10-557
`days360` 10-183
`days360e` 10-184
`days360isda` 10-186
`days360psa` 10-188
`days365` 10-190
`daysact` 10-191
`daysadd` 10-192
`daysdif` 10-194
`dec2thirtytwo` 10-196
decimal currency 10-273
to fractional currency 10-162
declining-balance depreciation
fixed 2-19, 10-197
general 2-19, 10-198
default values 7-3
definitions 1-4
delta 2-34

change, Black-Scholes sensitivity to underlying 10-59
demonstration program 7-24
`depfixdb` 10-197
`depgendb` 10-198
`deprdv` 10-199
depreciable value, remaining 10-199
depreciation 2-19
  fixed declining-balance 2-19, 10-197
  general declining-balance 2-19, 10-198
  straight-line 2-19, 10-201
  sum of years' digits 2-19, 10-200
`depsoyd` 10-200
`depstln` 10-201
derivatives
  equity, pricing and analyzing 2-34
  sensitivity measures for 2-34
description field
  component name 7-3
  setting 6-13
determining dates 2-9
`diff` 10-202
`disc2zero` 10-203
discount curve
  from zero curve 10-573
  to zero curve 10-203
discount rate of a security 10-206
discount security 10-27
  future value of 10-288
  price of 10-426
  yield of 10-559
`discrate` 10-206
dividing matrices 1-13
dot products of vectors 1-10
double-colon operator 7-9
duration
  cash-flow and modified 10-116
  constructing a bond portfolio to hedge against 5-6
  for fixed-income securities 2-30
  Macaulay 2-30
  modified 2-30
  portfolio 5-4, 5-6

E

ECM (expectation conditional maximization) 10-231
`ecmlsrmle` 10-207
`ecmlsrobj` 10-211
`ecmmvnrfish` 10-213
`ecmmvnrmle` 10-216
`ecmmvnrobj` 10-220
`ecmmvnrstd` 10-222
`ecmnfish` 10-224
`ecmnhess` 10-226
`ecmninit` 10-228
`ecmnmle` 10-230
`ecmnobj` 10-236
`ecmnstd` 10-237
effective rate of return 10-239
efficient frontier 3-5
  plotting an 5-19
  tracking error 3-20
`effrr` 10-239
elasticity
  Black-Scholes 10-62
element-by-element 1-7
  operating 1-16
elements, referencing matrix 1-4
`emaxdrawdown` 10-240
`end` 10-242
  MATLAB variable 7-12
end-of-month rule 2-24

enlarging matrices 1-5
eomdate 10-244
eomday 10-245
equal time series 7-15
equations
  solving simultaneous linear 1-13
equity derivatives 2-34
  analysis models for 2-35
European options 2-3
  constructing greek-neutral portfolios of 5-12
ewstats 10-246
Excel date number
  from MATLAB date number 10-329
  to MATLAB date number 10-552
exp 10-248
expectation conditional maximization 10-231
exponential weighting of covariance matrix 10-246
extfield 10-249
extracting data 7-3

F

fbusdate 10-250
fetch 10-252
fieldnames 10-256
fillts 10-257
  example 8-9
filter 10-262
financial data
  charting 2-12
fints 10-263
  syntaxes 6-3
first business date of month 10-250
first coupon date 2-21
fixed declining-balance depreciation 2-19, 10-197
fixed periodic payments
  future value with 10-289
fixed-income securities
  cash-flow dates 10-113
  Macaulay and modified durations for 2-30
  pricing 2-29
  pricing and computing yields for 2-21
  terminology 2-21
  yield functions for 2-29
fixed-income sensitivities 2-30
formats
  bank 10-163
  date 2-4
formatting currency and charting financial data 2-12
forward curve
  from zero curve 10-576
  to zero curve 10-292
fpctkd 10-270
frac2cur 10-273
fraction of
  coupon period 10-22
  year between dates 10-558
fractional currency 10-162, 10-273
frequency
  indicator field 7-3
  indicators 6-12
  setting 6-12
frequency conversion functions
  Data menu 8-11
  table 7-19
frontcon 3-5, 10-276
frontier
  plotting an efficient 5-19
frontier 10-279
frontier, efficient 3-5
fts2ascii 10-281
fts2mat 10-282

`ftsbound` 10-283
  displaying dates with 7-10
`ftsdata` subdirectory 6-14
`ftsgui` 10-284
  command 8-2
`ftsinfo` 10-285
`ftstomtx` 10-295
`ftsuniq` 10-287
function
  return arguments 1-20
future month, date of day in 10-170
future value 2-18, 10-39
  of discounted security 10-288
  of varying cash flow 10-290
  with fixed periodic payments 10-289
`fvdisc` 10-288
`fvfix` 10-289
`fvvar` 10-290
`fwd2zero` 10-292

