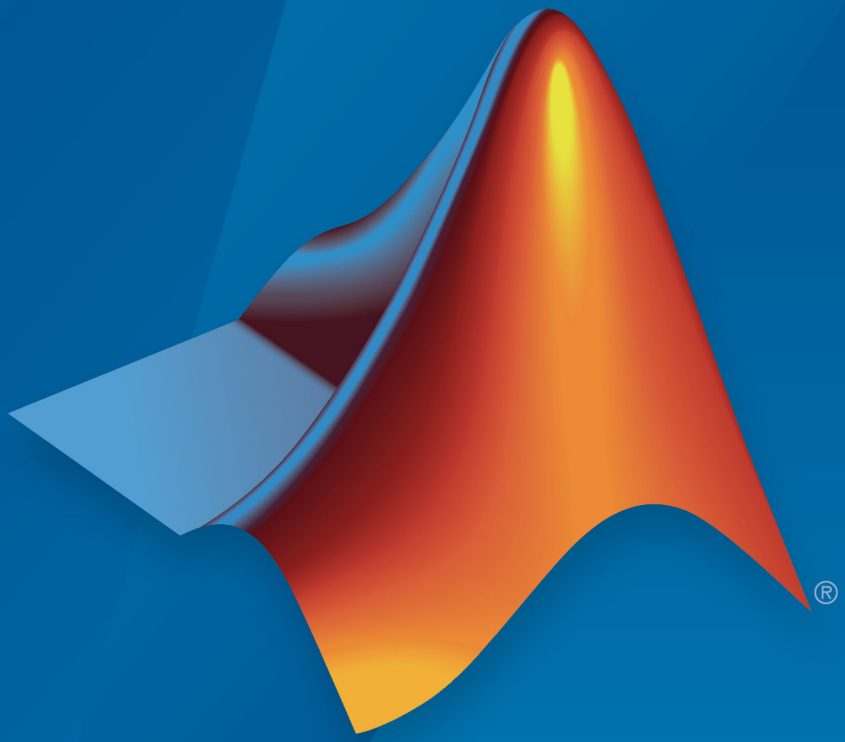
Financial Toolbox™

User's Guide



MATLAB®

R2017b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Financial Toolbox™ User's Guide

Revision History

October 1995	First printing	
January 1998	Second printing	Revised for Version 1.1
January 1999	Third printing	Revised for Version 2.0 (Release 11)
November 2000	Fourth printing	Revised for Version 2.1.2 (Release 12)
May 2003	Online only	Revised for Version 2.3 (Release 13)
June 2004	Online only	Revised for Version 2.4 (Release 14)
August 2004	Online only	Revised for Version 2.4.1 (Release 14+)
September 2005	Fifth printing	Revised for Version 2.5 (Release 14SP3)
March 2006	Online only	Revised for Version 3.0 (Release 2006a)
September 2006	Sixth printing	Revised for Version 3.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.4 (Release 2008a)
October 2008	Online only	Revised for Version 3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.6 (Release 2009a)
September 2009	Online only	Revised for Version 3.7 (Release 2009b)
March 2010	Online only	Revised for Version 3.7.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.8 (Release 2010b)
April 2011	Online only	Revised for Version 4.0 (Release 2011a)
September 2011	Online only	Revised for Version 4.1 (Release 2011b)
March 2012	Online only	Revised for Version 4.2 (Release 2012a)
September 2012	Online only	Revised for Version 5.0 (Release 2012b)
March 2013	Online only	Revised for Version 5.1 (Release 2013a)
September 2013	Online only	Revised for Version 5.2 (Release 2013b)
March 2014	Online only	Revised for Version 5.3 (Release 2014a)
October 2014	Online only	Revised for Version 5.4 (Release 2014b)
March 2015	Online only	Revised for Version 5.5 (Release 2015a)
September 2015	Online only	Revised for Version 5.6 (Release 2015b)
March 2016	Online only	Revised for Version 5.7 (Release 2016a)
September 2016	Online only	Revised for Version 5.8 (Release 2016b)
March 2017	Online only	Revised for Version 5.9 (Release 2017a)
September 2017	Online only	Revised for Version 5.10 (Release 2017b)

Contents

1 Getting Started

2 Performing Common Financial Tasks

3 Portfolio Analysis

4 Mean-Variance Portfolio Optimization Tools

5 CVaR Portfolio Optimization Tools

6 MAD Portfolio Optimization Tools

7 Investment Performance Metrics

8 Credit Risk Analysis

9 Regression with Missing Data

10 Solving Sample Problems

11 Financial Time Series Analysis

12 Using Financial Time Series

13 Financial Time Series App

14 Financial Time Series User Interface

15 Trading Date Utilities

16 Technical Analysis

17 Stochastic Differential Equations

18 Functions — Alphabetical List

A Bibliography

Glossary

Getting Started

Financial Toolbox Product Description

Analyze financial data and develop financial models

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

Key Features

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

Expected Users

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

In designing Financial Toolbox documentation, we assume that your title is like one of these:

- Analyst, quantitative analyst
- Risk manager
- Portfolio manager
- Asset allocator
- Financial engineer
- Trader
- Student, professor, or other academic

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

- Finance, economics, perhaps accounting
- Engineering, mathematics, physics, other quantitative sciences
- Focus on quantitative approaches to financial problems

Analyze Sets of Numbers Using Matrix Functions

In this section...

Introduction

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

Key Definitions

Matrix

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

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

Vector

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

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

Scalar

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

Referencing Matrix Elements

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

```
Bonds(1,2)

ans =

          0.06

Cash(3)

ans =

       5280.00
```

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

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

adds another row to the matrix and creates

```
Bonds =

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

Likewise,

```
Prices = [987.50
          475.00
          995.00]

Bonds = [Prices, Bonds]
```

adds another column and creates

```
Bonds =
```

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

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

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

BondItems =

    500.00   0.055   4
```

Transposing Matrices

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

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

produces

```
Cash =

        1500
        4470
        5280
       -1299
```

See Also

More About

- "Matrix Algebra Refresher" on page 1-7
- "Using Input and Output Arguments with Functions" on page 1-17

Matrix Algebra Refresher

In this section...

Introduction

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

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

`http://www.stanford.edu/~wfsharpe/mia/mia.htm`

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

Adding and Subtracting Matrices

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

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

Portfolios_RS = [175   125
                 200   200
                 100   500];

NewPortfolios = Portfolios_PQ + Portfolios_RS

NewPortfolios =

        275           325
        700           600
        400           650
```

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

```
SmallerPortf = NewPortfolios-10

SmallerPortf =
        265.00        315.00
        690.00        590.00
        390.00        640.00
```

Multiplying Matrices

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

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

Multiplying Vectors

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

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

The portfolio contains these numbers of shares of each stock.

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

To find the value of the portfolio, multiply the vectors

```
PortfValue = ClosePrices * NumShares
```

which yields:

```
PortfValue =

           3.5413e+004
```

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

To illustrate order dependence, switch the order of the vectors

```
Values = NumShares * ClosePrices
```

```
Values =

  1.0e+004 *

    0.4250    0.1500    0.7887
    2.1250    0.7500    3.9438
    1.2750    0.4500    2.3663
```

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

Computing Dot Products of Vectors

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

$$Y = [y_1, y_2, \ldots, y_n]$$
$$X = [x_1, x_2, \ldots, x_n]$$

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)`. Be sure that the two vectors have the same dimensions. To illustrate, use the previous vectors.

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

Value =

      3.5413e+004

Value = sum(ClosePrices .* NumShares')

Value =

      3.5413e+004
```

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

Multiplying Vectors and Matrices

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

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

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

```
WeekPortValue = WeekClosePr * PortQuan
```

```
WeekPortValue =

1.0e+004 *

    3.5412
    3.5587
    3.5475
    3.5550
    3.5513
```

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

Multiplying Two Matrices

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

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

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

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{12} + \ldots + a_{in}b_{nj}.$$

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

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

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

```
PortfolioValues = WeekClosePr * Portfolios

PortfolioValues =
```

```
1.0e+004 *

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

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

Multiplying a Matrix by a Scalar

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

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

DoublePort = Portfolios * 2

DoublePort =
        200           400
       1000           800
        600           300
```

Dividing Matrices

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

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

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

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

In general, matrix `A` must be a nonsingular square matrix; that is, it must be invertible and it must have the same number of rows and columns. (Generally, a matrix is invertible if the matrix times its inverse equals the identity matrix. To understand the theory and proofs, consult a textbook on linear algebra such as *Elementary Linear Algebra* by Hill listed in "Bibliography" on page A-2.) MATLAB gives a warning message if the matrix is singular or nearly so.

Solving Simultaneous Linear Equations

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

A linear equation is any equation of the form

$$a_1x + a_2y = b,$$

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

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

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

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

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

where $A*X = B$.

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

```
A = [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

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

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

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

	M1	M2

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

As MATLAB matrices:

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

Units = [10
        20];

Receipts = Cash * Units

Receipts =

       2400
       1600
```

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

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

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

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

Receipts = [7000
            5000];

Units = Cash \ Receipts
```

```
Units =

         43.7500
         37.5000
```

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

Operating Element by Element

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

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

Yields = Dividends ./ Prices

Yields =

    0.0741    0.0225    0.0597    0.0744
```

See Also

More About

- “Analyze Sets of Numbers Using Matrix Functions” on page 1-4
- “Using Input and Output Arguments with Functions” on page 1-17

Using Input and Output Arguments with Functions

In this section...

Input Arguments

Matrix Input

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

Because MATLAB can process vectors and matrices easily, most Financial Toolbox functions allow vector or matrix input arguments, rather than 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 that 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 Character Vector Input

Enter MATLAB character vectors surrounded by single quotes (`'string'`).

Character vector are stored as character arrays, one ASCII character per element. Thus, the date character vector

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

is actually a 1-by-9 vector. Character vectors making up the rows of a matrix or vector all must have the same length. To enter several date character vectors, therefore, use a column vector and be sure that all character vectors 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.

Do not mix numbers and character vectors in a matrix. If you do, MATLAB treats all entries as characters. For example,

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

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

```
Item =

SZc14-Sep-1999
```

Output Arguments

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

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

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

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

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

Some functions that return vectors accept only scalars as arguments. Why could such functions not accept vectors as arguments and return matrices, where each column in the

output matrix corresponds to an entry in the input vector? The answer is that the output vectors can be variable length and thus will not fit in a matrix without some convention to indicate that the shorter columns are missing data.

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

`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.

See Also

More About

- “Analyze Sets of Numbers Using Matrix Functions” on page 1-4
- “Matrix Algebra Refresher” on page 1-7

Performing Common Financial Tasks

Handle and Convert Dates

In this section...

Date Formats

Virtually all financial data derives from a time series, functions in Financial Toolbox have extensive date-handling capabilities. The toolbox functions support date or date-and-time formats as character vectors, datetime arrays, or serial date numbers.

- Date character vectors are text that represent date and time, which you can use with multiple formats. For example, `'dd-mmm-yyyy HH:MM:SS'`, `'dd-mmm-yyyy'`, and `'mm/dd/yyyy'` are all supported text formats for a date character vector. Most often, you work with date character vectors (such as `14-Sep-1999`) when dealing with dates.
- Datetime arrays, created using `datetime`, are the best data type for representing points in time. `datetime` values have flexible display formats and up to nanosecond precision, and can account for time zones, daylight saving time, and leap seconds. When `datetime` objects are used as inputs to other Financial Toolbox functions, the format of the input `datetime` object is perserved. For example:

```
originalDate = datetime('now','Format','yyyy-MM-dd HH:mm:ss');
  % Find the next business day
  b = busdate(originalDate)
  b =
  datetime
   2017-04-20 14:47:39
```

- Serial date numbers represent a calendar date as the number of days that have passed since a fixed base date. In MATLAB software, serial date number `1` is January 1,0000 A.D. Financial Toolbox works internally with serial date numbers (such as, `730377`). MATLAB also uses serial time to represent fractions of days beginning at

midnight. For example, 6 p.m. equals `0.75` serial days, so 6:00 p.m. on `14-Sep-1999`, in MATLAB, is serial date number `730377.75`

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

Many Financial Toolbox functions that require dates as input arguments accept date character vectors, datetime arrays, or serial date numbers. If you are dealing with a few dates at the MATLAB command-line level, date character vectors are more convenient. If you are using Financial Toolbox functions on large numbers of dates, as in analyzing large portfolios or cash flows, performance improves if you use datetime arrays or serial date numbers. For more information, see "Represent Dates and Times in MATLAB" (MATLAB).

Date Conversions

Financial Toolbox provides functions that convert date character vectors to or from serial date numbers. In addition, you can convert character vectors or serial date numbers to datetime arrays.

Functions that convert between date formats are:

`datedisp`	Displays a numeric matrix with date entries formatted as date character vectors.
`datenum`	Converts a date character vector to a serial date number.
`datestr`	Converts a serial date number to a date character vector.
`datetime`	Converts from date character vectors or serial date numbers to create a datetime array.
`datevec`	Converts a serial date number or date character vector to a date vector whose elements are `[Year Month Day Hour Minute Second]`.
`m2xdate`	Converts MATLAB serial date number to Excel® serial date number.

x2mdate	Converts Microsoft® Excel serial date number to MATLAB serial date number.

For more information, see “Convert Between Datetime Arrays, Numbers, and Text” (MATLAB).

Convert Between Datetime Arrays and Character Vectors

A date can be a character vector composed of fields related to a specific date and time. There are several ways to represent dates and times in several text formats. For example, all the following are character vectors represent August 23, 2010 at 04:35:42 PM:

```
'23-Aug-2010 04:35:06 PM'
'Wednesday, August 23'
'08/23/10 16:35'
'Aug 23 16:35:42.946'
```

A date character vector includes characters that separate the fields, such as the hyphen, space, and colon used here:

```
d = '23-Aug-2010 16:35:42'
```

Convert one or more date character vectors to a `datetime` array using the `datetime` function. For the best performance, specify the format of the input character vectors as an input to `datetime`.

Note The specifiers that `datetime` uses to describe date and time formats differ from the specifiers that the `datestr`, `datevec`, and `datenum` functions accept.

```
t = datetime(d,'InputFormat','dd-MMM-yyyy HH:mm:ss')
```

```
t =

   23-Aug-2010 16:35:42
```

Although the date string, `d`, and the `datetime` scalar, `t`, look similar, they are not equal. View the size and data type of each variable.

```
whos d t
```

```
  Name      Size             Bytes  Class       Attributes

  d         1x20                40  char
  t         1x1                121  datetime
```

Convert a `datetime` array to a character vector that uses `char` or `cellstr`. For example, convert the current date and time to a timestamp to append to a file name.

```
t = datetime('now','Format','yyyy-MM-dd''T''HHmmss')

t =

  datetime

   2016-12-11T125628

S = char(t);
filename = ['myTest_',S]

filename =

    'myTest_2016-12-11T125628'
```

Convert Serial Date Numbers to Datetime Arrays

Serial time can represent fractions of days beginning at midnight. For example, 6 p.m. equals `0.75` serial days, so the character vector `'31-Oct-2003, 6:00 PM'` in MATLAB is date number `731885.75`.

Convert one or more serial date numbers to a `datetime` array using the `datetime` function. Specify the type of date number that is being converted:

```
t = datetime(731885.75,'ConvertFrom','datenum')

t =

  datetime

   31-Oct-2003 18:00:00
```

Convert Datetime Arrays to Numeric Values

Some MATLAB functions accept numeric data types but not datetime values as inputs. To apply these functions to your date and time data, first, convert datetime values to meaningful numeric values, and then call the function. For example, the `log` function

accepts `double` inputs but not `datetime` inputs. Suppose that you have a `datetime` array of dates spanning the course of a research study or experiment.

```
t = datetime(2014,6,18) + calmonths(1:4)
```

```
t =

  1×4 datetime array

   18-Jul-2014   18-Aug-2014   18-Sep-2014   18-Oct-2014
```

Subtract the origin value. For example, the origin value can be the starting day of an experiment.

```
dt = t - datetime(2014,7,1)
```

```
dt =

  1×4 duration array

    408:00:00   1152:00:00   1896:00:00   2616:00:00
```

`dt` is a `duration` array. Convert `dt` to a `double` array of values in units of years, days, hours, minutes, or seconds by using the `years`, `days`, `hours`, `minutes`, or `seconds` function, respectively.

```
x = hours(dt)
```

```
x =

         408        1152        1896        2616
```

Pass the `double` array as the input to the `log` function.

```
y = log(x)
```

```
y =

    6.0113    7.0493    7.5475    7.8694
```

Input Conversions with datenum

The `datenum` function is important for using Financial Toolbox software efficiently. `datenum` takes an input date character vector in any of several formats, with `'dd-mmm-yyyy'`, `'mm/dd/yyyy'`, or `'dd-mmm-yyyy, hh:mm:ss.ss'` formats being the most

common. The input date character vector can have up to six fields formed by letters and numbers separated by any other characters, such that:

- The day field is an integer from `1` through `31`.
- The month field is either an integer from `1` through `12` or an alphabetical character vector 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 on the current year. If the year is omitted, the current year is 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 all these dates are equivalent:

```
'17-May-1999'
'17-May-99'
'17-may'
'May 17, 1999'
'5/17/99'
'5/17'
```

Also, both of these formats represent the same time.

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

The default format for numbers-only input follows the US convention. Therefore, 3/6 is March 6, not June 3.

With `datenum`, you can convert dates into serial date format, store them in a matrix variable, and 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)

ans =

  100.3503
```

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

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

ans =

  100.3503
```

`bndprice` is an example of a function designed to detect the presence of date character vectors and make the conversion automatically. For functions like `bndprice`, date character vectors can be passed directly.

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

ans =

  100.3503
```

The decision to represent dates as either date character vectors or serial date numbers is often a matter of convenience. For example, when formatting data for visual display or for debugging date-handling code, you can view dates more easily as date character vectors because serial date numbers are difficult to interpret. Alternately, serial date numbers are just another type of numeric data, which you can place in a matrix along with any other numeric data for convenient manipulation.

Remember that if you create a vector of input date character vectors, use a column vector, and be sure that all character vectors are the same length. To ensure that the character vectors are the same length, fill the character vectors with spaces or zeros. For more information, see “Matrices of Character Vector Input” on page 1-18.

Output Conversions with datestr

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

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

Current Date and Time

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

```
today

ans =

      736675

now
```

```
ans =

   7.3668e+05
```

The MATLAB function `date` returns a character vector for the current date.

```
date

ans =

    '11-Dec-2016'
```

Determining Specific Dates

Financial Toolbox provides many functions for determining specific dates. For example, assume that you schedule an accounting procedure for the last Friday of every month. Use the `lweekdate` function to return those dates for the year 2000. The input argument `6` specifies Friday.

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

Fridays =

  12×11 char array

    '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'
```

Another example of needing specific dates could be that your company closes on Martin Luther King Jr. Day, which is the third Monday in January. You can use the`nweekdate` function to determine those specific dates for 2011 through 2014.

```
MLKDates = nweekdate(3,2,2011:2014,1);
MLKDays = datestr(MLKDates)
```

```
MLKDays =

  4×11 char array

    '17-Jan-2011'
    '16-Jan-2012'
    '21-Jan-2013'
    '20-Jan-2014'
```

Determining Holidays

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

In this example, use `holidays` to determine the standard holidays in the last half of 2012.

```
LHHDates = holidays('1-Jul-2012','31-Dec-2012');
LHHDays = datestr(LHHDates)
```

```
LHHDays =

  6×11 char array

    '04-Jul-2012'
    '03-Sep-2012'
    '29-Oct-2012'
    '30-Oct-2012'
    '22-Nov-2012'
    '25-Dec-2012'
```

You can then use the `busdate` function to determine the next business day in 2012 after these holidays.

```
LHNextDates = busdate(LHHDates);
LHNextDays = datestr(LHNextDates)
```

```
LHNextDays =

  6×11 char array

    '05-Jul-2012'
    '04-Sep-2012'
    '31-Oct-2012'
    '31-Oct-2012'
    '23-Nov-2012'
    '26-Dec-2012'
```

Determining Cash-Flow Dates

To determine cash-flow dates for securities with periodic payments, use `cfdates`. This function accounts for the coupons per year, the day-count basis, and the end-of-month rule. For example, you can determine the cash-flow dates for a security that pays four coupons per year on the last day of the month using an `actual/365` day-count basis. To do so, enter the settlement date, the maturity date, and the parameters for `Period`, `Basis`, and `EndMonthRule`.

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

PayDays =

  7×11 char array

    '31-May-2000'
    '31-Aug-2000'
    '30-Nov-2000'
    '28-Feb-2001'
    '31-May-2001'
    '31-Aug-2001'
    '30-Nov-2001'
```

See Also

`busdate` | `cfdates` | `date` | `datedisp` | `datenum` | `datestr` | `datetime` | `datevec` | `format` | `holidays` | `lweekdate` | `m2xdate` | `nweekdate` | `nyseclosures` | `x2mdate`

Related Examples

- “Convert Between Datetime Arrays, Numbers, and Text” (MATLAB)
- “Read a Sequence of Spreadsheet Files” (MATLAB)
- “Charting Financial Data” on page 2-14
- “Trading Calendars User Interface” on page 15-2
- “UICalendar User Interface” on page 15-4

More About

- “Convert Dates Between Microsoft Excel and MATLAB” (Spreadsheet Link)

Charting Financial Data

Introduction

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, see “Chart Technical Indicators”.

Here are two plotting examples: a high-low-close chart on page 2-16 of sample IBM® stock price data, and a Bollinger band on page 2-18 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. The data in `ibm.dat` is fictional and for illustrative use only.

See Also

`bolling` | `bollinger` | `busdate` | `candle` | `candle` | `cfdates` | `cur2frac` | `cur2str` | `date` | `dateaxis` | `datedisp` | `datenum` | `datestr` | `datevec` | `frac2cur` | `highlow` | `highlow` | `holidays` | `load` | `lweekdate` | `m2xdate` | `movavg` | `nweekdate` | `nyseclosures` | `pointfig` | `size` | `x2mdate`

Related Examples

- “High-Low-Close Chart” on page 2-16
- “Bollinger Chart” on page 2-18
- “Handle and Convert Dates” on page 2-2

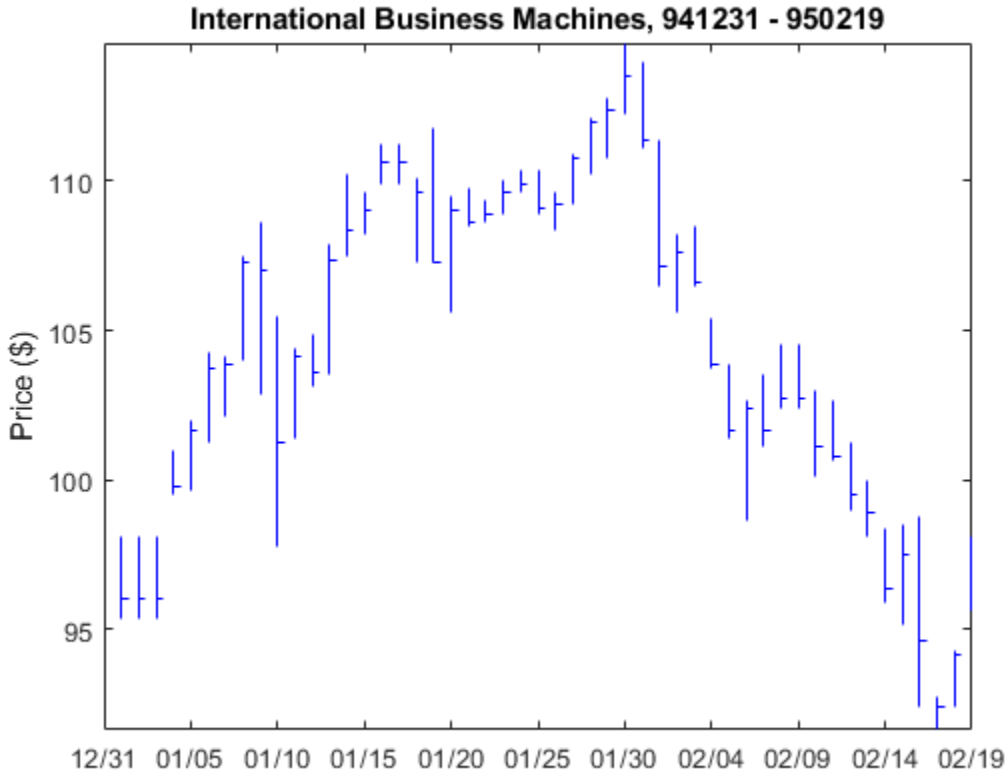
High-Low-Close Chart

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. 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.

```
figure;
highlow(ibm(ro-50:ro,2),ibm(ro-50:ro,3),ibm(ro-50:ro,4),[],'b');
xlabel('');
ylabel('Price ($)');
title('International Business Machines, 941231 - 950219');
axis([0 50 -inf inf]);
dateaxis('x',6,'31-Dec-1994')
```



See Also

`bolling` | `bollinger` | `busdate` | `candle` | `candle` | `cfdates` | `cur2frac` | `cur2str` | `date` | `dateaxis` | `datedisp` | `datenum` | `datestr` | `datevec` | `frac2cur` | `highlow` | `highlow` | `holidays` | `load` | `lweekdate` | `m2xdate` | `movavg` | `nweekdate` | `nyseclosures` | `pointfig` | `size` | `x2mdate`

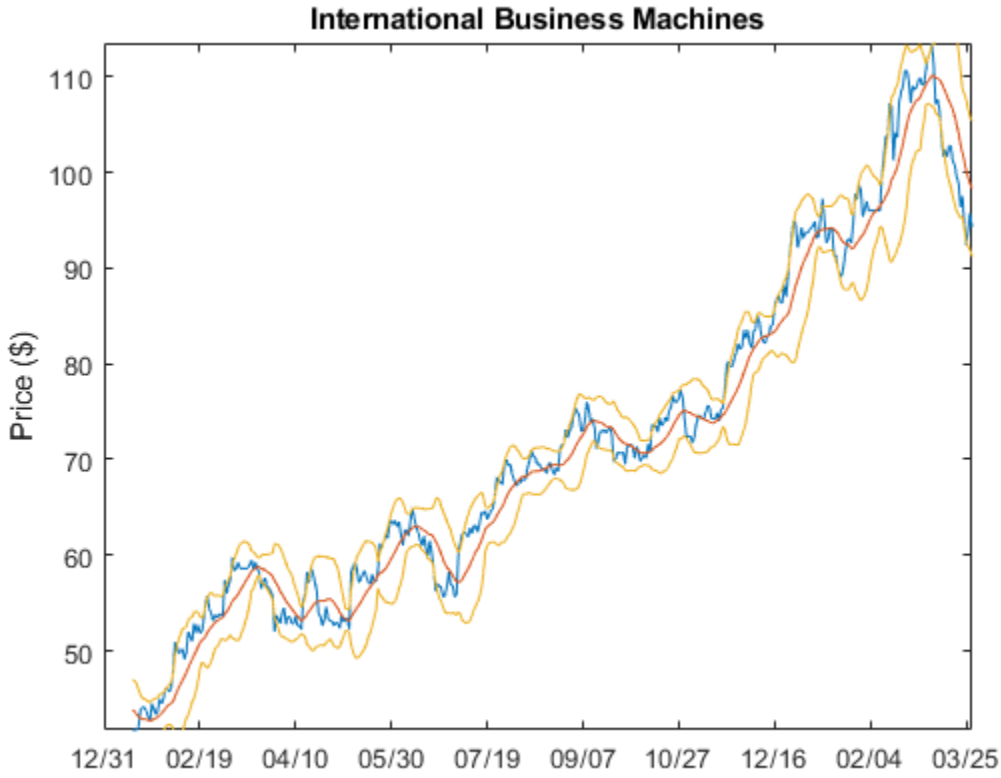
Related Examples

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

Bollinger Chart

The `bolling` function in Financial Toolbox™ software produces a Bollinger band chart using all the closing prices in an 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. First, load the data using the ibm.dat data file and then execute the `bolling` function to plot the Bollinger bands.

```
load ibm.dat;
[ro, co] = size(ibm);
bolling(ibm(:,4), 15, 0);

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



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

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

See Also

`bolling` | `bollinger` | `busdate` | `candle` | `candle` | `cfdates` | `cur2frac` | `cur2str` | `date` | `dateaxis` | `datedisp` | `datenum` | `datestr` | `datevec` | `frac2cur` | `highlow` | `highlow` | `holidays` | `load` | `lweekdate` | `m2xdate` | `movavg` | `nweekdate` | `nyseclosures` | `pointfig` | `size` | `x2mdate`

Related Examples

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

Analyzing and Computing Cash Flows

In this section...

Introduction

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

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

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

Interest Rates/Rates of Return

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

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

which gives a rate of return of 11.72%.

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 *n*th period, and *n* is the number of periods. `irr` finds the rate *r* such that the present value of the cash flow equals the initial investment. If all 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.

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

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

The answer is 9.38%.

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

Present or Future Values

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

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

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

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

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

Depreciation

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

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

```
Decline1 = 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 about 11.28% annually.

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

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

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

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

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

This function returns vectors containing the amount of principal paid,

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

the amount of interest paid,

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

the remaining balance for each period of the loan,

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

and a scalar for the monthly payment.

```
Payment = 437.26
```

See Also

effrr | fvfix | fvvar | irr | nomrr | pvfix | pvvar

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Charting Financial Data” on page 2-14
- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Pricing and Computing Yields for Fixed-Income Securities

In this section...

Introduction

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

Fixed-Income Terminology

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

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

The *first coupon date* and *last coupon date* are the dates when the first and last coupons are paid, respectively. Although bonds typically pay periodic annual or semiannual

coupons, the length of the first and last coupon periods may differ from the standard coupon period. The toolbox includes price and yield functions that handle these odd first and/or last periods.

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

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

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

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

Period of a Bond

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

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

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

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

Basis of a Bond

Basis Value	Meaning	Description
0 (default)	actual/actual	Actual days held over actual days in coupon period. Denominator is 365 in most years and 366 in a leap year.
1	30/360 (SIA)	Each month contains 30 days; a year contains 360 days. Payments are adjusted for bonds that pay coupons on the last day of February.
2	actual/360	Actual days held over 360.
3	actual/365	Actual days held over 365, even in leap years.
4	30/360 PSA	Number of days in every month is set to 30 (including February). If the start date of the period is either the 31st of a month or the last day of February, the start date is set to the 30th, while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
5	30/360 ISDA (International Swap Dealers Association)	Variant of 30/360 with slight differences for calculating number of days in a month.
6	30E/360 European	Variant of 30/360 used primarily in Europe.
7	actual/365 Japanese	All years contain 365 days. Leap days are ignored.

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

Basis Value	Meaning	Description
`13`	BUS/252	The number of business days between the previous coupon payment and the settlement data divided by 252. BUS/252 business days are non-weekend, non-holiday days. The `holidays.m` file defines holidays.

For more information, see **basis** on page Glossary-0 .

Note Although the concept of day count sounds deceptively simple, the actual calculation of day counts can be complex. You can find a good discussion of day counts and the formulas for calculating them in Chapter 5 of Stigum and Robinson, *Money Market and Bond Calculations* in "Bibliography" on page A-2.

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

End-of-Month Rule

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

Framework

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

Common Input Arguments

Input	Meaning
`Settle`	Settlement date
`Maturity`	Maturity date
`Period`	Coupon payment period
`Basis`	Day-count basis
`EndMonthRule`	End-of-month payment rule
`IssueDate`	Bond issue date
`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 are set to the default values if you do not explicitly set them. By default, the `FirstCouponDate` and `LastCouponDate` are nonapplicable. In other words, if you do not specify `FirstCouponDate` and `LastCouponDate`, the bond is assumed to have no odd first or last coupon periods. In this case, the bond is a standard bond with a coupon payment structure based solely on the maturity date.

Default Parameter Values

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

The complete calling syntax with the full input argument list is

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

while the minimal calling syntax requires only settlement and maturity dates

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

Single Bond Example

As an example, suppose that 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
```

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

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

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

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

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

Bond Portfolio Example

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

Now suppose that 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 that you have a two-bond portfolio.

The first call specifies a single (that is, scalar) 2 for `Period`. Passing a scalar tells `cfdates` to apply the scalar-valued input to all bonds in the portfolio. This is an example of implicit scalar-expansion. 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 is used. The fourth call is similar, except that a `NaN` has been passed. The fifth call passes two `NaN`'s, and has the same effect as the third. The last call passes the minimal input set.

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

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

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

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

Coupon Date Calculations

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

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

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

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

Yield Conventions

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

Pricing Functions

This example shows how easily you can compute the price of a bond with an odd first period using the function `bndprice`. Assume that 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 that there is no odd last coupon date and that the face value of the bond is $100. Calling the function

```
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, ...
Maturity, [], [], [], IssueDate, FirstCouponDate)
```

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

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

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

Yield Functions

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

```
Settle          = '12-Jan-2000';
Maturity        = '01-Oct-2001';
IssueDate       = '01-Jan-2000';
```

```
FirstCouponDate = '15-Jan-2000';
LastCouponDate  = '15-Apr-2000';
```

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

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

Calling the function

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

returns

`Yield = 0.0659` (6.60%).

Fixed-Income Sensitivities

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

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

Calculating Duration and Convexity for Bonds

The toolbox includes functions to perform sensitivity analysis such as convexity and the Macaulay and modified durations for fixed-income securities. The Macaulay duration of an income stream, such as a coupon bond, measures how long, on average, the owner waits before receiving a payment. It is the weighted average of the times payments are made, with the weights at time `T` equal to the present value of the money received at time `T`. The modified duration is the Macaulay duration discounted by the per-period interest rate; that is, divided by (1+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 (that is, semiannual coupons, and day-count `basis` = 0 (actual/actual), coupon payment structure synchronized to the maturity date, and end-of-month payment rule in effect).

```
CouponRate = 0.05;
Yield = 0.045;

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

The durations are

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

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

Calculating Key Rate Durations for Bonds

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

The calculation of `bndkrdur` supports:

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

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

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

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

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

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

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

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

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

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

```
krd =

    0.2865    0.8729    2.6451    8.5778
```

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

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

```
ans =

   12.3823   12.3919
```

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bnddury` | `bndkrdur`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Charting Financial Data” on page 2-14
- “Term Structure of Interest Rates” on page 2-45
- “Computing Treasury Bill Price and Yield” on page 2-41

More About

- “Treasury Bills Defined” on page 2-40

Treasury Bills Defined

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

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

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

See Also

`tbilldisc2yield` | `tbillprice` | `tbillrepo` | `tbillval01` | `tbillyield` | `tbillyield2disc` | `tbl2bond` | `tr2bonds` | `zbtprice` | `zbtyield`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Charting Financial Data” on page 2-14
- “Term Structure of Interest Rates” on page 2-45
- “Computing Treasury Bill Price and Yield” on page 2-41

Computing Treasury Bill Price and Yield

In this section...

Introduction

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

Treasury Bill Functions

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

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

Treasury Bill Repurchase Agreements

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

```
Maturity = '26-Dec-2002';
InitialDiscount = 0.0161;
```

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

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

BreakevenDiscount =

    0.0167
```

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

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

PriceOnPurchaseDate =

   99.5930
```

Next compute the interest due on the repurchase agreement.

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

RepoInterest =

    0.1237
```

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

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

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

PriceOnSaleDate =

   99.7167
```

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

Cash Flows from Repurchase Agreement

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

Treasury Bill Yields

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

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

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

```
BreakevenDiscount =

    0.0167


BEYield =

    0.0164
    0.0170


MMYield =

    0.0162
    0.0168
```

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

See Also

`tbilldisc2yield` | `tbillprice` | `tbillrepo` | `tbillval01` | `tbillyield` | `tbillyield2disc` | `tbl2bond` | `tr2bonds` | `zbtprice` | `zbtyield`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Charting Financial Data” on page 2-14
- “Term Structure of Interest Rates” on page 2-45

More About

- “Treasury Bills Defined” on page 2-40

Term Structure of Interest Rates

In this section...

Introduction

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

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

First, capture Treasury bill quotes in their reported format

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

then capture Treasury bond quotes in their reported format

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

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

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

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

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

Deriving an Implied Zero Curve

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

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

```
ZeroRates =

     0.05
     0.06

CurveDates =

      730846
      731160
```

`CurveDates` gives the investment horizon.

```
datestr(CurveDates)

ans =

26-Dec-2000
05-Nov-2001
```

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

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

See Also

`tbilldisc2yield` | `tbillprice` | `tbillrepo` | `tbillval01` | `tbillyield` | `tbillyield2disc` | `tbl2bond` | `tr2bonds` | `zbtprice` | `zbtyield`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Charting Financial Data” on page 2-14
- “Computing Treasury Bill Price and Yield” on page 2-41

More About

- “Treasury Bills Defined” on page 2-40

Pricing and Analyzing Equity Derivatives

In this section...

Introduction

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

Sensitivity Measures

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

Delta

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

Gamma

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

Lambda

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

Rho

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

Theta

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

Vega

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

Implied Volatility

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

Analysis Models

Toolbox functions for analyzing equity derivatives use the Black-Scholes model for European options and the binomial model for American options. The Black-Scholes model makes several assumptions about the underlying securities and their behavior. The binomial model, on the other hand, makes far fewer assumptions about the processes underlying an option. For further explanation, see Options, Futures, and Other Derivatives by John Hull in “Bibliography” on page A-2.

Black-Scholes Model

Using the Black-Scholes model entails several assumptions:

- The prices of the underlying asset follow an Ito process. (See Hull on page A-3, page 222.)

- The option can be exercised only on its expiration date (European option).
- Short selling is permitted.
- There are no transaction costs.
- All securities are divisible.
- There is no riskless arbitrage.
- Trading is a continuous process.
- The risk-free interest rate is constant and remains the same for all maturities.

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

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

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

yields:

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

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

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

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

Binomial Model

The binomial model for pricing options or other equity derivatives assumes that the probability over time of each possible price follows a binomial distribution. The basic assumption is that prices can move to only two values, one up and one down, over any short time period. Plotting the two values, and then the subsequent two values each, and then the subsequent two values each, and so on over time, is known as "building a binomial tree." This model applies to American options, which can be exercised any time up to and including their expiration date.

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

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

returns the tree of prices of the underlying asset

```
StockPrice =

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

and the tree of option values.

```
OptionPrice =

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

The output from the binomial function is a binary tree. Read the `StockPrice` matrix this way: column 1 shows the price for period 0, column 2 shows the up and down prices

for period 1, column 3 shows the up-up, up-down, and down-down prices for period 2, and so on. Ignore the zeros. The `OptionPrice` matrix gives the associated option value for each node in the price tree. Ignore the zeros that correspond to a zero in the price tree.

See Also

`binprice` | `blkimpv` | `blkprice` | `blsdelta` | `blsgamma` | `blsimpv` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega` | `opprofit`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Charting Financial Data” on page 2-14
- “Greek-Neutral Portfolios of European Stock Options” on page 10-19
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

About Life Tables

Life tables are used for life insurance and work with the probability distribution of human mortality. This distribution, which is age-dependent, has a number of characteristic features that are consequences of biological, cultural, and behavioral factors. In most cases, the practitioners of life studies use life tables that contain age-dependent series for specific demographics. The tables are in a standard format with standard notation that is specific to the life studies field. An example of a life table is shown in Table 1 from CDC life tables for the United States.

Table 1. Life table for the total population: United States, 2009

Age (years)	Probability of dying between ages x and $x+1$ q_x	Number surviving to age x l_x	Number dying between ages x and $x+1$ d_x	Person-years lived between ages x and $x+1$ L_x	Total number of person-years lived above age x T_x	Expectation of life at age x e_x
0–1	0.006372	100,000	637	99,444	7,846,926	78.5
1–2	0.000407	99,363	40	99,343	7,747,481	78.0
2–3	0.000274	99,322	27	99,309	7,648,139	77.0
3–4	0.000209	99,295	21	99,285	7,548,830	76.0
4–5	0.000160	99,274	16	99,266	7,449,545	75.0
5–6	0.000150	99,259	15	99,251	7,350,279	74.1
6–7	0.000135	99,244	13	99,237	7,251,028	73.1
7–8	0.000122	99,230	12	99,224	7,151,791	72.1
8–9	0.000109	99,218	11	99,213	7,052,566	71.1
9–10	0.000095	99,207	9	99,203	6,953,354	70.1
10–11	0.000087	99,198	9	99,194	6,854,151	69.1
11–12	0.000093	99,189	9	99,185	6,754,957	68.1
12–13	0.000127	99,180	13	99,174	6,655,773	67.1
13–14	0.000193	99,167	19	99,158	6,556,599	66.1
14–15	0.000279	99,148	28	99,134	6,457,441	65.1
15–16	0.000370	99,121	37	99,102	6,358,307	64.1
16–17	0.000454	99,084	45	99,061	6,259,205	63.2
17–18	0.000537	99,039	53	99,012	6,160,143	62.2
18–19	0.000615	98,986	61	98,955	6,061,131	61.2
19–20	0.000691	98,925	68	98,891	5,962,175	60.3

In many cases, these life tables can have numerous variations such as abridged tables (which pose challenges due to the granularity of the data) and different termination criteria (that can make it difficult to compare tables or to compute life expectancies).

Most raw life tables have one or more of the first three series in this table (q_x, l_x, and d_x) and the notation for these three series is standard in the field.

- The q_x series is basically the discrete hazard function for human mortality.
- The l_x series is the survival function multiplied by a radix of 100,000.
- The d_x series is the discrete probability density for the distribution as a function of age.

Financial Toolbox can handle arbitrary life table data supporting several standard models of mortality and provides various interpolation methods to calibrate and analyze the life table data.

Although primarily designed for life insurance applications, the life tables functions (`lifetableconv`, `lifetablefit`, and `lifetablegen` can also be used by social scientists, behavioral psychologists, public health officials, and medical researchers.

Life Tables Theory

Life tables are based on hazard functions and survival functions which are, in turn, derived from probability distributions. Specifically, given a continuous probability distribution, its cumulative distribution function is F(x) and its probability density function is f(x) = d F(x)/dx.

For the analysis of mortality, the random variable of interest X is the distribution of ages at which individuals die within a population. So, the probability that someone dies by age x is

$$\Pr[X \le x] = F(x)$$

The survival function, ($s(x)$), which characterizes the probability that an individual lives beyond a specified age $x > 0$, is

$$s(x) = Pr[X > x]$$
$$= 1 - F(x)$$

For a continuous probability distribution, the hazard function is a function of the survival function with

$$h(x) = \lim_{\Delta x \to 0} \frac{Pr[x \le X < x + \Delta x \mid X \ge x]}{\Delta x}$$
$$= -\frac{1}{s(x)} \frac{d(s(x)}{dx}$$

and the survival functions is a function of the hazard function with

$$s(x) = \exp\left(-\int_0^x h(\xi) d\xi \right)$$

Life table models generally specify either the hazard function or the survival function. However, life tables are discrete and work with discrete versions of the hazard and survival functions. Three series are used for life tables and the notation is the convention. The discrete hazard function is denoted as

$$q_x \approx h(x)$$
$$= 1 - \frac{s(x+1)}{s(x)}$$

which is the probability a person at age x dies by age $x + 1$ (where x is in years). The discrete survival function is presented in terms of an initial number of survivors at birth called the life table radix (which is usually 100,000 individuals) and is denoted as

$$l_x = l_0 s(x)$$

with radix $l_0 = 100000$. This number, l_x, represents the number of individuals out of 100,000 at birth who are still alive at age x.

A third series is related to the probability density function which is the number of "standardized" deaths in a given year denoted as

$$d_x = l_x - l_{x+1}$$

Based on a few additional rules about how to initialize and terminate these series, any one series can be derived from any of the other series.

See Also

`lifetableconv` | `lifetablefit` | `lifetablegen`

Related Examples

- “Case Study for Life Tables Analysis” on page 2-56

Case Study for Life Tables Analysis

This example shows how to use the basic workflow for life tables.

Load the life table data file.