G

gamma 2-34
general declining-balance depreciation 2-19, 10-198
generating and referencing matrix elements 1-6
`getfield` 10-295
graphical user interface 8-2
graphics
  producing 5-19
  three-dimensional 5-12
greek-neutral portfolios, constructing 5-12
greeks 2-34
  neutrality 5-12
GUI 8-2
  starting with `ftsgui` 10-284

H

handling and converting dates 2-4
hedging 5-3
  a bond portfolio against duration and convexity 5-6
`hhigh` 10-298
high, low, open, close chart 10-302
`highlow` 10-300, 10-302
`hist` 10-303
`holdings2weights` 10-305
holidays 2-10
`holidays` 10-306
holidays and nontrading days 10-306
`horzcat` 10-307
`hour` 10-309
hour of date or time 10-309

I

identity matrix 1-13
iid (independent identically-distributed data) 10-228
implied volatility 2-35
  Black-Scholes 10-60
independent identically-distributed data 10-228
indexing
  date range 7-9
  date string 7-8
  integer 7-10
  with time-of-day data 7-12
indices
  of date numbers in matrix 10-169
  of nonrepeating integers in matrix 10-169
indifference curve 3-8
inner dimension rule 1-8
input
  conversions 2-5

string 1-19
interest 10-35
accrued 10-25, 10-27
on loan 2-19
interest rate swap 5-16
interest rates
arguments 1-21
Black-Scholes sensitivity to change 10-66
of annuity, periodic 10-38
rate of return 2-17
risk-free 5-24
sensitivity of bond prices to changes in 5-3
term structure 2-2, 2-31
internal rate of return 10-310
for nonperiodic cash flow 10-554
modified 10-345
inversion, matrix 1-13
investment horizon 2-32
`irr` 10-310
`isbusday` 10-311
`iscompatible` 10-313
`isequal` 10-314
`isfield` 10-315
`issorted` 10-316
issue date 2-21
Ito process 2-35

L

lagging and leading moving averages chart 10-349
`lagts` 10-317
lambda 2-34
last
business date of month 10-318
date of month 10-244
day of month 10-245
weekday in month 10-327
last coupon date 2-21
`lbusdate` 10-318
leading and lagging moving averages chart 10-349
`leadts` 10-321
left division 1-16
`length` 10-320
leverage of an option 10-62
linear algebra 1-8, 1-13
linear equations 5-7
solving simultaneous 1-13
system of 1-13
`llow` 10-322
loan
interest on 2-19
payment with odd first period 10-373
periodic payment of 10-374
`log` 10-324
`log10` 10-326
`log2` 10-325
`lweekdate` 10-327

M

`m2xdate` 10-329
Macaulay duration 5-3
for fixed-income securities 2-30
`macd` 10-331
MACD signal line 10-331
main GUI window 8-2
MATLAB
date number
from Excel date number 10-552
to Excel date number 10-329
matrices
adding and subtracting 1-7

as arguments, limitations 1-21
dividing 1-13
enlarging 1-5
multiplying 1-8, 1-11
multiplying vectors and 1-10
of string input 1-19
singular 1-13
square 1-13
transposing 1-6
matrix 1-4
adding or subtracting a scalar 1-8
algebra refresher 1-7
covariance 10-246
elements
generating 1-6
referencing 1-4
identity 1-13
indices of date numbers 10-169
indices of integers in 10-169
inversion 1-13
multiplying by a scalar 1-12
numbers and strings in a 1-20
maturity
price with interest at 10-428
yield of a security paying interest at 10-560
maturity date 2-22
`max` 10-333
`maxdrawdown` 10-334
maximum likelihood estimate (MLE) 10-233
`mean` 10-336
`medprice` 10-337
`merge` 10-339
`min` 10-342
`minus` 10-343
`minute` 10-344
minute of date or time 10-344
`mirr` 10-345
MLE (maximum likelihood estimate) 10-233
modified duration 5-3, 10-116
for fixed-income securities 2-30
modified internal rate of return 10-345
momentum 10-517
month
add, to starting date 10-170
date of specific weekday 10-367
day of 10-182
first business date of 10-250
last business date 10-318
last date of 10-244
last day of 10-245
`month` 10-347
months
last weekday in 10-327
number of months between dates 10-348
`months` 10-348
`movavg` 10-349
Moving Average Convergence/Divergence (MACD) 10-331
moving averages chart 10-349
`mrdivide` 10-351
`mtimes` 10-352
multiplying
a matrix by a scalar 1-12
matrices 1-8
two matrices 1-11
vectors 1-9
vectors and matrices 1-10
`mvnrfish` 10-353
`mvnrmle` 10-355
`mvnrobj` 10-359
`mvnrstd` 10-361