```
load us_lifetable_2009
```

Calibrate life table from survival data with the default `heligman-pollard` parametric model.

```
a = lifetablefit(x, lx);
```

Generate life table series from the calibrated mortality model.

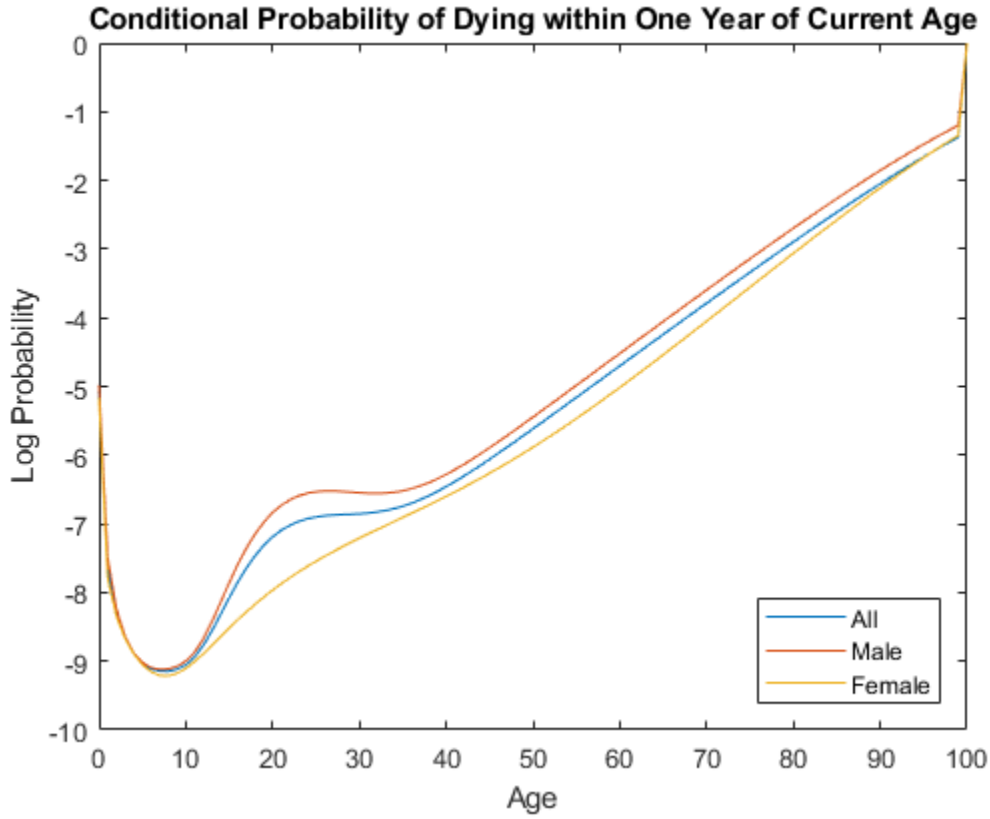
```
qx = lifetablegen((0:100), a);
display(qx(1:40,:))
```

```
    0.0063    0.0069    0.0057
    0.0005    0.0006    0.0004
    0.0002    0.0003    0.0002
    0.0002    0.0002    0.0002
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0002    0.0002    0.0001
    0.0002    0.0002    0.0002
    0.0002    0.0003    0.0002
    0.0003    0.0004    0.0002
    0.0004    0.0005    0.0002
    0.0005    0.0006    0.0003
    0.0006    0.0008    0.0003
    0.0007    0.0009    0.0003
    0.0008    0.0011    0.0003
    0.0008    0.0012    0.0004
    0.0009    0.0013    0.0004
    0.0009    0.0014    0.0005
    0.0010    0.0014    0.0005
    0.0010    0.0015    0.0005
```

```
    0.0010    0.0015    0.0006
    0.0010    0.0015    0.0006
    0.0010    0.0015    0.0007
    0.0010    0.0014    0.0007
    0.0011    0.0014    0.0007
    0.0011    0.0014    0.0008
    0.0011    0.0014    0.0008
    0.0011    0.0014    0.0009
    0.0011    0.0014    0.0009
    0.0012    0.0015    0.0010
    0.0012    0.0015    0.0011
    0.0013    0.0016    0.0011
    0.0014    0.0017    0.0012
    0.0015    0.0018    0.0013
```

Plot the `qx` series and display the legend. The series `qx` is the conditional probability that a person at age x will die between age x and the next age in the series

```
plot((0:100), log(qx));
legend(series, 'location', 'southeast');
title('Conditional Probability of Dying within One Year of Current Age');
xlabel('Age');
ylabel('Log Probability');
```



See Also

`lifetableconv` | `lifetablefit` | `lifetablegen`

More About

- “About Life Tables” on page 2-53

Portfolio Analysis

Analyzing Portfolios

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

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

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

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

Warning `frontcon` has been removed. Use `Portfolio` instead.

`portopt` has been partially removed and will no longer accept `ConSet` or `varargin` arguments. `portopt` will only solve the portfolio problem for long-only fully invested portfolios. Use `Portfolio` instead.

See Also

`Portfolio` | `abs2active` | `active2abs` | `frontier` | `pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portalloc` | `portcons` | `portopt` | `portvrisk`

Related Examples

- “Portfolio Optimization Functions” on page 3-4

- “Portfolio Construction Examples” on page 3-7
- “Portfolio Selection and Risk Aversion” on page 3-9
- “Active Returns and Tracking Error Efficient Frontier” on page 3-43
- “Plotting an Efficient Frontier Using portopt” on page 10-27

More About

- “Portfolio Object Workflow” on page 4-21

Portfolio Optimization Functions

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

Capital Allocation	Description
`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	**Description**
`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. **Warning** `frontcon` has been removed. Use `Portfolio` instead. For more information on migrating `frontcon` code to `Portfolio`, see “frontcon Migration to Portfolio Object” on page 3-25.
`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. **Warning** `portopt` has been partially removed and will no longer accept `ConSet` or `varargin` arguments. `portopt` will only solve the portfolio problem for long-only fully invested portfolios. Use `Portfolio` instead. For more information on migrating `portopt` code to `Portfolio`, see “portopt Migration to Portfolio Object” on page 3-14.

Constraint Specification	Description
`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.
`portvrisk`	Portfolio value at risk (VaR) returns the maximum potential loss in the value of a portfolio over one period of time, given the loss probability level `RiskThreshold`.
`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	**Description**
`abs2active`	Transforms a constraint matrix expressed in absolute weight format to an equivalent matrix expressed in active weight format.
`active2abs`	Transforms a constraint matrix expressed in active weight format to an equivalent matrix expressed in absolute weight format.

Note An alternative to using these portfolio optimization functions is to use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

See Also

`Portfolio` | `abs2active` | `active2abs` | `frontier` | `pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portalloc` | `portcons` | `portopt` | `portvrisk`

Related Examples

- “Portfolio Construction Examples” on page 3-7
- “Portfolio Selection and Risk Aversion” on page 3-9
- “Active Returns and Tracking Error Efficient Frontier” on page 3-43
- “Plotting an Efficient Frontier Using portopt” on page 10-27
- “portopt Migration to Portfolio Object” on page 3-14
- “frontcon Migration to Portfolio Object” on page 3-25

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-21

Portfolio Construction Examples

In this section...

Introduction

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

Note An alternative to using these portfolio optimization functions is to use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

Efficient Frontier Example

`frontcon` has been removed. To model the efficient frontier, use the `Portfolio` object instead. For example, using the `Portfolio` object, you can model an efficient frontier:

- "Obtaining Portfolios Along the Entire Efficient Frontier" on page 4-109
- "Obtaining Endpoints of the Efficient Frontier" on page 4-112
- "Obtaining Efficient Portfolios for Target Returns" on page 4-115
- "Obtaining Efficient Portfolios for Target Risks" on page 4-119
- "Efficient Portfolio That Maximizes Sharpe Ratio" on page 4-123
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129

- “Plotting the Efficient Frontier for a Portfolio Object” on page 4-132

See Also

`Portfolio` | `abs2active` | `active2abs` | `frontier` | `pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portalloc` | `portcons` | `portopt` | `portvrisk`

Related Examples

- “Portfolio Optimization Functions” on page 3-4
- “Portfolio Selection and Risk Aversion” on page 3-9
- “Active Returns and Tracking Error Efficient Frontier” on page 3-43
- “Plotting an Efficient Frontier Using portopt” on page 10-27
- “portopt Migration to Portfolio Object” on page 3-14
- “frontcon Migration to Portfolio Object” on page 3-25

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-21

Portfolio Selection and Risk Aversion

In this section...

Introduction

One of the factors to consider when selecting the optimal portfolio for a particular investor is the 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 requires to make an increment in risk worthwhile. Typical risk aversion coefficients range from 2.0 through 4.0, with the higher number representing lesser tolerance to risk. The equation used to represent risk aversion in Financial Toolbox software is

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

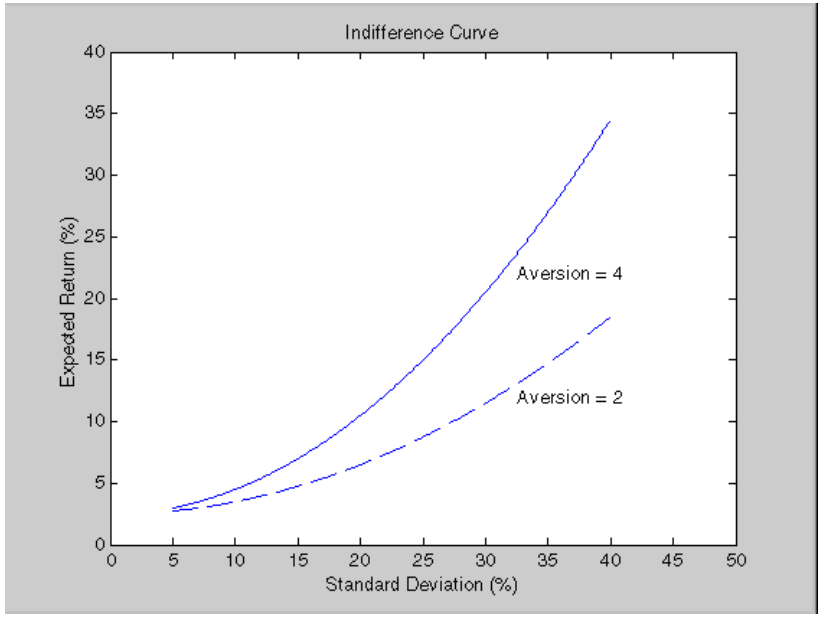
where:

`U` is the utility value.

`E(r)` is the expected return.

`A` is the index of investor's aversion.

`sig` is the standard deviation.



Note An alternative to using these portfolio optimization functions is to use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

Optimal Risky Portfolio

This example 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. You do this with the function `portalloc`.

First generate the efficient frontier data using `portopt`.

```
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];
```

Consider 20 different points along the efficient frontier.

```
NumPorts = 20;

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

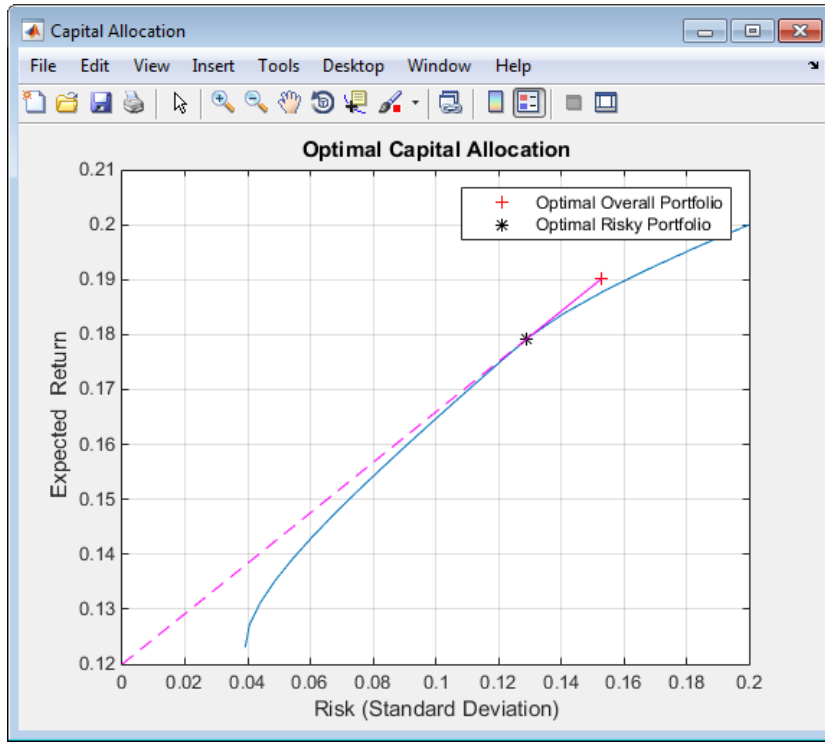
Calling `portopt`, while specifying output arguments, returns the corresponding vectors and arrays representing the risk, return, and weights for each of the portfolios along the efficient frontier. Use these as the first three input arguments to the function `portalloc`.

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 is invested in the risk-free asset. This borrowed capital is added to the original capital available for investment. In this example, the customer tolerates borrowing 18.69% of the original capital amount.

See Also

`Portfolio` | `abs2active` | `active2abs` | `frontier` | `pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portalloc` | `portcons` | `portopt` | `portvrisk`

Related Examples

- “Portfolio Optimization Functions” on page 3-4
- “Portfolio Selection and Risk Aversion” on page 3-9
- “Active Returns and Tracking Error Efficient Frontier” on page 3-43
- “Plotting an Efficient Frontier Using portopt” on page 10-27
- “portopt Migration to Portfolio Object” on page 3-14
- “frontcon Migration to Portfolio Object” on page 3-25

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-21

portopt Migration to Portfolio Object

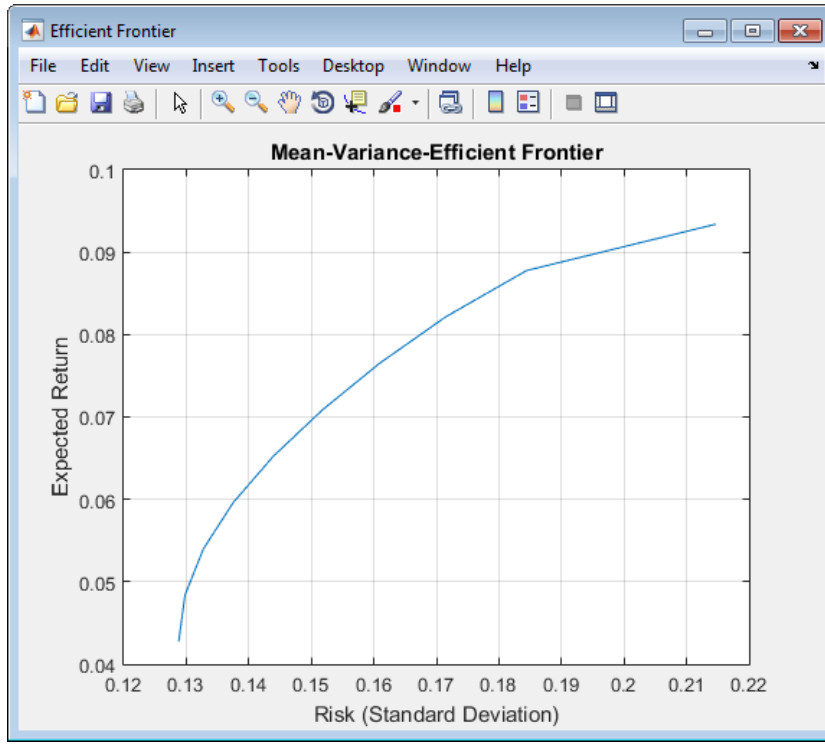
In this section...

Migrate portopt Without Output Arguments

This example shows how to migrate `portopt` without output arguments to a Portfolio object.

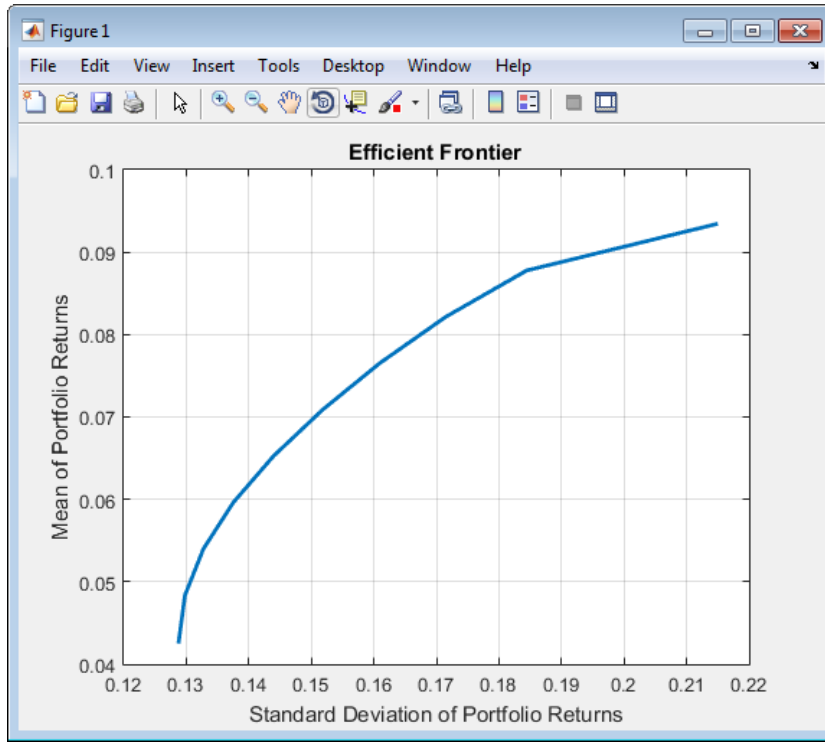
The basic `portopt` functionality is represented as:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

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



To migrate a `portopt` syntax without output arguments to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

plotFrontier(p, NumPorts);
```



Without output arguments, `portopt` plots the efficient frontier. The Portfolio object has similar behavior although the Portfolio object writes to the current figure window rather than create a new window each time a plot is generated.

Migrate portopt with Output Arguments

This example shows how to migrate `portopt` with output arguments to a Portfolio object.

With output arguments, the basic functionality of `portopt` returns portfolio moments and weights. Once the Portfolio object is set up, moments and weights are obtained in separate steps.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
```

```
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

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

display(PortWts);

PortWts =

    0.2103    0.2746    0.1157    0.1594    0.2400
    0.1744    0.2657    0.1296    0.2193    0.2110
    0.1386    0.2567    0.1436    0.2791    0.1821
    0.1027    0.2477    0.1575    0.3390    0.1532
    0.0668    0.2387    0.1714    0.3988    0.1242
    0.0309    0.2298    0.1854    0.4587    0.0953
         0    0.2168    0.1993    0.5209    0.0629
         0    0.1791    0.2133    0.5985    0.0091
         0    0.0557    0.2183    0.7260         0
         0         0         0    1.0000         0
```

To migrate a `portopt` syntax with output arguments:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

display(PortWts);

PortWts =

    0.2103    0.1744    0.1386    0.1027    0.0668    0.0309         0         0         0         0
    0.2746    0.2657    0.2567    0.2477    0.2387    0.2298    0.2168    0.1791    0.0557         0
    0.1157    0.1296    0.1436    0.1575    0.1714    0.1854    0.1993    0.2133    0.2183         0
    0.1594    0.2193    0.2791    0.3390    0.3988    0.4587    0.5209    0.5985    0.7260    1.0000
    0.2400    0.2110    0.1821    0.1532    0.1242    0.0953    0.0629    0.0091         0         0
```

The Portfolio object returns `PortWts` with portfolios going down columns, not across rows. Portfolio risks and returns are still in column format.

Migrate portopt for Target Returns Within Range of Efficient Portfolio Returns

This example shows how to migrate `portopt` target returns within range of efficient portfolio returns to a Portfolio object.

`portopt` can obtain portfolios with specific targeted levels of return but requires that the targeted returns fall within the range of efficient returns. The Portfolio object handles this by selecting portfolios at the ends of the efficient frontier.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09 ];

[PortRisk, PortReturn, PortWts] = portopt(ExpReturn, ExpCovariance, [], TargetReturn);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
 Efficient    Target
    0.0500    0.0500
    0.0600    0.0600
    0.0700    0.0700
    0.0800    0.0800
    0.0900    0.0900
```

To migrate a `portopt` syntax for target returns within range of efficient portfolio returns to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;
```

```
TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09 ];

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

PortWts = estimateFrontierByReturn(p, TargetReturn);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
 Efficient    Target
    0.0500    0.0500
    0.0600    0.0600
    0.0700    0.0700
    0.0800    0.0800
    0.0900    0.0900
```

Migrate portopt for Target Return Outside Range of Efficient Portfolio Returns

This example shows how to migrate `portopt` target returns outside of range of efficient portfolio returns to a Portfolio object.

When the target return is outside of the range of efficient portfolio returns, `portopt` generates an error. The Portfolio object handles this effectively by selecting portfolios at the ends of the efficient frontier.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09; 0.10 ];

[PortRisk, PortReturn, PortWts] = portopt(ExpReturn, ExpCovariance, [], TargetReturn);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
> In portopt at 85
Error using portopt (line 297)
One or more requested returns are greater than the maximum achievable return of 0.093400.
```

To migrate a `portopt` syntax for target returns outside of the range of efficient portfolio returns to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09; 0.10 ];

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

PortWts = estimateFrontierByReturn(p, TargetReturn);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
Warning: One or more target return values are outside the feasible range [
0.0427391, 0.0934 ].
    Will return portfolios associated with endpoints of the range for these
    values.
> In Portfolio/estimateFrontierByReturn (line 106)
 Efficient    Target
    0.0500    0.0500
    0.0600    0.0600
    0.0700    0.0700
    0.0800    0.0800
    0.0900    0.0900
    0.0934    0.1000
```

Migrate portopt Using portcons Output for ConSet

This example shows how to migrate `portopt` when the `ConSet` output from `portcons` is used with `portopt`.

`portopt` accepts as input the outputs from `portcons`, `pcalims`, `pcglims`, and `pcgcomp`. This example focuses on `portcons`. `portcons` sets up linear constraints for `portopt` in the form `A*Port <= b`. In a matrix `ConSet = [ A, b ]` and break into separate `A` and `b` arrays with `A = ConSet(:,1:end-1);` and `b = ConSet(:,end);`. In addition, to illustrate default problem with additional group constraints, consider three groups. Assets 2, 3, and 4 can constitute up to 80% of portfolio, Assets 1 and 2 can constitute up to 70% of portfolio, and Assets 3, 4, and 5 can constitute up to 90% of portfolio.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];
```

```
ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];

LowerGroup = GroupBounds(:,1);
UpperGroup = GroupBounds(:,2);

ConSet = portcons('default', 5, 'grouplims', Groups, LowerGroup, UpperGroup);

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

disp([PortRisk, PortReturn]);
```

```
Error using portopt (line 83)
In the current and future releases, portopt will no longer accept ConSet or varargin arguments.
'It will only solve the portfolio problem for long-only fully-invested portfolios.
To solve more general problems, use the Portfolio object.
See the release notes for details, including examples to make the conversion.
```

To migrate `portopt` to a Portfolio object when the `ConSet` output from `portcons` is used with `portopt`:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];

LowerGroup = GroupBounds(:,1);
UpperGroup = GroupBounds(:,2);

ConSet = portcons('default', 5, 'grouplims', Groups, LowerGroup, UpperGroup);

A = ConSet(:,1:end-1);
b = ConSet(:,end);

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setInequality(p, A, b);                    % implement group constraints here

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp([PortRisk, PortReturn]);
```

```
0.1288    0.0427
0.1292    0.0465
```

```
0.1306    0.0503
0.1328    0.0540
0.1358    0.0578
0.1395    0.0615
0.1440    0.0653
0.1504    0.0690
0.1590    0.0728
0.1806    0.0766
```

The constraints are entered directly into the Portfolio object with the `setInequality` or `addInequality` functions.

Integrate Output from portcons pcalims, pcglims, and pcgcomp with a Portfolio Object

This example shows how to integrate output from `pcalims`, `pcalims`, `pcglims`, or `pcgcomp` with a Portfolio object implementation.

`portcons`, `pcalims`, `pcglims`, and `pcgcomp` setup linear constraints for `portopt` in the form `A*Port <= b`. Although some functions permit two outputs, assume that the output is a single matrix `ConSet`. Break into separate `A` and `b` arrays with:

- `A = ConSet(:,1:end-1);`
- `b = ConSet(:,end);`

In addition, to illustrate default problem with additional group constraints, consider three groups:

- Assets 2, 3, and 4 can constitute up to 80% of portfolio.
- Assets 1 and 2 can constitute up to 70% of portfolio.
- Assets 3, 4, and 5 can constitute up to 90% of portfolio.

```
Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];
```

To integrate the `ConSet` output of `portcons` with a Portfolio object implementation:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
```

```
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];

LowerGroup = GroupBounds(:,1);
UpperGroup = GroupBounds(:,2);

ConSet = portcons('default', 5, 'grouplims', Groups, LowerGroup, UpperGroup);

A = ConSet(:,1:end-1);
b = ConSet(:,end);

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);               % implement default constraints here
p = setInequality(p, A, b);                   % implement group constraints here

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp([PortRisk, PortReturn]);
```

```
    0.1288    0.0427
    0.1292    0.0465
    0.1306    0.0503
    0.1328    0.0540
    0.1358    0.0578
    0.1395    0.0615
    0.1440    0.0653
    0.1504    0.0690
    0.1590    0.0728
    0.1806    0.0766
```

To integrate the output of `pcalims` and `pcglims` with a Portfolio object implementation:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];

LowerGroup = GroupBounds(:,1);
UpperGroup = GroupBounds(:,2);

AssetMin = [ 0; 0; 0; 0; 0 ];
AssetMax = [ 0.8; 0.8; 0.8; 0.8; 0.8 ];

[Aa, ba] = pcalims(AssetMin, AssetMax);
```

```
[Ag, bg] = pcglims(Groups, LowerGroup, UpperGroup);

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);               % implement default constraints first
p = addInequality(p, Aa, ba);               % implement bound constraints here
p = addInequality(p, Ag, bg);               % implement group constraints here

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp([PortRisk, PortReturn]);
```

```
0.1288    0.0427
0.1292    0.0465
0.1306    0.0503
0.1328    0.0540
0.1358    0.0578
0.1395    0.0615
0.1440    0.0653
0.1504    0.0690
0.1590    0.0728
0.1806    0.0766
```

See Also

`Portfolio` | `addInequality` | `estimateFrontier` | `estimateFrontierByReturn` | `estimatePortMoments` | `pcalims` | `pcgcomp` | `pcglims` | `portcons` | `portopt` | `setAssetMoments` | `setDefaultConstraints` | `setInequality`

Related Examples

- “frontcon Migration to Portfolio Object” on page 3-25

More About

- “Portfolio Object Workflow” on page 4-21

frontcon Migration to Portfolio Object

In this section...

Migrate frontcon Without Output Arguments

This example shows how to migrate `frontcon` without output arguments to a Portfolio object.

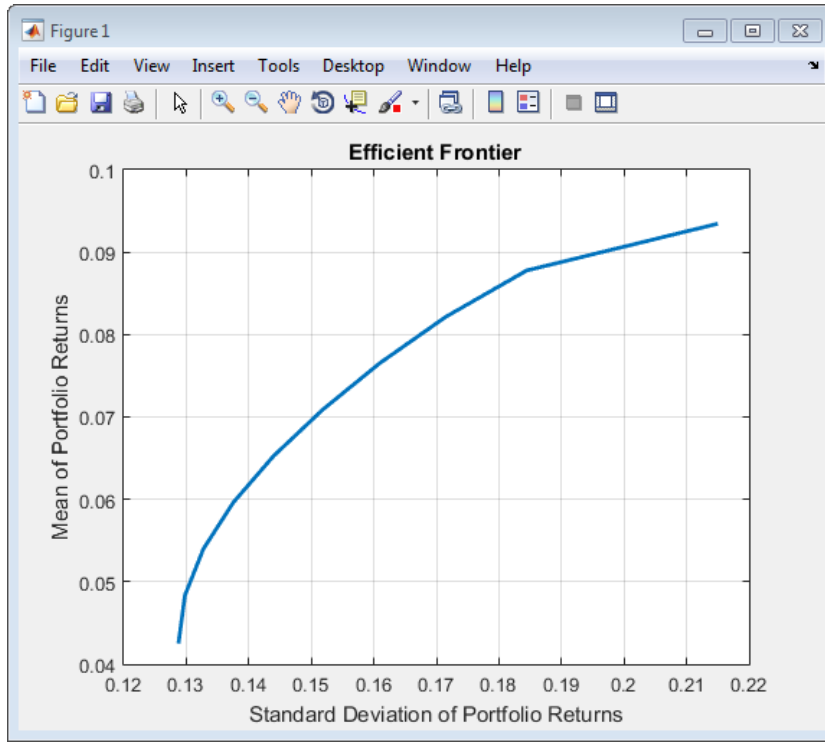
The basic `frontcon` functionality is represented as:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

frontcon(ExpReturn, ExpCovariance, NumPorts);

Undefined function or variable 'frontcon'.
```

To migrate a `frontcon` syntax without output arguments to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];
```

```
NumPorts = 10;

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

plotFrontier(p, NumPorts);
```



The Portfolio object writes to the current figure window rather than create a new window each time a plot is generated.

Migrate frontcon with Output Arguments

This example shows how to migrate `frontcon` with output arguments to a Portfolio object.

The basic `frontcon` functionality is represented as:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;


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

display(PortWts);

Undefined function or variable 'frontcon'.
```

To migrate a `frontcon` syntax with output arguments:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

display(PortWts);

PortWts =

    0.2103    0.1744    0.1386    0.1027    0.0668    0.0309         0         0         0         0
    0.2746    0.2657    0.2567    0.2477    0.2387    0.2298    0.2168    0.1791    0.0557         0
    0.1157    0.1296    0.1436    0.1575    0.1714    0.1854    0.1993    0.2133    0.2183         0
    0.1594    0.2193    0.2791    0.3390    0.3988    0.4587    0.5209    0.5985    0.7260    1.0000
    0.2400    0.2110    0.1821    0.1532    0.1242    0.0953    0.0629    0.0091         0         0
```

The Portfolio object returns `PortWts` with portfolios going down columns, not across rows. Portfolio risks and returns are still in column format.

Migrate frontcon for Target Returns Within Range of Efficient Portfolio Returns

This example shows how to migrate `frontcon` target returns within range of efficient portfolio returns to a Portfolio object.

`frontcon` can obtain portfolios with specific targeted levels of return but requires that the targeted returns fall within the range of efficient returns. The Portfolio object handles this by selecting portfolios at the ends of the efficient frontier.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09 ];

[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn, ExpCovariance, [], TargetReturn);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
Undefined function or variable 'frontcon'.
```

To migrate a `frontcon` syntax for target returns within range of efficient portfolio returns to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;
TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09 ];

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

PortWts = estimateFrontierByReturn(p, TargetReturn);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
Efficient    Target
    0.0500    0.0500
    0.0600    0.0600
    0.0700    0.0700
```

```
    0.0800    0.0800
    0.0900    0.0900
```

Migrate frontcon for Target Returns Outside Range of Efficient Portfolio Returns

This example shows how to migrate `frontcon` target returns outside of range of efficient portfolio returns to a Portfolio object.

When the target return is outside of the range of efficient portfolio returns, `frontcon` generates an error. The Portfolio object handles this effectively by selecting portfolios at the ends of the efficient frontier.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09; 0.10 ];

[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn, ExpCovariance, [], TargetReturn);

disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
Undefined function or variable 'frontcon'.
```

To migrate a `frontcon` syntax for target returns outside of the range of efficient portfolio returns to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

TargetReturn = [ 0.05; 0.06; 0.07; 0.08; 0.09; 0.10 ];

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);

PortWts = estimateFrontierByReturn(p, TargetReturn);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);
```

```
disp(' Efficient    Target');
disp([PortReturn, TargetReturn]);
```

```
Warning: One or more target return values are outside the feasible range [
0.0427391, 0.0934 ].
    Will return portfolios associated with endpoints of the range for these
    values.
> In Portfolio/estimateFrontierByReturn (line 106)
 Efficient    Target
    0.0500    0.0500
    0.0600    0.0600
    0.0700    0.0700
    0.0800    0.0800
    0.0900    0.0900
    0.0934    0.1000
```

Migrate frontcon Syntax When Using Bounds

This example shows how to migrate `frontcon` syntax for `AssetBounds` to a Portfolio object.

Use `frontcon` with an input specification for `AssetBounds` that contains the lower and upper bounds on the weight allocated to each asset in the portfolio:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

AssetBounds = [ 0.1, 0.1, 0.1, 0.1, 0.1; 0.5, 0.5, 0.5, 0.5, 0.5 ];

[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn, ExpCovariance, NumPorts, [], AssetBounds);

disp([PortRisk, PortReturn]);
```

```
Undefined function or variable 'frontcon'.
```

To migrate a `frontcon` syntax using `AssetBounds` to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;
```

```
AssetBounds = [ 0.1, 0.1, 0.1, 0.1, 0.1; 0.5, 0.5, 0.5, 0.5, 0.5 ];

LowerBound = AssetBounds(1,:);
UpperBound = AssetBounds(2,:);

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);
p = setBounds(p, LowerBound, UpperBound);

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp([PortRisk, PortReturn]);
```

```
0.1288    0.0427
0.1291    0.0457
0.1299    0.0487
0.1313    0.0516
0.1332    0.0546
0.1356    0.0576
0.1385    0.0605
0.1419    0.0635
0.1461    0.0665
0.1519    0.0694
```

Migrate frontcon Syntax When Using Groups

This example shows how to migrate `frontcon` syntax for `Groups` and `GroupBounds` to a Portfolio object.

Use `frontcon` with an input specification for `Groups` (asset groups or classes.) and `GroupBounds` (the lower and upper bounds of the total weights of all assets in a group). Consider three groups: Assets 2, 3, and 4 can constitute up to 80% of a portfolio, Assets 1 and 2 can constitute up to 70% of a portfolio, and Assets 3, 4, and 5 can constitute up to 90% of a portfolio.

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;
```

```
Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];

[PortRisk, PortReturn, PortWgts] = frontcon(ExpReturn, ExpCovariance, NumPorts, [], [], ...
    Groups, GroupBounds);

disp([PortRisk, PortReturn]);

Undefined function or variable 'frontcon'.
```

To migrate a `frontcon` syntax using `Groups` and `GroupBounds` to a Portfolio object:

```
ExpReturn = [ 0.0054; 0.0531; 0.0779; 0.0934; 0.0130 ];

ExpCovariance = [ 0.0569,  0.0092,  0.0039,  0.0070,  0.0022;
    0.0092,  0.0380,  0.0035,  0.0197,  0.0028;
    0.0039,  0.0035,  0.0997,  0.0100,  0.0070;
    0.0070,  0.0197,  0.0100,  0.0461,  0.0050;
    0.0022,  0.0028,  0.0070,  0.0050,  0.0573 ];

NumPorts = 10;

Groups = [ 0 1 1 1 0; 1 1 0 0 0; 0 0 1 1 1 ];
GroupBounds = [ 0, 0.8; 0, 0.7; 0, 0.9 ];

LowerGroup = GroupBounds(:,1);
UpperGroup = GroupBounds(:,2);

p = Portfolio;
p = setAssetMoments(p, ExpReturn, ExpCovariance);
p = setDefaultConstraints(p);
p = setGroups(p, Groups, LowerGroup, UpperGroup);

PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

disp([PortRisk, PortReturn]);
```

```
0.1288    0.0427
0.1292    0.0465
0.1306    0.0503
0.1328    0.0540
0.1358    0.0578
0.1395    0.0615
0.1440    0.0653
0.1504    0.0690
0.1590    0.0728
0.1806    0.0766
```

See Also

`Portfolio` | `addInequality` | `estimateFrontier` | `estimateFrontierByReturn` | `estimatePortMoments` | `pcalims` | `pcgcomp` | `pcglims` | `portcons` | `portopt` |

`setAssetMoments` | `setBounds` | `setDefaultConstraints` | `setGroups` | `setInequality`

Related Examples

- "portopt Migration to Portfolio Object" on page 3-14

More About

- "Portfolio Object Workflow" on page 4-21

Constraint Specification Using a Portfolio Object

In this section...

Constraints for Efficient Frontier

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

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

The covariance matrix is

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

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

Asset	INTC	XON	RD
Sector	Technology	Energy	Energy

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

    To solve this problem, use `Portfolio`, 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. Since the bounds are the same for each asset, only one pair of bounds is needed because of scalar expansion.

```
AssetBounds = [-0.1, 1.1];
```

Constraint 2 must 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 `1`s and `0`s 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 `Portfolio` to obtain the vectors and arrays representing the risk, return, and weights for each of the eight portfolios computed along the efficient frontier. A budget constraint is added to ensure that the portfolio weights sum to 1.

```
p = Portfolio('AssetMean', ExpReturn, 'AssetCovar', ExpCovariance);
p = setBounds(p, AssetBounds(1), AssetBounds(2));
p = setBudget(p, 1, 1);
p = setGroups(p, Groups, GroupBounds(:,1), GroupBounds(:,2));

PortWts = estimateFrontier(p, NumPorts);

[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

PortRisk
PortReturn
PortWts
```

```
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    0.6031    0.4864    0.3696    0.2529    0.2000    0.2000    0.2000
    0.2582    0.3244    0.3708    0.4172    0.4636    0.5738    0.7369    0.9000
    0.0418    0.0725    0.1428    0.2132    0.2835    0.2262    0.0631   -0.1000
```

The outputs are represented as columns for the portfolio's risk and return. Portfolio weights are identified as corresponding column vectors in a matrix.

Linear Constraint Equations

While the `Portfolio` object 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. `Portfolio` also addresses this need, by accepting an arbitrary set of constraints.

This example requires specifying the minimum and maximum investment in various groups.