N

names
  variable 1-7
NaN 2-26
negative cash flows 2-17
negvolidx 10-363
Newton's method 2-29
next
  business day 2-10
  coupon date after settlement date 10-138
  or previous business day 10-100
nominal rate of return 10-365
nomrr 10-365
nontrading days 2-10, 10-306
notation 1-4
  row, column 1-4
now 10-366
number of
  days in year 10-557
  periods to obtain value 10-39
  whole months between dates 10-348
numbers
  and strings in a matrix 1-20
  date 2-4
nweekdate 10-367

O

object structure 6-3
observation 10-230
odd first period
  payment of loan or annuity with 10-373
On-Balance Volume (OBV) 9-9
onbalvol 10-369
operating element-by-element 1-16
operations, array 1-16
opprofit 10-371
optimal portfolio 3-2
option
  leverage of 10-62
  plotting sensitivities of 5-21
  plotting sensitivities of a portfolio of 5-23
  pricing
    Black's model 10-55
  profit 10-371
output conversions, date 2-7
overloaded functions
  most common 7-23
  types of 7-15

P

par value 2-22
par yield curve
  from zero curve 10-579
  to zero curve 10-436
past month, date of day in 10-170
payadv 10-372
payment
  of loan or annuity with odd first period 10-373
  periodic, given number of advance payments 10-372
  periodic, of loan or annuity 10-374
  uniform, equal to varying cash flow 10-375
payodd 10-373
payper 10-374
payuni 10-375
pcalims 10-376
pcgcomp 10-379
pcglims 10-381
pcpval 10-384
peravg 10-386
period 2-22
periodic interest rate of annuity 10-38

periodic payment
  future value with fixed 10-289
  given advance payments 10-372
  of loan or annuity 10-374
  present value with fixed 10-431
periodicreturns 10-387
pivot year 10-173
plot 10-388
plotting
  efficient frontier 5-19
  sensitivities of a portfolio of options 5-23
  sensitivities of an option 5-21
plus 10-390
point and figure chart 10-391
pointfig 10-391
portalloc 3-9, 3-10, 10-392
portcons 3-14, 10-395
portfolio
  convexity 5-4, 5-6
  duration 5-4, 5-6
  expected rate of return 10-413
  of options, plotting sensitivities of 5-23
  optimal 3-2
  optimization 3-3
  risks, returns, and weights
    randomized 10-402
  selection 3-8
portfolios
  analyzing 2-38
  of European stock options
    constructing greek-neutral 5-12
portopt 10-399
portrand 10-402
portsim 10-403
portstats 10-413
portvrisk 10-415
posvolidx 10-417
power 10-419
prbyzero 10-420
prcroc 10-424
prdisc 10-426
present value 2-18
  of varying cash flow 10-434
  with fixed periodic payments 10-431
previous quasi coupon date 10-149
price
  change, Black-Scholes sensitivity to underlying 10-57
  of discounted security 10-426
  of Treasury bill 10-430
  volatility, Black-Scholes sensitivity to underlying 10-70
  with interest at maturity 10-428
pricing
  and analyzing equity derivatives 2-34
  and computing yields for fixed-income securities 2-21
  fixed-income securities 2-29
principal 10-35
prmat 10-428
profit, option 10-371
prtbill 10-430
purchase price 2-22
put and call pricing
  binomial 10-51
  Black-Scholes 10-64
pvfix 10-431
pvtrend 10-432
pvvar 10-434
pyld2zero 10-436

Q

quasi coupon date

previous 10-149
quasi-coupon dates 2-21

R

randomized portfolio risks, returns, and weights 10-402
rate of a security, discount 10-206
rate of return 2-17
after-tax 10-471
effective 10-239
internal 10-310
internal for nonperiodic cash flow 10-554
modified internal 10-345
nominal 10-365
portfolio expected 10-413
`rdivide` 10-440
record 10-230
redemption value 2-22
reference date 2-28
referencing matrix elements 1-4, 1-6
`refield` 10-445
Relative Strength Index (RSI) 9-8
remaining depreciable value 2-19, 10-199
`resamplets` 10-441
`ret2tick` 10-442
return arguments, function 1-20
rho 2-34
risk aversion 3-8
risk-free interest rates 5-24
risks
returns, and weights
randomized portfolio 10-402
row, column notation 1-4
row-by-column 1-4
`rsindex` 10-446