Maximum and Minimum Group Exposure

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

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

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

Group Membership

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

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

Specifically

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

Since you must 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.	$-W1 - W2$	≤	-0.30
2.	$W1 + W2$	≤	0.75
3.	$-W3$	≤	-0.10
4.	$W3$	≤	0.55
5.	$-W1$	≤	-0.20
6.	$W1$	≤	0.50
7.	$-W2 - W3$	≤	-0.50
8.	$W2 + W3$	≤	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]
```

One approach to solving this portfolio problem is to explicitly use the `setInequality` function:

```
p = Portfolio('AssetMean', ExpReturn, 'AssetCovar', ExpCovariance);
p = setBounds(p, AssetBounds(1), AssetBounds(2));
p = setBudget(p, 1, 1);
p = setInequality(p, A, b);
PortWts = estimateFrontier(p, NumPorts);
```

```
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

PortRisk
PortReturn
PortWts
```

```
PortRisk =

    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586


PortReturn =

    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375


PortWts =

    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500
    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500
```

In this case, the constraints allow only one optimum portfolio. Since eight portfolios were requested, all eight portfolios are the same. Note that the solution to this portfolio problem using the `setInequality` function is the same as using the `setGroups` function in the next example (“Specifying Group Constraints” on page 3-39).

Specifying Group Constraints

The example above (“Linear Constraint Equations” on page 3-36) defines a constraint matrix that specifies a set of typical scenarios. It defines groups of assets, specifies upper and lower bounds for total allocation in each of these groups, and it sets the total allocation of one group to a fixed value. Constraints like these are common occurrences. `Portfolio` enables you to simplify the creation of the constraint matrix for these and other common portfolio requirements.

An alternative approach for solving the portfolio problem is to use the `Portfolio` object to define:

- A `Group` matrix, indicating the assets that belong to each group.
- A `GroupMin` vector, indicating the minimum bounds for each group.
- A `GroupMax` vector, indicating the maximum bounds for each group.

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

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

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

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

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

```
p = Portfolio('AssetMean', ExpReturn, 'AssetCovar', ExpCovariance);
p = setBounds(p, AssetBounds(1), AssetBounds(2));
p = setBudget(p, 1, 1);
p = setGroups(p, Group, GroupMin, GroupMax);
PortWts = estimateFrontier(p, NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p, PortWts);

PortRisk
PortReturn
PortWts
```

```
PortRisk =

    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586


PortReturn =

    0.1375
```

```
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375


PortWts =

    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500
    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500
```

In this case, the constraints allow only one optimum portfolio. Since eight portfolios were requested, all eight portfolios are the same. Note that the solution to this portfolio problem using the `setGroups` function is the same as using the `setInequality` function in the previous example ("Linear Constraint Equations" on page 3-36).

See Also

`Portfolio` | `estimateFrontier` | `estimatePortMoments` | `setGroups` | `setInequality`

Related Examples

- "Setting Default Constraints for Portfolio Weights Using Portfolio Object" on page 4-67
- "Working with Bound Constraints Using Portfolio Object" on page 4-72
- "Working with Budget Constraints Using Portfolio Object" on page 4-75
- "Working with Group Constraints Using Portfolio Object" on page 4-78
- "Working with Group Ratio Constraints Using Portfolio Object" on page 4-82
- "Working with Linear Equality Constraints Using Portfolio Object" on page 4-86
- "Working with Linear Inequality Constraints Using Portfolio Object" on page 4-89
- "Working with Average Turnover Constraints Using Portfolio Object" on page 4-92
- "Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-96
- "Working with Tracking Error Constraints Using Portfolio Object" on page 4-100
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- “Portfolio Set for Optimization Using Portfolio Object” on page 4-10
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

Active Returns and Tracking Error Efficient Frontier

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

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

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

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

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

```
NumAssets    =  5;

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

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

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

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

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

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

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

Generate an asset constraint matrix using `portcons`. The constraint matrix `AbsConSet` is expressed in absolute format (unadjusted for the index), and is formatted as `[A b]`, corresponding to constraints of the form `A*w <= 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 differ only in the last column (the columns corresponding to the `b` in `A*w <= b`).

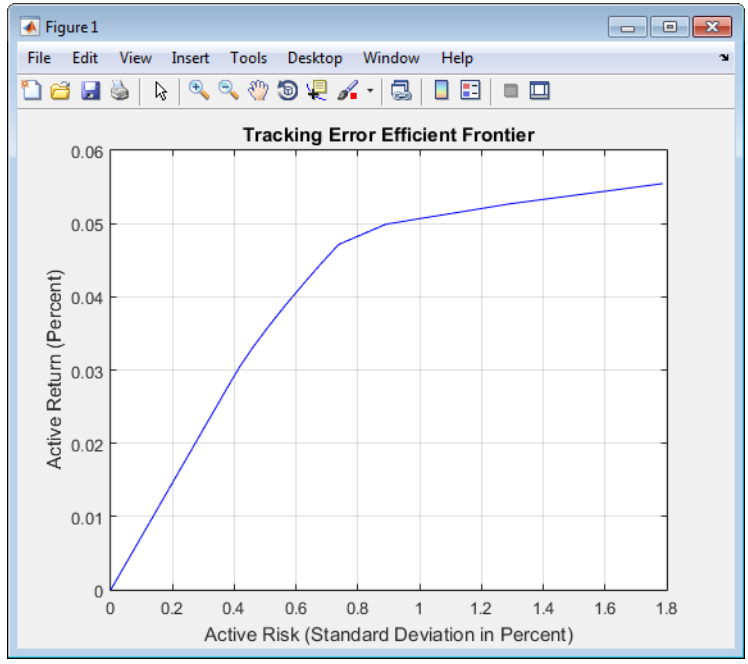
```
[AbsConSet(:,end)  ActiveConSet(:,end)]

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:

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

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

```
p = Portfolio('AssetMean', ExpReturn, 'AssetCovar', ExpCovariance);
p = p.setInequality(ActiveConSet(:,1:end-1), ActiveConSet(:,end));
[ActiveRisk, ActiveReturn] = p.plotFrontier(21);

plot(ActiveRisk*100, ActiveReturn*100, 'blue')
grid('on')
xlabel('Active Risk (Standard Deviation in Percent)')
ylabel('Active Return (Percent)')
title('Tracking Error Efficient Frontier')
```



Of particular interest is the lower-left portfolio along the frontier. This zero-risk/zero-return portfolio has a practical economic significance. It represents a full investment in the index portfolio itself. 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.

```
ActiveWeights = p.estimateFrontier(21);
AbsoluteWeights = ActiveWeights + repmat(Index, 1, 21);
```

See Also

`Portfolio` | `abs2active` | `active2abs` | `estimateFrontier` | `frontier` | `pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `plotFrontier` | `portalloc` | `portcons` | `portvrisk` | `setInequality`

Related Examples

- “Portfolio Optimization Functions” on page 3-4
- “Portfolio Selection and Risk Aversion” on page 3-9
- “Plotting an Efficient Frontier Using portopt” on page 10-27

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-21

Mean-Variance Portfolio Optimization Tools

Portfolio Optimization Theory

In this section...

Portfolio Optimization Problems

Portfolio optimization problems involve identifying portfolios that satisfy three criteria:

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

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

The set of feasible portfolios is necessarily a nonempty, closed, and bounded set. The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices. The proxy for return is a function that characterizes either the gross or net benefits associated with portfolio choices. The terms “risk” and “risk proxy” and “return” and “return proxy” are interchangeable. The fundamental insight of Markowitz (see “Portfolio Optimization” on page A-7) is that the goal of the portfolio choice problem is to seek minimum risk for a given level of return and to seek maximum return for a given level of risk. Portfolios satisfying these criteria are efficient portfolios and the graph of the risks and returns of these portfolios forms a curve called the efficient frontier.

Portfolio Problem Specification

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

- Proxy for portfolio return (μ)

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

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

- The Portfolio object (`Portfolio`) supports mean-variance portfolio optimization (see Markowitz [46], [47] at "Portfolio Optimization" on page A-7). This object has either gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set.
- The PortfolioCVaR object (`PortfolioCVaR`) implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at "Portfolio Optimization" on page A-7), which is referred to as CVaR portfolio optimization. CVaR portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses conditional value-at-risk of portfolio returns as the risk proxy.
- The PortfolioMAD object (`PortfolioMAD`) implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7), which is referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

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

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

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

and (sample) covariance of asset returns

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

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

Gross Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

Net Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

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

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

Risk Proxy

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

Variance

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

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

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

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

Although the risk proxy in mean-variance portfolio optimization is the variance of portfolio returns, the square root, which is the standard deviation of portfolio returns, is often reported and displayed. Moreover, this quantity is often called the "risk" of the portfolio. For details, see Markowitz ("Portfolio Optimization" on page A-7).

Conditional Value-at-Risk

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

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

where:

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

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

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

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

The value-at-risk is defined as

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

An alternative formulation for CVaR has the form:

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

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

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

To describe the probability distribution of returns, the `PortfolioCVaR` object takes a finite sample of return scenarios y_s, with $s = 1,\ldots,S$. Each y_s is an n vector that contains the returns for each of the n assets under the scenario s. This sample of S scenarios is stored as a scenario matrix of size S-by-n. Then, the risk proxy for CVaR portfolio optimization, for a given portfolio $x \in X$ and $\alpha \in (0,1)$, is computed as

$$\sum(x) = VaR_\alpha(x) + \frac{1}{(1-\alpha)S}\sum_{s=1}^{S} \max\left\{0, -y_s^T x - VaR_\alpha(x)\right\}$$

The value-at-risk, $VaR_\alpha(x)$, is estimated whenever the CVaR is estimated. The loss function is $f(x, y_s) = -y_s^T x$, which is the portfolio loss under scenario s.

Under this definition, VaR and CVaR are sample estimators for VaR and CVaR based on the given scenarios. Better scenario samples yield more reliable estimates of VaR and CVaR.

For more information, see Rockafellar and Uryasev [48], [49], and Cornuejols and Tütüncü, [51], at "Portfolio Optimization" on page A-7.

Mean Absolute-Deviation

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

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

where:

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

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

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

such that

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

For more information, see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7.

See Also

`Portfolio`

Related Examples

- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- `Portfolio`
- "Portfolio Object Workflow" on page 4-21
- "Portfolio Set for Optimization Using Portfolio Object" on page 4-10
- "Default Portfolio Problem" on page 4-19

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Set for Optimization Using Portfolio Object

The final element for a complete specification of a portfolio optimization problem is the

set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset R^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

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

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

Linear Inequality Constraints

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

$A_I x \le b_I$

where:

x is the portfolio (n vector).

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

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

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

Portfolio object properties to specify linear inequality constraints are:

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

The default is to ignore these constraints.

Linear Equality Constraints

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

$$A_E x = b_E$$

where:

x is the portfolio (n vector).

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

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

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

Portfolio object properties to specify linear equality constraints are:

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

The default is to ignore these constraints.

Bound Constraints

Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Since every portfolio set must be bounded, it is

often a good practice, albeit not necessary, to set explicit bounds for the portfolio problem. To obtain explicit bounds for a given portfolio set, use the `estimateBounds` function. Bound constraints take the form

$$l_B \le x \le u_B$$

where:

x is the portfolio (n vector).

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

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

n is the number of assets in the universe.

Portfolio object properties to specify bound constraints are:

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

The default is to ignore these constraints.

The default portfolio optimization problem (see "Default Portfolio Problem" on page 4-19) has l_B = `0` with u_B set implicitly through a budget constraint.

Budget Constraints

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

$$l_S \le 1^T x \le u_S$$

where:

x is the portfolio (n vector).

`1` is the vector of ones (n vector).

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

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

n is the number of assets in the universe.

Portfolio object properties to specify budget constraints are:

- `LowerBudget` for l_S
- `UpperBudget` for u_S
- `NumAssets` for n

The default is to ignore this constraint.

The default portfolio optimization problem (see "Default Portfolio Problem" on page 4-19) has $l_S = u_S = 1$, which means that the portfolio weights sum to `1`. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0–100% invested in cash. If cash is to be a portfolio choice, set `RiskFreeRate` (r_0) to a suitable value (see "Return Proxy" on page 4-4 and "Working with a Riskless Asset" on page 4-60).

Group Constraints

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

$$l_G \le Gx \le u_G$$

where:

x is the portfolio (n vector).

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

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

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

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

Portfolio object properties to specify group constraints are:

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

The default is to ignore these constraints.

Group Ratio Constraints

Group ratio constraints are specialized linear constraints that enforce relationships among groups of assets. The constraints take the form
$l_{Ri}(G_B x)_i \le (G_A x)_i \le u_{Ri}(G_B x)_i$

for $i = 1,..., n_R$ where:

x is the portfolio (n vector).

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

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

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

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

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

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

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

Portfolio object properties to specify group ratio constraints are:

- `GroupA` for G_A
- `GroupB` for G_B
- `LowerRatio` for l_R

- `UpperRatio` for u_R
- `NumAssets` for n

The default is to ignore these constraints.

Average Turnover Constraints

Turnover constraint is a linear absolute value constraint that ensures estimated optimal portfolios differ from an initial portfolio by no more than a specified amount. Although portfolio turnover is defined in many ways, the turnover constraints implemented in Financial Toolbox computes portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

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

where:

x is the portfolio (n vector).

`1` is the vector of ones (n vector).

x_0 is the initial portfolio (n vector).

τ is the upper bound for turnover (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

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

The default is to ignore this constraint.

One-way Turnover Constraints

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

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

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

where:

x is the portfolio (n vector)

`1` is the vector of ones (n vector).

x_0 is the Initial portfolio (n vector).

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

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

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

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

The default is to ignore this constraint.

Note The average turnover constraint (see "Working with Average Turnover Constraints Using Portfolio Object" on page 4-92) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

Tracking Error Constraints

Tracking error constraint, within a portfolio optimization framework, is an additional constraint to specify the set of feasible portfolios known as a portfolio set. The tracking-error constraint has the form

$$(x - x_T)^T C (x - x_T) \le {\tau_T}^2$$

where:

x is the portfolio (n vector).

x_T is the tracking portfolio against which risk is to be measured (n vector).

τ_T is the upper bound for tracking error (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

- `TrackingPort` for x_T
- `TrackingError` for τ_T

The default is to ignore this constraint.

Note The tracking error constraints can be used with any of the other supported constraints in the Portfolio object without restrictions. However, since the portfolio set necessarily and sufficiently must be a non-empty compact set, the application of a tracking error constraint may result in an empty portfolio set. Use `estimateBounds` to confirm that the portfolio set is non-empty and compact.

See Also

`Portfolio`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- `Portfolio`
- “Portfolio Object Workflow” on page 4-21
- “Default Portfolio Problem” on page 4-19

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Default Portfolio Problem

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

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

See Also

`Portfolio`

Related Examples

- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- `Portfolio`
- "Portfolio Set for Optimization Using Portfolio Object" on page 4-10
- "Portfolio Object Workflow" on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Object Workflow

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

1 Create a Portfolio.

   Create a `Portfolio` object for mean-variance portfolio optimization. For more information, see "Creating the Portfolio Object" on page 4-28.

2 Estimate the mean and covariance for returns.

   Evaluate the mean and covariance for portfolio asset returns, including assets with missing data and financial time series data. For more information, see "Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-48.

3 Specify the Portfolio Constraints.

   Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, turnover, and tracking error constraints. For more information, see "Working with Portfolio Constraints Using Defaults" on page 4-67.

4 Validate the Portfolio.

   Identify errors for the portfolio specification. For more information, see "Validate the Portfolio Problem for Portfolio Object" on page 4-104.

5 Estimate the efficient portfolios and frontiers.

   Analyze the efficient portfolios and efficient frontiers for a portfolio. For more information, see "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109 and "Estimate Efficient Frontiers for Portfolio Object" on page 4-129.

6 Postprocess the results.

   Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see "Postprocessing Results to Set Up Tradable Portfolios" on page 4-138.

For an example of this workflow, see "Asset Allocation Case Study" on page 4-175 and "Portfolio Optimization Examples" on page 4-147.

See Also

Related Examples

- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Optimization Theory” on page 4-3

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

Portfolio Object

In this section...

Portfolio Object Properties and Functions

The Portfolio object implements mean-variance portfolio optimization. Every property and function of the Portfolio object is public, although some properties and functions are hidden. See `Portfolio` for the properties and functions of the Portfolio object. The Portfolio object is a value object where every instance of the object is a distinct version of the object. Since the Portfolio object is also a MATLAB object, it inherits the default functions associated with MATLAB objects.

Working with Portfolio Objects

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

1. Design your portfolio problem.
2. Use the `Portfolio` function to create the Portfolio object or use the various set functions to set up your portfolio problem.
3. Use estimate functions to solve your portfolio problem.

In addition, functions are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a Portfolio object, you can save

and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of mean-variance portfolio optimization, means that you have either data or moments for asset returns and a collection of constraints on your portfolios, use the `Portfolio` function to set the properties for the Portfolio object. The `Portfolio` function lets you create an object from scratch or update an existing object. Since the Portfolio object is a value object, it is easy to create a basic object, then use functions to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see "Creating the Portfolio Object" on page 4-28.

Setting and Getting Properties

You can set properties of a Portfolio object using either the `Portfolio` function or various set functions.

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

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

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

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

- The `Portfolio` function and set functions try to determine the dimensions of your problem with either explicit or implicit inputs.
- The `Portfolio` function and set functions try to resolve ambiguities with default choices.

- The `Portfolio` function and set functions perform scalar expansion on arrays when possible.
- The associated Portfolio object functions try to diagnose and warn about problems.

Displaying Portfolio Objects

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

Saving and Loading Portfolio Objects

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

Estimating Efficient Portfolios and Frontiers

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

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

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

Arrays of Portfolio Objects

Although all functions associated with a Portfolio object are designed to work on a scalar Portfolio object, the array capabilities of MATLAB enables you to set up and work with

arrays of Portfolio objects. The easiest way to do this is with the `repmat` function. For example, to create a 3-by-2 array of Portfolio objects:

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

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

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

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

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

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

and, once set, you can access these bounds with

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

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

Subclassing Portfolio Objects

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

Conventions for Representation of Data

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

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

See Also

`Portfolio`

Related Examples

- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

Creating the Portfolio Object

In this section...

To create a fully specified mean-variance portfolio optimization problem, instantiate the Portfolio object using the `Portfolio` function. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

Syntax

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

```
p = Portfolio;
```

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

The `Portfolio` function also accepts collections of argument name-value pair arguments for properties and their values. The `Portfolio` function accepts inputs for public properties with the general syntax:

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

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

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

Input argument names are not case-sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 4-33). The `Portfolio` function tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process to

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

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

Portfolio Problem Sufficiency

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

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

  The first condition is satisfied by setting the properties associated with the moments of asset returns.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded.

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

Although the general sufficiency conditions for mean-variance portfolio optimization go beyond these two conditions, the Portfolio object implemented in Financial Toolbox implicitly handles all these additional conditions. For more information on the Markowitz model for mean-variance portfolio optimization, see “Portfolio Optimization” on page A-7.

Portfolio Function Examples

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

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

Portfolio with properties:
```

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

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

Using the Portfolio Function for a Single-Step Setup

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

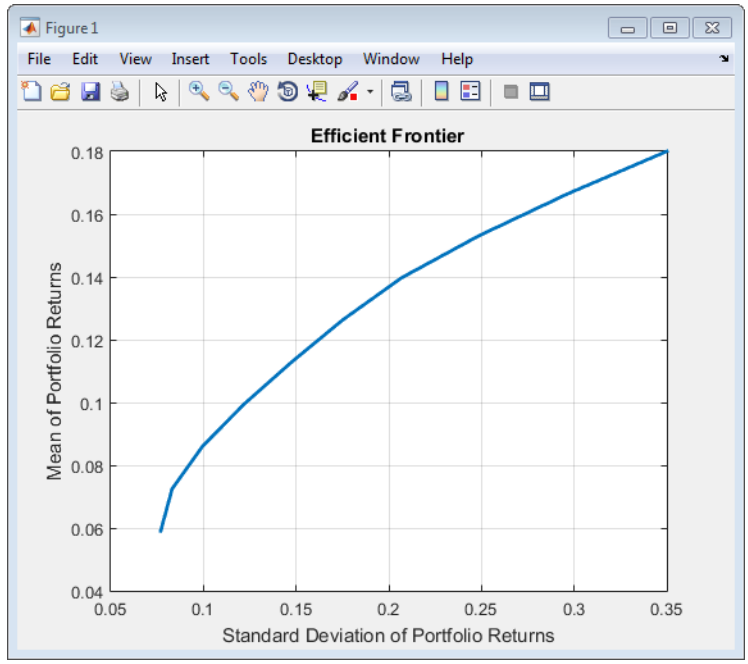
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
```

```
    0 0.0119 0.0336 0.1225 ];

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

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

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

```
p.plotFrontier;
```

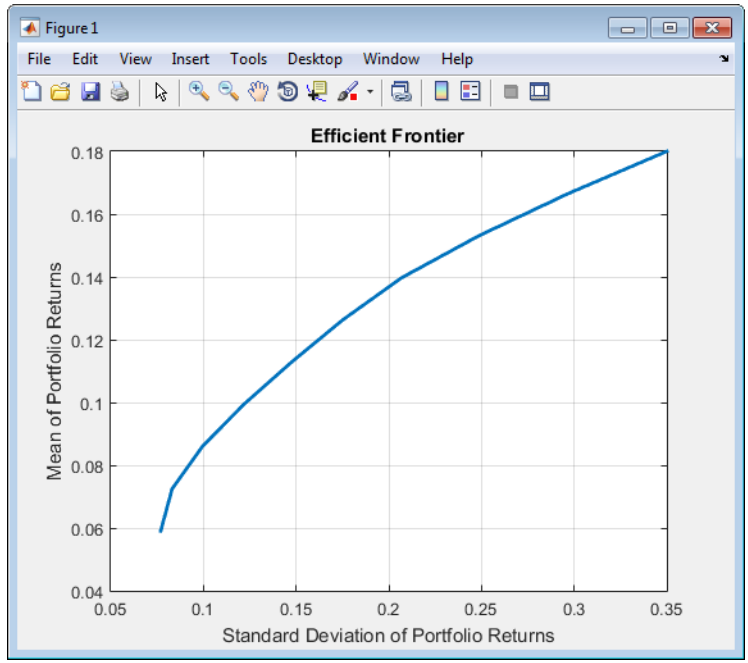


Using the Portfolio Function with a Sequence of Steps

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

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

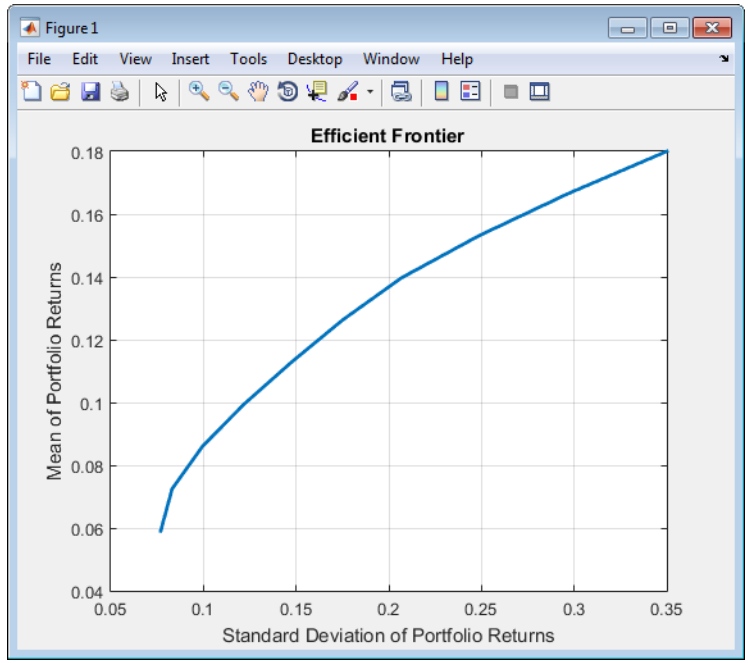
```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

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

plotFrontier(p);
```



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

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

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

plotFrontier(p);
```



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

Shortcuts for Property Names

The `Portfolio` function has shorter argument names that replace longer argument names associated with specific properties of the Portfolio object. For example, rather than enter `'assetcovar'`, the `Portfolio` function accepts the case-insensitive name `'covar'` to set the `AssetCovar` property in a Portfolio object. Every shorter argument name corresponds with a single property in the `Portfolio` function. The one exception is the alternative argument name `'budget'`, which signifies both the `LowerBudget` and

`UpperBudget` properties. When `'budget'` is used, then the `LowerBudget` and `UpperBudget` properties are set to the same value to form an equality budget constraint.

Shortcuts for Property Names

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

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

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

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

Direct Setting of Portfolio Object Properties

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

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

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

plotFrontier(p);
```

See Also

`Portfolio` | `estimateBounds`

Related Examples

- “Common Operations on the Portfolio Object” on page 4-36
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Common Operations on the Portfolio Object

In this section...

Naming a Portfolio Object

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

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

Asset Allocation Fund
```

Configuring the Assets in the Asset Universe

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

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

4
```

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

Setting Up a List of Asset Identifiers

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

Setting Up Asset Lists Using the Portfolio Function

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

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

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

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

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

```
5
```

Setting Up Asset Lists Using the setAssetList Function

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

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

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

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

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

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

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

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

```
p = Portfolio('numassets',5);
p = setAssetList(p);
disp(p.AssetList);
```

```
'Asset1'    'Asset2'    'Asset3'    'Asset4'    'Asset5'
```

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

```
p = Portfolio('numassets',5);
p.defaultforAssetList = 'ETF';
p = setAssetList(p);
disp(p.AssetList);
```

```
'ETF1'    'ETF2'    'ETF3'    'ETF4'    'ETF5'
```

Truncating and Padding Asset Lists

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

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

```
Warning: Input list of assets has 2 too few identifiers. Padding with numbered assets.
> In Portfolio.setAssetList at 121
    '921937835'    '922908769'    '922042775'    'UnknownCUSIP4'    'UnknownCUSIP5'
```

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

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

```
Warning: AssetList has 1 too many identifiers. Using first 4 assets.
> In Portfolio.checkarguments at 434
  In Portfolio.Portfolio>Portfolio.Portfolio at 171
    'AGG'    'EEM'    'MDY'    'SPY'
```

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

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

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

See Also

`Portfolio` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort` | `setTrackingPort`

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 4-41
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-48

- "Validate the Portfolio Problem for Portfolio Object" on page 4-104
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Setting Up an Initial or Current Portfolio

In many applications, creating a new optimal portfolio requires comparing the new portfolio with an initial or current portfolio to form lists of purchases and sales. The Portfolio object property `InitPort` lets you identify an initial or current portfolio. The initial portfolio also plays an essential role if you have either transaction costs or turnover constraints. The initial portfolio need not be feasible within the constraints of the problem. This can happen if the weights in a portfolio have shifted such that some constraints become violated. To check if your initial portfolio is feasible, use the `checkFeasibility` function described in "Validating Portfolios" on page 4-106. Suppose that you have an initial portfolio in `x0`, then use the `Portfolio` function to set up an initial portfolio:

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

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

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

To clear an initial portfolio from your Portfolio object, use either the `Portfolio` or the `setInitPort` function with an empty input for the `InitPort` property. If transaction costs or turnover constraints are set, it is not possible to clear the `InitPort` property in

this way. In this case, to clear `InitPort`, first clear the dependent properties and then clear the`InitPort` property.

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

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

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

```
 0.2500
 0.2500
 0.2500
 0.2500
```

Portfolio object functions that work with either transaction costs or turnover constraints also depend on the `InitPort` property. So, the set functions for transaction costs or turnover constraints permit the assignment of a value for the `InitPort` property as part of their implementation. For details, see “Working with Average Turnover Constraints Using Portfolio Object” on page 4-92, “Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-96, and “Working with Transaction Costs” on page 4-62 for details. If either transaction costs or turnover constraints are used, then the `InitPort` property must have a nonempty value. Absent a specific value assigned through the `Portfolio` function or various set functions, the Portfolio object sets `InitPort` to `0` and warns if `BuyCost`, `SellCost`, or `Turnover` properties are set. The following example illustrates what happens if an average turnover constraint is specified with an initial portfolio:

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

```
p = Portfolio('Turnover', 0.3);
disp(p.InitPort);
```

```
Warning: InitPort and NumAssets are empty and either transaction costs or turnover constraints specified.
Will set NumAssets = 1 and InitPort = 0.
> In Portfolio.checkarguments at 367
  In Portfolio.Portfolio>Portfolio.Portfolio at 171
     0
```

See Also

`Portfolio` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort`

Related Examples

- “Setting Up a Tracking Portfolio” on page 4-45
- “Common Operations on the Portfolio Object” on page 4-36
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-48
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

Setting Up a Tracking Portfolio

Given a benchmark or tracking portfolio, you can ensure that the risk of a portfolio relative to the benchmark portfolio is no greater than a specified amount. The Portfolio object property `TrackingPort` lets you identify a tracking portfolio. For more information on using a tracking portfolio with tracking error constraints, see “Working with Tracking Error Constraints Using Portfolio Object” on page 4-100.

The tracking error constraints can be used with any of the other supported constraints in the Portfolio object without restrictions. However, since the portfolio set necessarily and sufficiently must be a non-empty compact set, the application of a tracking error constraint can result in an empty portfolio set. Use `estimateBounds` to confirm that the portfolio set is non-empty and compact.

Suppose that you have an initial portfolio in `x0`, then use the `Portfolio` function to set up a tracking portfolio:

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

```
0.3000
    0.2000
    0.2000
         0
```

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

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

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

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

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

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

```
p = Portfolio;
p = setTrackingPort(p, 1/4, 4);
disp(p.TrackingPort);
```

```
 0.2500
 0.2500
 0.2500
 0.2500
```

See Also

`Portfolio` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort` | `setTrackingError` | `setTrackingPort`

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 4-41
- “Common Operations on the Portfolio Object” on page 4-36
- “Working with Portfolio Constraints Using Defaults” on page 4-67

- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-48
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

Asset Returns and Moments of Asset Returns Using Portfolio Object

In this section...

Since mean-variance portfolio optimization problems require estimates for the mean and covariance of asset returns, the Portfolio object has several ways to set and get the properties `AssetMean` (for the mean) and `AssetCovar` (for the covariance). In addition, the return for a riskless asset is kept in the property `RiskFreeRate` so that all assets in `AssetMean` and `AssetCovar` are risky assets. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

Assignment Using the Portfolio Function

Suppose that you have a mean and covariance of asset returns in variables `m` and `C`. The properties for the moments of asset returns are set using the `Portfolio` function:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
p = Portfolio('AssetMean', m, 'AssetCovar', C);
disp(p.NumAssets);
disp(p.AssetMean);
disp(p.AssetCovar);
```

```
4
    0.0042
    0.0083
```

```
    0.0100
    0.0150

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

Notice that the Portfolio object determines the number of assets in `NumAssets` from the moments. The `Portfolio` function enables separate initialization of the moments, for example:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = Portfolio(p, 'AssetMean', m);
p = Portfolio(p, 'AssetCovar', C);
[assetmean, assetcovar] = p.getAssetMoments

assetmean =

    0.0042
    0.0083
    0.0100
    0.0150


assetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

The `getAssetMoments` function lets you get the values for `AssetMean` and `AssetCovar` properties at the same time.

Assignment Using the setAssetMoments Function

You can also set asset moment properties using the `setAssetMoments` function. For example, given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = setAssetMoments(p, m, C);
[assetmean, assetcovar] = getAssetMoments(p)

assetmean =

    0.0042
    0.0083
    0.0100
    0.0150

assetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

Scalar Expansion of Arguments

Both the `Portfolio` function and the `setAssetMoments` function perform scalar expansion on arguments for the moments of asset returns. When using the `Portfolio` function, the number of assets must be already specified in the variable `NumAssets`. If `NumAssets` has not already been set, a scalar argument is interpreted as a scalar with `NumAssets` set to `1`. `setAssetMoments` provides an additional optional argument to specify the number of assets so that scalar expansion works with the correct number of assets. In addition, if either a scalar or vector is input for the covariance of asset returns, a diagonal matrix is formed such that a scalar expands along the diagonal and a vector

becomes the diagonal. This example demonstrates scalar expansion for four jointly independent assets with a common mean `0.1` and common variance `0.03`:

```
p = Portfolio;
p = setAssetMoments(p, 0.1, 0.03, 4);
[assetmean, assetcovar] = getAssetMoments(p)

assetmean =

    0.1000
    0.1000
    0.1000
    0.1000

assetcovar =

    0.0300         0         0         0
         0    0.0300         0         0
         0         0    0.0300         0
         0         0         0    0.0300
```

If at least one argument is properly dimensioned, you do not need to include the additional `NumAssets` argument. This example illustrates a constant-diagonal covariance matrix and a mean of asset returns for four assets:

```
p = Portfolio;
p = setAssetMoments(p, [ 0.05; 0.06; 0.04; 0.03 ], 0.03);
[assetmean, assetcovar] = getAssetMoments(p)

assetmean =

    0.0500
    0.0600
    0.0400
    0.0300

assetcovar =

    0.0300         0         0         0
         0    0.0300         0         0
         0         0    0.0300         0
         0         0         0    0.0300
```

In addition, scalar expansion works with the `Portfolio` function if `NumAssets` is known, or is deduced from the inputs.

Estimating Asset Moments from Prices or Returns

Another way to set the moments of asset returns is to use the `estimateAssetMoments` function which accepts either prices or returns and estimates the mean and covariance of asset returns. Either prices or returns are stored as matrices with samples going down the rows and assets going across the columns. In addition, prices or returns can be stored in a financial time series (`fints`) object (see "Estimating Asset Moments from Time Series Data" on page 4-57). To illustrate using `estimateAssetMoments`, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with `portsim`. The default behavior of `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by `portsim` in the variable `X`, a price series is created in the variable `Y`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
X = portsim(m', C, 120);
Y = ret2tick(X);
```

Note Portfolio optimization requires that you use total returns and not just price returns. So, "returns" should be total returns and "prices" should be total return prices.

Given asset returns and prices in variables `X` and `Y` from above, this sequence of examples demonstrates equivalent ways to estimate asset moments for the Portfolio object. A Portfolio object is created in `p` with the moments of asset returns set directly in the `Portfolio` function, and a second Portfolio object is created in `q` to obtain the mean and covariance of asset returns from asset return data in `X` using `estimateAssetMoments`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
```

```
C = C/12;

X = portsim(m', C, 120);
p = Portfolio('mean', m, 'covar', C);
q = Portfolio;
q = estimateAssetMoments(q, X);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =

    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
    0.0000    0.0010    0.0028    0.0102
```

Notice how either approach has the same moments. The default behavior of `estimateAssetMoments` is to work with asset returns. If, instead, you have asset prices in the variable `Y`, `estimateAssetMoments` accepts a name-value pair argument name `'DataFormat'` with a corresponding value set to `'prices'` to indicate that the input to the function is in the form of asset prices and not returns (the default value for the `'DataFormat'` argument is `'returns'`). This example compares direct assignment of

moments in the Portfolio object `p` with estimated moments from asset price data in `Y` in the Portfolio object `q`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
Y = ret2tick(X);

p = Portfolio('mean',m,'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Y, 'dataformat', 'prices');

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =
```

```
    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
    0.0000    0.0010    0.0028    0.0102
```

Estimating Asset Moments with Missing Data

Often when working with multiple assets, you have missing data indicated by `NaN` values in your return or price data. Although "Multivariate Normal Regression" on page 9-2 goes into detail about regression with missing data, the `estimateAssetMoments` function has a name-value pair argument name `'MissingData'` that indicates with a Boolean value whether to use the missing data capabilities of Financial Toolbox software. The default value for `'MissingData'` is `false` which removes all samples with `NaN` values. If, however, `'MissingData'` is set to `true`, `estimateAssetMoments` uses the ECM algorithm to estimate asset moments. This example illustrates how this works on price data with missing values:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
Y = ret2tick(X);
Y(1:20,1) = NaN;
Y(1:12,4) = NaN;

p = Portfolio('mean',m,'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Y, 'dataformat', 'prices');

r = Portfolio;
r = estimateAssetMoments(r, Y, 'dataformat', 'prices', 'missingdata', true);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)
[rassetmean, rassetcovar] = getAssetMoments(r)

passetmean =

    0.0042
    0.0083
    0.0100
```

```
    0.0150


passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102


qassetmean =

    0.0045
    0.0082
    0.0101
    0.0091


qassetcovar =

    0.0006    0.0003    0.0001   -0.0000
    0.0003    0.0023    0.0017    0.0011
    0.0001    0.0017    0.0048    0.0029
   -0.0000    0.0011    0.0029    0.0112


rassetmean =

    0.0045
    0.0083
    0.0100
    0.0113


rassetcovar =

    0.0008    0.0005    0.0001   -0.0001
    0.0005    0.0032    0.0022    0.0015
    0.0001    0.0022    0.0063    0.0040
   -0.0001    0.0015    0.0040    0.0144
```

The Portfolio object `p` contains raw moments, the object `q` contains estimated moments in which `NaN` values are discarded, and the object `r` contains raw moments that

accommodate missing values. Each time you run this example, you will get different estimates for the moments in `q` and `r`, and these will also differ from the moments in `p`.

Estimating Asset Moments from Time Series Data

The `estimateAssetMoments` function also accepts asset returns or prices stored in financial time series (`fints`) objects. `estimateAssetMoments` implicitly works with matrices of data or data in a `fints` object using the same rules for whether the data are returns or prices.

To illustrate, use `fints` to create a `fints` objects `Xfts` that contains asset returns generated with `portsim` (see "Estimating Asset Moments from Prices or Returns" on page 4-52) and add series labels:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);

d = (datenum('31-jan-2001'):datenum('31-dec-2010'))';
Xfts = fints(d, zeros(numel(d),4), {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'});
Xfts = tomonthly(Xfts);

Xfts.Bonds = X(:,1);
Xfts.LargeCap = X(:,2);
Xfts.SmallCap = X(:,3);
Xfts.Emerging = X(:,4);

p = Portfolio('mean',m,'covar',C);

q = Portfolio;
q = estimateAssetMoments(q, Xfts);

[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)
```

```
passetmean =

    0.0042
    0.0083
    0.0100
    0.0150

passetcovar =

    0.0005    0.0003    0.0002         0
```

```
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150

qassetcovar =

    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
    0.0000    0.0010    0.0028    0.0102
```

As you can see, the moments match. The argument name-value inputs `'DataFormat'` to handle return or price data and `'MissingData'` to ignore or use samples with missing values also work for `fints` data. In addition, `estimateAssetMoments` also extracts asset names or identifiers from a `fints` object with the argument name `'GetAssetList'` set to `true` (its default value is `false`). If the `'GetAssetList'` value is `true`, the identifiers are used to set the `AssetList` property of the object. Thus, repeating the formation of the Portfolio object `q` from the previous example with the `'GetAssetList'` flag set to `true` extracts the series labels from the `fints` object:

```
q = estimateAssetMoments(q, Xfts, 'getassetlist', true);
disp(q.AssetList)

'Bonds'    'LargeCap'    'SmallCap'    'Emerging'
```

Note if you set the `'GetAssetList'` flag set to `true` and your input data is in a matrix, `estimateAssetMoments` uses the default labeling scheme from `setAssetList` described in “Setting Up a List of Asset Identifiers” on page 4-37.

See Also

`Portfolio` | `estimateAssetMoments` | `getAssetMoments` | `setAssetMoments` | `setCosts`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with a Riskless Asset

You can specify a riskless asset with the mean and covariance of asset returns in the `AssetMean` and `AssetCovar` properties such that the riskless asset has variance of `0` and is completely uncorrelated with all other assets. In this case, the Portfolio object uses a separate `RiskFreeRate` property that stores the rate of return of a riskless asset. Thus, you can separate your universe into a riskless asset and a collection of risky assets. For example, assume that your riskless asset has a return in the scalar variable `r0`, then the property for the `RiskFreeRate` is set using the `Portfolio` function:

```
r0 = 0.01/12;
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio('RiskFreeRate', r0, 'AssetMean', m, 'AssetCovar', C);
disp(p.RiskFreeRate);
```

```
 8.3333e-004
```

Note If your problem has a budget constraint such that your portfolio weights must sum to `1`, then the riskless asset is irrelevant.

See Also

`Portfolio` | `estimateAssetMoments` | `getAssetMoments` | `setAssetMoments`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

Working with Transaction Costs

The difference between net and gross portfolio returns is transaction costs. The net portfolio return proxy has distinct proportional costs to purchase and to sell assets which are maintained in the Portfolio object properties `BuyCost` and `SellCost`. Transaction costs are in units of total return and, as such, are proportional to the price of an asset so that they enter the model for net portfolio returns in return form. For example, suppose that you have a stock currently priced $40 and your usual transaction costs are five cents per share. Then the transaction cost for the stock is 0.05/40 = 0.00125 (as defined in "Net Portfolio Returns" on page 4-5). Costs are entered as positive values and credits are entered as negative values.

Setting Transaction Costs Using the Portfolio Function

To set up transaction costs, you must specify an initial or current portfolio in the `InitPort` property. If the initial portfolio is not set when you set up the transaction cost properties, `InitPort` is `0`. The properties for transaction costs can be set using the`Portfolio` function. For example, assume that purchase and sale transaction costs are in the variables `bc` and `sc` and an initial portfolio is in the variable `x0`, then transaction costs are set:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio('BuyCost', bc, 'SellCost', sc, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
     5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
```

```
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Setting Transaction Costs Using the setCosts Function

You can also set the properties for transaction costs using `setCosts`. Assume that you have the same costs and initial portfolio as in the previous example. Given a Portfolio object `p` with an initial portfolio already set, use `setCosts` to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = Portfolio('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
    5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
```

```
    0.1000
    0.1000
```

You can also set up the initial portfolio's `InitPort` value as an optional argument to `setCosts` so that the following is an equivalent way to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = Portfolio;
p = setCosts(p, bc, sc, x0);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
    5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

For an example of setting costs, see “Portfolio Analysis with Turnover Constraints”.