S

scalar 1-4
adding or subtracting 1-8
multiplying a matrix by 1-12
`second` 10-448
seconds of date or time 10-448
securities industry association 2-21
`selectreturn` 10-449
sensitivity
fixed-income 2-30
measures for derivatives 2-34
of a portfolio of options, plotting 5-23
of an option, plotting 5-21
of bond prices to changes in interest rates 5-3
of cash flow 2-19
to
interest rate change, Black-Scholes 10-66
to time-until-maturity change, Black-Scholes 10-68
to underlying delta change, Black-Scholes 10-59
to underlying price change, Black-Scholes 10-57
to underlying price volatility, Black-Scholes 10-70
visualizing to parallel shifts in the yield curve 5-8
serial dates 7-7
`setfield` 10-450
settlement date 2-21
coupon period containing 10-158
days between previous coupon date and 10-155
days between, and coupon date 10-152
next coupon date after 10-138
SIA 2-21
compatibility 2-21

default parameter values 2-25
framework 2-24
order of precedence 2-28
use of nonlinear formulas 2-29
SIA conventions 2-21
signal line 10-331
single quotes 1-19
singular matrices 1-13
size 10-452
smoothts 10-453
solving
sample problems with the toolbox 5-2
sortfts 10-455
spctkd 10-456
spreadsheets 1-4
square matrices 1-13
std 10-459
stochosc 10-460
straight-line depreciation 2-19, 10-201
strings
and numbers in a matrix 1-20
date 2-4, 10-176
input, matrices of 1-19
stored as character array 1-19
structures 7-3
subsasgn 10-463
subsref 10-466
subtracting
a scalar and a matrix 1-8
matrices 1-7
sum of years' digits depreciation 2-19, 10-200
swap 5-16
synch date 2-28
synchronization date 2-28
system of linear equations 1-13

T

targetreturn 10-470
taxedrr 10-471
tbl2bond 10-472
technical analysis 9-2
term structure 2-2, 2-31, 5-3, 10-203, 10-292, 10-436, 10-472, 10-563, 10-568, 10-573, 10-576, 10-579
parameters from Treasury bond parameters 10-512
terminology, fixed-income securities 2-21
text file transformation 6-13
theta 2-35
thirdwednesday 10-474
thirtytwo2dec 10-476
three-dimensional graphics 5-12
tick labels 10-166
tick2ret 10-477
time
current 2-8, 10-366
hour of 10-309
minute of 10-344
seconds of 10-448
time factor 10-121
time2date 10-480
times 10-479
time-until-maturity change
Black-Scholes sensitivity to 10-68
toannual 10-483
todaily 10-487
today 10-489
todecimal 10-490
tomonthly 10-491
toquarterly 10-496
toquoted 10-501
tosemi 10-502
totalreturnprice 10-507

toweekly 10-508
tr2bonds 10-512
tracking error 3-20
tracking error efficient frontier 3-20
transposing matrices 1-6
Treasury bill 2-31
  bond equivalent yield for 10-50
  parameters to Treasury bond parameters 10-472
  price of 10-430
  yield of 10-562
Treasury bond 2-31
  parameters
    from Treasury bill parameters 10-472
    to term-structure parameters 10-512
tsaccel 10-515
tsmom 10-517
tsmovavg 10-519
typprice 10-521

U

ugarch 10-523
ugarchllf 10-525
ugarchpred 10-527
ugarchsim 10-530
uminus 10-535
uniform payment equal to varying cash flow 10-375
uplus 10-536

V

variable names 1-7
vector 1-4
  date 10-179
  of dates 1-20
vectors
  as arguments, limitations 1-21
  computing dot products of 1-10
  multiplying 1-9
  multiplying matrices and 1-10
vega 2-35
vertcat 10-537
visualizing the sensitivity of a bond portfolio's price to parallel shifts in the yield curve 5-8
volatility
  Black-Scholes implied 10-60
  implied 2-35
volroc 10-539

W

wclose 10-541
week, day of 10-543
weekday
  date of specific, in month 10-367
weekday 10-543
weights2holdings 10-545
willad 10-546
Williams %R 9-6
willpctr 10-548
  example 9-6
workday, date of future or past 10-181
working days between dates 10-551
wrkdydif 10-551

X

x2mdate 10-552
xirr 10-554

Y

year
fraction of between dates 10-558
number of days in 10-557
of date 10-556
`year` 10-556
`yeardays` 10-557
`yearfrac` 10-558
yield
curve 5-3, 5-6
visualizing sensitivity of bond portfolio's price to parallel shifts in 5-8
for Treasury bill, bond equivalent 10-50
functions for fixed-income securities 2-29
of discounted security 10-559
of security paying interest at maturity 10-560
of Treasury bill 10-562
yields
for fixed-income securities, pricing and computing 2-21
yield-to-maturity 2-22
`ylddisc` 10-559
`yldmat` 10-560
`yldtbill` 10-562

Z

`zbtprice` 10-563
`zbtyield` 10-568
zero curve 10-512, 10-564, 10-569
from coupon bond prices 10-563
from coupon bond yields 10-568
from discount curve 10-203
from forward curve 10-292
from par yield curve 10-436
to discount curve 10-573
to forward curve 10-576
to par yield curve 10-579
`zero2disc` 10-573
`zero2fwd` 10-576
`zero2pyld` 10-579
zero-coupon bond 10-204, 10-564, 10-569
Zoom tool 6-20