Setting Transaction Costs with Scalar Expansion

Both the `Portfolio` function and the `setCosts` function implement scalar expansion on the arguments for transaction costs and the initial portfolio. If the `NumAssets` property

is already set in the Portfolio object, scalar arguments for these properties are expanded to have the same value across all dimensions. In addition, `setCosts` lets you specify `NumAssets` as an optional final argument. For example, assume that you have an initial portfolio `x0` and you want to set common transaction costs on all assets in your universe. You can set these costs in any of these equivalent ways:

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio('InitPort', x0, 'BuyCost', 0.002, 'SellCost', 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio('InitPort', x0);
p = setCosts(p, 0.002, 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = Portfolio;
p = setCosts(p, 0.002, 0.002, x0);
```

To clear costs from your Portfolio object, use either the `Portfolio` function or `setCosts` with empty inputs for the properties to be cleared. For example, you can clear sales costs from the Portfolio object `p` in the previous example:

```
p = Portfolio(p, 'SellCost', []);
```

See Also

`Portfolio` | `estimateAssetMoments` | `getAssetMoments` | `setAssetMoments` | `setCosts`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129

- “Portfolio Analysis with Turnover Constraints”
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Portfolio Constraints Using Defaults

The final element for a complete specification of a portfolio optimization problem is the

set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset R^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or "default" portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to `1` (using the budget constraint). For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

Setting Default Constraints for Portfolio Weights Using Portfolio Object

The "default" portfolio problem has two constraints on portfolio weights:

- Portfolio weights must be nonnegative.
- Portfolio weights must sum to `1`.

Implicitly, these constraints imply that portfolio weights are no greater than `1`, although this is a superfluous constraint to impose on the problem.

Setting Default Constraints Using the Portfolio Function

Given a portfolio optimization problem with `NumAssets` = `20` assets, use the `Portfolio` function to set up a default problem and explicitly set bounds and budget constraints:

```
p = Portfolio('NumAssets', 20, 'LowerBound', 0, 'Budget', 1);
disp(p);
```

```
 Portfolio with properties:

        BuyCost: []
       SellCost: []
   RiskFreeRate: []
      AssetMean: []
     AssetCovar: []
  TrackingError: []
   TrackingPort: []
```

```
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
            Name: []
       NumAssets: 20
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [20x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Setting Default Constraints Using the setDefaultConstraints Function

An alternative approach is to use the `setDefaultConstraints` function. If the number of assets is already known in a Portfolio object, use `setDefaultConstraints` with no arguments to set up the necessary bound and budget constraints. Suppose that you have 20 assets to set up the portfolio set for a default problem:

```
p = Portfolio('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);
```

```
 Portfolio with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
       AssetMean: []
      AssetCovar: []
   TrackingError: []
    TrackingPort: []
        Turnover: []
```

```
     BuyTurnover: []
    SellTurnover: []
            Name: []
       NumAssets: 20
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [20x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

If the number of assets is unknown, `setDefaultConstraints` accepts `NumAssets` as an optional argument to form a portfolio set for a default problem. Suppose that you have 20 assets:

```
p = Portfolio;
p = setDefaultConstraints(p, 20);
disp(p);
```

```
Portfolio with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
       AssetMean: []
      AssetCovar: []
   TrackingError: []
    TrackingPort: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
            Name: []
       NumAssets: 20
       AssetList: []
```

```
   InitPort: []
AInequality: []
bInequality: []
  AEquality: []
  bEquality: []
 LowerBound: [20x1 double]
 UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
 LowerGroup: []
 UpperGroup: []
     GroupA: []
     GroupB: []
 LowerRatio: []
 UpperRatio: []
```

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Working with Bound Constraints Using Portfolio Object” on page 4-72
- “Working with Budget Constraints Using Portfolio Object” on page 4-75
- “Working with Group Constraints Using Portfolio Object” on page 4-78
- “Working with Group Ratio Constraints Using Portfolio Object” on page 4-82
- “Working with Linear Equality Constraints Using Portfolio Object” on page 4-86
- “Working with Linear Inequality Constraints Using Portfolio Object” on page 4-89
- “Working with Average Turnover Constraints Using Portfolio Object” on page 4-92
- “Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-96
- “Working with Tracking Error Constraints Using Portfolio Object” on page 4-100
- “Creating the Portfolio Object” on page 4-28
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Bound Constraints Using Portfolio Object

Bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see "Bound Constraints" on page 4-11). Although every portfolio set must be bounded, it is not necessary to specify a portfolio set with explicit bound constraints. For example, you can create a portfolio set with an implicit upper bound constraint or a portfolio set with average turnover constraints. The bound constraints have properties `LowerBound` for the lower-bound constraint and `UpperBound` for the upper-bound constraint. Set default values for these constraints using the `setDefaultConstraints` function (see "Setting Default Constraints for Portfolio Weights Using Portfolio Object" on page 4-67).

Setting Bounds Using the Portfolio Function

The properties for bound constraints are set through the `Portfolio` function. Suppose that you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. The bound constraints for a balanced fund are set with:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = Portfolio('LowerBound', lb, 'UpperBound', ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
 2

 0.5000
 0.2500

 0.7500
 0.5000
```

To continue with this example, you must set up a budget constraint. For details, see "Working with Budget Constraints Using Portfolio Object" on page 4-75.

Setting Bounds Using the setBounds Function

You can also set the properties for bound constraints using `setBounds`. Suppose that you have a balanced fund with stocks that can range from 50% to 75% of your portfolio

and bonds that can range from 25% to 50% of your portfolio. Given a Portfolio object `p`, use `setBounds` to set the bound constraints:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = Portfolio;
p = setBounds(p, lb, ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
  2

  0.5000
  0.2500

  0.7500
  0.5000
```

Setting Bounds Using the Portfolio Function or setBounds Function

Both the `Portfolio` function and `setBounds` function implement scalar expansion on either the `LowerBound` or `UpperBound` properties. If the `NumAssets` property is already set in the Portfolio object, scalar arguments for either property expand to have the same value across all dimensions. In addition, `setBounds` lets you specify `NumAssets` as an optional argument. Suppose that you have a universe of 500 assets and you want to set common bound constraints on all assets in your universe. Specifically, you are a long-only investor and want to hold no more than 5% of your portfolio in any single asset. You can set these bound constraints in any of these equivalent ways:

```
p = Portfolio('NumAssets', 500, 'LowerBound', 0, 'UpperBound', 0.05);
```

or

```
p = Portfolio('NumAssets', 500);
p = setBounds(p, 0, 0.05);
```

or

```
p = Portfolio;
p = setBounds(p, 0, 0.05, 500);
```

To clear bound constraints from your Portfolio object, use either the `Portfolio` function or `setBounds` with empty inputs for the properties to be cleared. For example, to clear the upper-bound constraint from the Portfolio object `p` in the previous example:

```
p = Portfolio(p, 'UpperBound', []);
```

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Budget Constraints Using Portfolio Object

The budget constraint is an optional linear constraint that maintains upper and lower bounds on the sum of portfolio weights (see "Budget Constraints" on page 4-12). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to `1` using `setDefaultConstraints` (see "Setting Default Constraints for Portfolio Weights Using Portfolio Object" on page 4-67).

Setting Budget Constraints Using the Portfolio Function

The properties for the budget constraint can also be set using the `Portfolio` function. Suppose that you have an asset universe with many risky assets and a riskless asset and you want to ensure that your portfolio never holds more than 1% cash, that is, you want to ensure that you are 99–100% invested in risky assets. The budget constraint for this portfolio can be set with:

```
p = Portfolio('LowerBudget', 0.99, 'UpperBudget', 1);
disp(p.LowerBudget);
disp(p.UpperBudget);
```

```
 0.9900

1
```

Setting Budget Constraints Using the setBudget Function

You can also set the properties for a budget constraint using `setBudget`. Suppose that you have a fund that permits up to 10% leverage which means that your portfolio can be from 100% to 110% invested in risky assets. Given a Portfolio object `p`, use `setBudget` to set the budget constraints:

```
p = Portfolio;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);
disp(p.UpperBudget);
```

```
1

 1.1000
```

If you were to continue with this example, then set the `RiskFreeRate` property to the borrowing rate to finance possible leveraged positions. For details on the `RiskFreeRate` property, see “Working with a Riskless Asset” on page 4-60. To clear either bound for the budget constraint from your Portfolio object, use either the `Portfolio` function or `setBudget` with empty inputs for the properties to be cleared. For example, clear the upper-budget constraint from the Portfolio object `p` in the previous example with:

```
p = Portfolio(p, 'UpperBudget', []);
```

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Group Constraints Using Portfolio Object

Group constraints are optional linear constraints that group assets together and enforce bounds on the group weights (see “Group Constraints” on page 4-13). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the Portfolio Function

The properties for group constraints are set through the `Portfolio` function. Suppose that you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio, then you can set group constraints:

```
G = [ 1 1 1 0 0 ];
p = Portfolio('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

The group matrix `G` can also be a logical matrix so that the following code achieves the same result.

```
G = [ true true true false false ];
p = Portfolio('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

Setting Group Constraints Using the setGroups and addGroups Functions

You can also set the properties for group constraints using `setGroups`. Suppose that you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a Portfolio object `p`, use `setGroups` to set the group constraints:

```
G = [ true true true false false ];
p = Portfolio;
p = setGroups(p, G, [], 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

In this example, you would set the `LowerGroup` property to be empty (`[]`).

Suppose that you want to add another group constraint to make odd-numbered assets constitute at least 20% of your portfolio. Set up an augmented group matrix and introduce infinite bounds for unconstrained group bounds or use the `addGroups` function to build up group constraints. For this example, create another group matrix for the second group constraint:

```
p = Portfolio;
G = [ true true true false false ];    % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ];    % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
 5

1     1     1     0     0
1     0     1     0     1

  -Inf
-Inf
0.2000
0.2000
```

```
0.3000
  Inf
```

`addGroups` determines which bounds are unbounded so you only need to focus on the constraints that you want to set.

The `Portfolio` function and `setGroups` and `addGroups` implement scalar expansion on either the `LowerGroup` or `UpperGroup` properties based on the dimension of the group matrix in the property `GroupMatrix`. Suppose that you have a universe of 30 assets with six asset classes such that assets 1–5, assets 6–12, assets 13–18, assets 19–22, assets 23–27, and assets 28–30 constitute each of your six asset classes and you want each asset class to fall from 0% to 25% of your portfolio. Let the following group matrix define your groups and scalar expansion define the common bounds on each group:

```
p = Portfolio;
G = blkdiag(true(1,5), true(1,7), true(1,6), true(1,4), true(1,5), true(1,3));
p = setGroups(p, G, 0, 0.25);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
30

  Columns 1 through 16

     1     1     1     1     1     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     1     1     1     1     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     1     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0

  Columns 17 through 30

     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     1     1     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     1     1     1     1     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     1     1     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     1     1     1

     0
     0
     0
     0
     0
     0

    0.2500
    0.2500
    0.2500
    0.2500
    0.2500
    0.2500
```

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Validate the Portfolio Problem for Portfolio Object" on page 4-104
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Constraint Specification Using a Portfolio Object" on page 3-34
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21
- "Setting Up a Tracking Portfolio" on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Group Ratio Constraints Using Portfolio Object

Group ratio constraints are optional linear constraints that maintain bounds on proportional relationships among groups of assets (see “Group Ratio Constraints” on page 4-14). Although the constraints are implemented as general constraints, the usual convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group compared to a comparison group fall within specified bounds. Group ratio constraints have properties:

- `GroupA` for the base membership matrix
- `GroupB` for the comparison membership matrix
- `LowerRatio` for the lower-bound constraint on the ratio of groups
- `UpperRatio` for the upper-bound constraint on the ratio of groups

Setting Group Ratio Constraints Using the Portfolio Function

The properties for group ratio constraints are set using the `Portfolio` function. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose that you have six assets with three financial companies (assets 1–3) and three nonfinanical companies (assets 4–6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ];    % financial companies
GB = [ 0 0 0 1 1 1 ];    % nonfinancial companies
p = Portfolio('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = Portfolio('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Setting Group Ratio Constraints Using the setGroupRatio and addGroupRatio Functions

You can also set the properties for group ratio constraints using `setGroupRatio`. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose that you have six assets with three financial companies (assets 1–3) and three nonfinanical companies (assets 4–6). Given a Portfolio object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose that you want to add another group ratio constraint to ensure that the weights in odd-numbered assets constitute at least 20% of the weights in nonfinancial assets your

portfolio. You can set up augmented group ratio matrices and introduce infinite bounds for unconstrained group ratio bounds, or you can use the `addGroupRatio` function to build up group ratio constraints. For this example, create another group matrix for the second group constraint:

```
p = Portfolio;
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ];   % odd-numbered companies
GB = [ false false false true true true ];   % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);
```

```
 6

1     1     1     0     0     0
1     0     1     0     1     0

0     0     0     1     1     1
0     0     0     1     1     1

  -Inf
0.2000

0.5000
  Inf
```

Notice that `addGroupRatio` determines which bounds are unbounded so you only need to focus on the constraints you want to set.

The `Portfolio` function, `setGroupRatio`, and `addGroupRatio` implement scalar expansion on either the `LowerRatio` or `UpperRatio` properties based on the dimension of the group matrices in `GroupA` and `GroupB` properties.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Linear Equality Constraints Using Portfolio Object

Linear equality constraints are optional linear constraints that impose systems of equalities on portfolio weights (see “Linear Equality Constraints” on page 4-11). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `bEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the Portfolio Function

The properties for linear equality constraints are set using the `Portfolio` function. Suppose that you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. To set this constraint:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio('AEquality', A, 'bEquality', b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);
```

```
5

1     1     1     0     0

0.5000
```

Setting Linear Equality Constraints Using the setEquality and addEquality Functions

You can also set the properties for linear equality constraints using `setEquality`. Suppose that you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. Given a Portfolio object `p`, use `setEquality` to set the linear equality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio;
p = setEquality(p, A, b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);
```

```
5

1     1     1     0     0

0.5000
```

Suppose that you want to add another linear equality constraint to ensure that the last three assets also constitute 50% of your portfolio. You can set up an augmented system of linear equalities or use `addEquality` to build up linear equality constraints. For this example, create another system of equalities:

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % first equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % second equality constraint
b = 0.5;
p = addEquality(p, A, b);

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

```
5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000
```

The `Portfolio` function, `setEquality`, and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Linear Inequality Constraints Using Portfolio Object

Linear inequality constraints are optional linear constraints that impose systems of inequalities on portfolio weights (see “Linear Inequality Constraints” on page 4-10). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the Portfolio Function

The properties for linear inequality constraints are set using the `Portfolio` function. Suppose that you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. To set up these constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio('AInequality', A, 'bInequality', b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5

1     1     1     0     0

0.5000
```

Setting Linear Inequality Constraints Using the setInequality and addInequality Functions

You can also set the properties for linear inequality constraints using `setInequality`. Suppose that you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 50% of your portfolio. Given a Portfolio object `p`, use `setInequality` to set the linear inequality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio;
p = setInequality(p, A, b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5

1     1     1     0     0

0.5000
```

Suppose that you want to add another linear inequality constraint to ensure that the last three assets constitute at least 50% of your portfolio. You can set up an augmented system of linear inequalities or use the `addInequality` function to build up linear inequality constraints. For this example, create another system of inequalities:

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

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

```
5

1     1     1     0     0
0     0    -1    -1    -1

0.5000
-0.5000
```

The `Portfolio` function, `setInequality`, and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Average Turnover Constraints Using Portfolio Object

The turnover constraint is an optional linear absolute value constraint (see "Average Turnover Constraints" on page 4-15) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `Portfolio` function or the `setTurnover` function. The turnover constraint depends on an initial or current portfolio, which is assumed to be zero if not set when the turnover constraint is set. The turnover constraint has properties `Turnover`, for the upper bound on average turnover, and `InitPort`, for the portfolio against which turnover is computed.

Setting Average Turnover Constraints Using the Portfolio Function

The properties for the turnover constraints are set using the `Portfolio` function. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and you want to ensure that average turnover is no more than 30%. To set this turnover constraint:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('Turnover', 0.3, 'InitPort', x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

Note if the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see "Setting Up an Initial or Current Portfolio" on page 4-41).

Setting Average Turnover Constraints Using the setTurnover Function

You can also set properties for portfolio turnover using `setTurnover` to specify both the upper bound for average turnover and an initial portfolio. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that average turnover is no more than 30%. Given a Portfolio object `p`, use `setTurnover` to set the turnover constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('InitPort', x0);
p = setTurnover(p, 0.3);

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

```
10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio;
p = setTurnover(p, 0.3, x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10

0.3000

0.1200
0.0900
0.0800
```

```
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

For an example of setting turnover, see “Portfolio Analysis with Turnover Constraints”.

`setTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the Portfolio object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setTurnover` lets you specify `NumAssets` as an optional argument. To clear turnover from your Portfolio object, use the `Portfolio` function or `setTurnover` with empty inputs for the properties to be cleared.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Portfolio Analysis with Turnover Constraints”
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147
- “Portfolio Analysis with Turnover Constraints”

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21
- "Setting Up a Tracking Portfolio" on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with One-Way Turnover Constraints Using Portfolio Object

One-way turnover constraints are optional constraints (see “One-way Turnover Constraints” on page 4-15) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the `Portfolio` function or the `setOneWayTurnover` function. One-way turnover constraints depend upon an initial or current portfolio, which is assumed to be zero if not set when the turnover constraints are set. One-way turnover constraints have properties `BuyTurnover`, for the upper bound on net purchases, `SellTurnover`, for the upper bound on net sales, and `InitPort`, for the portfolio against which turnover is computed.

Setting One-Way Turnover Constraints Using the Portfolio Function

The Properties for the one-way turnover constraints are set using the `Portfolio` function. Suppose that you have an initial portfolio with 10 assets in a variable `x0` and you want to ensure that turnover on purchases is no more than 30% and turnover on sales is no more than 20% of the initial portfolio. To set these turnover constraints:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('BuyTurnover', 0.3, 'SellTurnover', 0.2, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
    10

    0.3000

    0.2000

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

If the NumAssets or InitPort properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see "Setting Up an Initial or Current Portfolio" on page 4-41).

Setting Turnover Constraints Using the setOneWayTurnover Function

You can also set properties for portfolio turnover using setOneWayTurnover to specify the upper bounds for turnover on purchases (BuyTurnover) and sales (SellTurnover) and an initial portfolio. Suppose that you have an initial portfolio of 10 assets in a variable x0 and want to ensure that turnover on purchases is no more than 30% and that turnover on sales is no more than 20% of the initial portfolio. Given a Portfolio object p, use setOneWayTurnover to set the turnover constraints with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);

disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
    10

    0.3000

    0.2000

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio;
p = setOneWayTurnover(p, 0.3, 0.2, x0);
```

```
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
    10

    0.3000

    0.2000

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

`setOneWayTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the Portfolio object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setOneWayTurnover` lets you specify `NumAssets` as an optional argument. To remove one-way turnover from your Portfolio object, use the `Portfolio` function or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109

- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with Tracking Error Constraints Using Portfolio Object

Tracking error constraints are optional constraints (see “Tracking Error Constraints” on page 4-16) that measure the risk relative to a portfolio called a tracking portfolio. Tracking error constraints can be set using the `Portfolio` function or the `setTrackingError` function.

The tracking error constraint is an optional quadratic constraint that enforces an upper bound on tracking error, which is the relative risk between a portfolio and a designated tracking portfolio. For more information, see “Tracking Error Constraints” on page 4-16.

The tracking error constraint can be set using the `Portfolio` function or the `setTrackingPort` and `setTrackingError` functions. The tracking error constraint depends on a tracking portfolio, which is assumed to be zero if not set when the tracking error constraint is set. The tracking error constraint has properties `TrackingError`, for the upper bound on tracking error, and `TrackingPort`, for the portfolio against which tracking error is computed.

Note The initial portfolio in the Portfolio object property `InitPort` is distinct from the tracking portfolio in the Portfolio object property `TrackingPort`.

Setting Tracking Error Constraints Using the Portfolio Function

The properties for the tracking error constraints are set using the `Portfolio` function. Suppose that you have a tracking portfolio of 10 assets in a variable *x0* and you want to ensure that the tracking error of any portfolio on the efficient frontier is no more than 8% relative to this portfolio. To set this constraint:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('TrackingError', 0.08, 'TrackingPort', x0);
disp(p.NumAssets);
disp(p.TrackingError);
disp(p.TrackingPort);
```

```
 10

    0.0800

    0.1200
    0.0900
    0.0800
```

```
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

If the `NumAssets` or `TrackingPort` properties are not set before or when the tracking error constraint is set, various rules are applied to assign default values to these properties (see "Setting Up a Tracking Portfolio" on page 4-45).

Setting Tracking Error Constraints Using the setTrackingError Function

You can also set properties for portfolio tracking error using the `setTrackingError` function to specify both the upper bound for tracking error and a designated tracking portfolio. Suppose that you have a tracking portfolio of 10 assets in a variable x0 and want to ensure that tracking error is no more than 8%. Given a Portfolio object `p`, use `setTrackingError` to set the tracking error constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('TrackingPort', x0);
p = setTrackingError(p, 0.08);

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

```
    10

    0.0800

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('TrackingPort', x0);
p = setTrackingError(p, 0.08, x0);

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

```
10

    0.0800

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

Note that if the `NumAssets` or `TrackingPort` properties are not set before or when the tracking error constraint is set, various rules are applied to assign default values to these properties (see “Setting Up a Tracking Portfolio” on page 4-45).

`setTrackingError` implements scalar expansion on the argument for the tracking portfolio. If the `NumAssets` property is already set in the Portfolio object, a scalar argument for `TrackingPort` expands to have the same value across all dimensions. In addition, `setTrackingError` lets you specify `NumAssets` as an optional argument. To clear tracking error from your Portfolio object, use the `Portfolio` function or `setTrackingError` with empty inputs for the properties to be cleared.

See Also

`Portfolio` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTrackingError` | `setTrackingPort` | `setTurnover`

Related Examples

- “Creating the Portfolio Object” on page 4-28

- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Constraint Specification Using a Portfolio Object” on page 3-34
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21
- “Setting Up a Tracking Portfolio” on page 4-45

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Validate the Portfolio Problem for Portfolio Object

In this section...

Sometimes, you may want to validate either your inputs to, or outputs from, a portfolio optimization problem. Although most error checking that occurs during the problem setup phase catches most difficulties with a portfolio optimization problem, the processes to validate portfolio sets and portfolios are time consuming and are best done offline. So, the portfolio optimization tools have specialized functions to validate portfolio sets and portfolios. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

Validating a Portfolio Set

Since it is necessary and sufficient that your portfolio set must be a nonempty, closed, and bounded set to have a valid portfolio optimization problem, the `estimateBounds` function lets you examine your portfolio set to determine if it is nonempty and, if nonempty, whether it is bounded. Suppose that you have the following portfolio set which is an empty set because the initial portfolio at `0` is too far from a portfolio that satisfies the budget and turnover constraint:

```
p = Portfolio('NumAssets', 3, 'Budget', 1);
p = setTurnover(p, 0.3, 0);
```

If a portfolio set is empty, `estimateBounds` returns `NaN` bounds and sets the `isbounded` flag to `[]`:

```
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =

   NaN
   NaN
   NaN

ub =

   NaN
```

```
   NaN
   NaN

isbounded =

     []
```

Suppose that you create an unbounded portfolio set as follows:

```
p = Portfolio('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

  -Inf
  -Inf

ub =

  1.0e-008 *

   -0.3712
       Inf

isbounded =

     0
```

In this case, `estimateBounds` returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Finally, suppose that you created a portfolio set that is both nonempty and bounded. `estimateBounds` not only validates the set, but also obtains tighter bounds which are useful if you are concerned with the actual range of portfolio choices for individual assets in your portfolio set:

```
p = Portfolio;
p = setBudget(p, 1,1);
p = setBounds(p, [ -0.1; 0.2; 0.3; 0.2 ], [ 0.5; 0.3; 0.9; 0.8 ]);

[lb, ub, isbounded] = estimateBounds(p)

lb =

   -0.1000
```

```
    0.2000
    0.3000
    0.2000

ub =

    0.3000
    0.3000
    0.7000
    0.6000

isbounded =

     1
```

In this example, all but the second asset has tighter upper bounds than the input upper bound implies.

Validating Portfolios

Given a portfolio set specified in a Portfolio object, you often want to check if specific portfolios are feasible with respect to the portfolio set. This can occur with, for example, initial portfolios and with portfolios obtained from other procedures. The `checkFeasibility` function determines whether a collection of portfolios is feasible. Suppose that you perform the following portfolio optimization and want to determine if the resultant efficient portfolios are feasible relative to a modified problem.

First, set up a problem in the Portfolio object `p`, estimate efficient portfolios in `pwgt`, and then confirm that these portfolios are feasible relative to the initial problem:

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

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)
```

```
ans =

     1     1     1     1     1     1     1     1     1     1
```

Next, set up a different portfolio problem that starts with the initial problem with an additional a turnover constraint and an equally weighted initial portfolio:

```
q = setTurnover(p, 0.3, 0.25);
checkFeasibility(q, pwgt)

ans =

     0     0     0     1     1     0     0     0     0     0
```

In this case, only two of the 10 efficient portfolios from the initial problem are feasible relative to the new problem in Portfolio object `q`. Solving the second problem using `checkFeasibility` demonstrates that the efficient portfolio for Portfolio object `q` is feasible relative to the initial problem:

```
qwgt = estimateFrontier(q);
checkFeasibility(p, qwgt)

ans =

     1     1     1     1     1     1     1     1     1     1
```

See Also

`Portfolio` | `checkFeasibility` | `estimateBounds`

Related Examples

- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object

There are two ways to look at a portfolio optimization problem that depends on what you are trying to do. One goal is to estimate efficient portfolios and the other is to estimate efficient frontiers. This section focuses on the former goal and "Estimate Efficient Frontiers for Portfolio Object" on page 4-129 focuses on the latter goal. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

Obtaining Portfolios Along the Entire Efficient Frontier

The most basic way to obtain optimal portfolios is to obtain points over the entire range of the efficient frontier. Given a portfolio optimization problem in a Portfolio object, the `estimateFrontier` function computes efficient portfolios spaced evenly according to the return proxy from the minimum to maximum return efficient portfolios. The number of portfolios estimated is controlled by the hidden property `defaultNumPorts` which is set to `10`. A different value for the number of portfolios estimated is specified as input to `estimateFrontier`. This example shows the default number of efficient portfolios over the entire range of the efficient frontier:

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

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);

disp(pwgt);
```

```
0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0         0         0         0
0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049    0.2314    0.0579         0
0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320    0.1394    0.1468         0
0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630    0.6292    0.7953    1.0000
```

If you want only four portfolios in the previous example:

```
pwgt = estimateFrontier(p, 4);
disp(pwgt);
```

```
    0.8891    0.3865         0         0
```

```
    0.0369    0.3129    0.4049         0
    0.0404    0.0893    0.1320         0
    0.0336    0.2113    0.4630    1.0000
```

Starting from the initial portfolio, `estimateFrontier` also returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0         0         0         0
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049    0.2314    0.0579         0
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320    0.1394    0.1468         0
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630    0.6292    0.7953    1.0000


pbuy =

    0.5891    0.4215    0.2540    0.0865         0         0         0         0         0         0
         0         0         0    0.0129    0.1049    0.1969    0.1049         0         0         0
         0         0         0         0         0         0         0         0         0         0
         0         0    0.0521    0.1113    0.1705    0.2297    0.3630    0.5292    0.6953    0.9000


psell =

    0         0         0         0    0.0810    0.2485    0.3000    0.3000    0.3000    0.3000
    0.2631    0.1711    0.0791         0         0         0         0    0.0686    0.2421    0.3000
    0.1596    0.1433    0.1270    0.1107    0.0944    0.0781    0.0680    0.0606    0.0532    0.2000
    0.0664    0.0071         0         0         0         0         0         0         0         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Obtaining Endpoints of the Efficient Frontier" on page 4-112
- "Obtaining Efficient Portfolios for Target Returns" on page 4-115
- "Obtaining Efficient Portfolios for Target Risks" on page 4-119
- "Efficient Portfolio That Maximizes Sharpe Ratio" on page 4-123
- "Plotting the Efficient Frontier for a Portfolio Object" on page 4-132
- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Obtaining Endpoints of the Efficient Frontier

Often, you might be interested in the endpoint portfolios for the efficient frontier. Suppose that you want to determine the range of returns from minimum to maximum to refine a search for a portfolio with a specific target return. Use the `estimateFrontierLimits` function to obtain the endpoint portfolios:

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

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);

disp(pwgt);
```

```
    0.8891         0
    0.0369         0
    0.0404         0
    0.0336    1.0000
```

The `estimatePortMoments` function shows the range of risks and returns for efficient portfolios:

```
[prsk, pret] = estimatePortMoments(p, pwgt);
disp([prsk, pret]);
```

```
 0.0769    0.0590
 0.3500    0.1800
```

Starting from an initial portfolio, `estimateFrontierLimits` also returns purchases and sales to get from the initial portfolio to the endpoint portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

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

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierLimits(p);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

    0.8891         0
    0.0369         0
    0.0404         0
    0.0336    1.0000

pbuy =

    0.5891         0
         0         0
         0         0
         0    0.9000

psell =

         0    0.3000
    0.2631    0.3000
    0.1596    0.2000
    0.0664         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Obtaining Efficient Portfolios for Target Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` function accepts one or more target portfolios returns and obtains efficient portfolios with the specified returns. For example, assume that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 6%, 9%, and 12%:

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

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);

display(pwgt);

pwgt =

    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022
```

Sometimes, you can request a return for which no efficient portfolio exists. Based on the previous example, suppose that you want a portfolio with a 5% return (which is the return of the first asset). A portfolio that is fully invested in the first asset, however, is inefficient. `estimateFrontierByReturn` warns if your target returns are outside the range of efficient portfolio returns and replaces it with the endpoint portfolio of the efficient frontier closest to your target return:

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

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByReturn(p, [0.05, 0.09, 0.12]);
```

```
display(pwgt);

Warning: One or more target return values are outside the feasible range [ 0.0590468, 0.18 ].
    Will return portfolios associated with endpoints of the range for these values.
> In Portfolio.estimateFrontierByReturn at 70

pwgt =

    0.8891    0.5032    0.1293
    0.0369    0.2488    0.4541
    0.0404    0.0780    0.1143
    0.0336    0.1700    0.3022
```

The best way to avoid this situation is to bracket your target portfolio returns with `estimateFrontierLimits` and `estimatePortReturn` (see "Obtaining Endpoints of the Efficient Frontier" on page 4-112 and "Obtaining Portfolio Risks and Returns" on page 4-129).

```
pret = estimatePortReturn(p, p.estimateFrontierLimits);

display(pret);

pret =

    0.0590
    0.1800
```

This result indicates that efficient portfolios have returns that range between 5.9% and 18%.

If you have an initial portfolio, `estimateFrontierByReturn` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, to obtain purchases and sales with target returns of 6%, 9%, and 12%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022
```

```
pbuy =

    0.5772    0.2032         0
         0         0    0.1541
         0         0         0
         0    0.0700    0.2022

psell =

         0         0    0.1707
    0.2566    0.0512         0
    0.1584    0.1220    0.0857
    0.0622         0         0
```

If you do not have an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3

- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Obtaining Efficient Portfolios for Target Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` function accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Suppose that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

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

 p = Portfolio;
 p = setAssetMoments(p, m, C);
 p = setDefaultConstraints(p);
 pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

 display(pwgt);

pwgt =

    0.3984    0.2659    0.1416
    0.3064    0.3791    0.4474
    0.0882    0.1010    0.1131
    0.2071    0.2540    0.2979
```

Sometimes, you can request a risk for which no efficient portfolio exists. Based on the previous example, suppose that you want a portfolio with 7% risk (individual assets in this universe have risks ranging from 8% to 35%). It turns out that a portfolio with 7% risk cannot be formed with these four assets. `estimateFrontierByRisk` warns if your target risks are outside the range of efficient portfolio risks and replaces it with the endpoint of the efficient frontier closest to your target risk:

```
pwgt = estimateFrontierByRisk(p, 0.07)

Warning: One or more target risk values are outside the feasible range [ 0.0769288, 0.35 ].
    Will return portfolios associated with endpoints of the range for these values.
> In Portfolio.estimateFrontierByRisk at 82

pwgt =

    0.8891
    0.0369
```

```
    0.0404
    0.0336
```

The best way to avoid this situation is to bracket your target portfolio risks with `estimateFrontierLimits` and `estimatePortRisk` (see "Obtaining Endpoints of the Efficient Frontier" on page 4-112 and "Obtaining Portfolio Risks and Returns" on page 4-129).

```
prsk = estimatePortRisk(p, p.estimateFrontierLimits);

display(prsk);
```

```
prsk =

    0.0769
    0.3500
```

This result indicates that efficient portfolios have risks that range from 7.7% to 35%.

Starting with an initial portfolio, `estimateFrontierByRisk` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the example with target risks of 12%, 14%, and 16%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.3984    0.2659    0.1416
    0.3064    0.3791    0.4474
    0.0882    0.1010    0.1131
    0.2071    0.2540    0.2979


pbuy =

    0.0984         0         0
    0.0064    0.0791    0.1474
         0         0         0
    0.1071    0.1540    0.1979
```

```
psell =

         0    0.0341    0.1584
         0         0         0
    0.1118    0.0990    0.0869
         0         0         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Efficient Portfolio That Maximizes Sharpe Ratio

The Sharpe ratio is defined as the ratio

$$\frac{\mu(x) - r_0}{\sqrt{\sum(x)}}$$

where $x \in R^n$ and r_0 is the risk-free rate (μ and Σ proxies for portfolio return and risk). For more information, see "Portfolio Optimization Theory" on page 4-3.

Portfolios that maximize the Sharpe ratio are portfolios on the efficient frontier that satisfy a number of theoretical conditions in finance. For example, such portfolios are called tangency portfolios since the tangent line from the risk-free rate to the efficient frontier touches the efficient frontier at portfolios that maximize the Sharpe ratio.

To obtain efficient portfolios that maximizes the Sharpe ratio, the `estimateMaxSharpeRatio` function accepts a Portfolio object and obtains efficient portfolios that maximize the Sharpe Ratio.

Suppose that you have a universe with four risky assets and a riskless asset and you want to obtain a portfolio that maximizes the Sharpe ratio, where, in this example, r_0 is the return for the riskless asset.

```
r0 = 0.03;
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio('RiskFreeRate', r0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateMaxSharpeRatio(p);

display(pwgt);

pwgt =

    0.4251
    0.2917
```

```
    0.0856
    0.1977
```

If you start with an initial portfolio, `estimateMaxSharpeRatio` also returns purchases and sales to get from your initial portfolio to the portfolio that maximizes the Sharpe ratio. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the previous example:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateMaxSharpeRatio(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.4251
    0.2917
    0.0856
    0.1977


pbuy =

    0.1251
         0
         0
    0.0977


psell =

         0
    0.0083
    0.1144
         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that you initial portfolio is `0`.

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Choosing and Controlling the Solver for Mean-Variance Portfolio Optimization

The default solver for mean-variance portfolio optimization is `lcprog`, which implements a linear complementarity programming (LCP) algorithm. Although `lcprog` works for most problems, you can adjust arguments to control the algorithm. Alternatively, the mean-variance portfolio optimization tools let you use any of the variations of `quadprog` from Optimization Toolbox™ software. Unlike Optimization Toolbox which uses the `trust-region-reflective` algorithm as the default algorithm for `quadprog`, the portfolio optimization tools use the `interior-point-convex` algorithm. For details about `quadprog` and quadratic programming algorithms and options, see "Quadratic Programming Algorithms" (Optimization Toolbox).

To modify either `lcprog` or to specify `quadprog` as your solver, use the `setSolver` function to set the hidden properties `solverType` and `solverOptions` that specify and control the solver. Since the solver properties are hidden, you cannot set these using the `Portfolio` function. The default solver is `lcprog` so you do not need to use `setSolver` to specify this solver. To use `quadprog`, you must set up the `interior-point-convex` version of `quadprog` using:

```
p = Portfolio;
p = setSolver(p, 'quadprog');
display(p.solverType);
```

```
quadprog
```

and you can switch back to`lcprog` with:

```
p = setSolver(p, 'lcprog');
display(p.solverType);
```

```
lcprog
```

In both cases, `setSolver` sets up default options associated with either solver. If you want to specify additional options associated with a given solver, `setSolver` accepts these options with argument name-value pair arguments in the function call. For example, if you intend to use `quadprog` and want to use the `active-set` algorithm, call `setSolver` with:

```
p = setSolver(p, 'quadprog', 'Algorithm', 'active-set');
display(p.solverOptions.Algorithm);
```

```
active-set
```

In addition, if you want to specify any of the options for `quadprog` that are normally set through `optimoptions`, `setSolver` accepts an `optimoptions` object as the second argument. For example, you can start with the default options for `quadprog` set by `setSolver` and then change the algorithm to `'trust-region-reflective'` with no displayed output:

```
p = Portfolio;
options = optimoptions('quadprog', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'quadprog', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

See Also

`Portfolio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Estimate Efficient Frontiers for Portfolio Object

Whereas "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109 focused on estimation of efficient portfolios, this section focuses on the estimation of efficient frontiers. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

Obtaining Portfolio Risks and Returns

Given any portfolio and, in particular, efficient portfolios, the functions `estimatePortReturn`, `estimatePortRisk`, and `estimatePortMoments` provide estimates for the return (or return proxy), risk (or the risk proxy), and, in the case of mean-variance portfolio optimization, the moments of expected portfolio returns. Each function has the same input syntax but with different combinations of outputs. Suppose that you have this following portfolio optimization problem that gave you a collection of portfolios along the efficient frontier in `pwgt`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = Portfolio('AssetMean', m, 'AssetCovar', C, 'InitPort', pwgt0);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
```

Given `pwgt0` and `pwgt`, use the portfolio risk and return estimation functions to obtain risks and returns for your initial portfolio and the portfolios on the efficient frontier:

```
[prsk0, pret0] = estimatePortMoments(p, pwgt0);
[prsk, pret] = estimatePortMoments(p, pwgt);
```

or

```
prsk0 = estimatePortRisk(p, pwgt0);
pret0 = estimatePortReturn(p, pwgt0);
prsk = estimatePortRisk(p, pwgt);
pret = estimatePortReturn(p, pwgt);
```

In either case, you obtain these risks and returns:

```
display(prsk0);
display(pret0);
display(prsk);
display(pret);

prsk0 =

    0.1103

pret0 =

    0.0870

prsk =

    0.0769
    0.0831
    0.0994
    0.1217
    0.1474
    0.1750
    0.2068
    0.2487
    0.2968
    0.3500

pret =

    0.0590
    0.0725
    0.0859
    0.0994
    0.1128
    0.1262
    0.1397
    0.1531
    0.1666
    0.1800
```

The returns and risks are at the periodicity of the moments of asset returns so that, if you have values for `AssetMean` and `AssetCovar` in terms of monthly returns, the estimates for portfolio risk and return are in terms of monthly returns as well. In addition, the estimate for portfolio risk in the mean-variance case is the standard deviation of portfolio returns, not the variance of portfolio returns.

See Also

`Portfolio` | `estimatePortMoments` | `estimatePortReturn` | `plotFrontier`

Related Examples

- "Plotting the Efficient Frontier for a Portfolio Object" on page 4-132
- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Postprocessing Results to Set Up Tradable Portfolios" on page 4-138
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- "Portfolio Object" on page 4-23
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-21

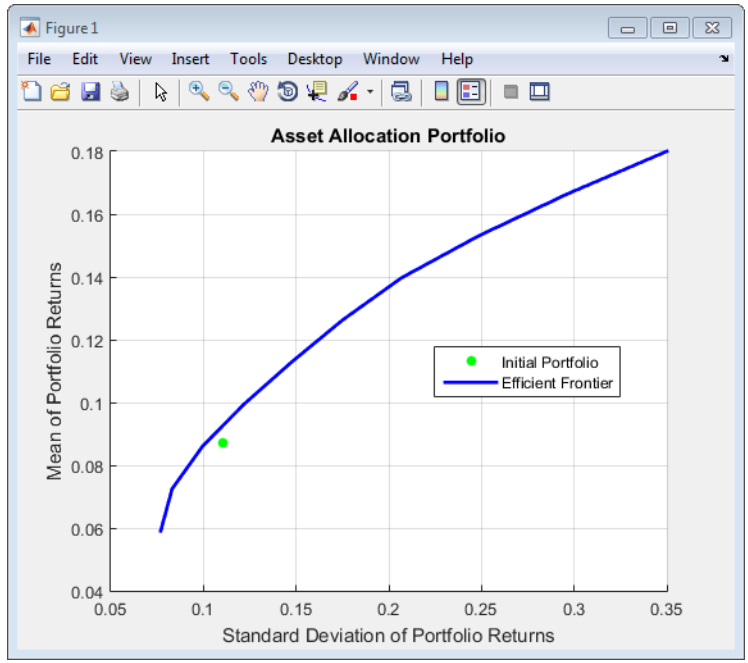
External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Plotting the Efficient Frontier for a Portfolio Object

The `plotFrontier` function creates a plot of the efficient frontier for a given portfolio optimization problem. This function accepts several types of inputs and generates a plot with an optional possibility to output the estimates for portfolio risks and returns along the efficient frontier. `plotFrontier` has four different ways that it can be used. In addition to a plot of the efficient frontier, if you have an initial portfolio in the `InitPort` property, `plotFrontier` also displays the return versus risk of the initial portfolio on the same plot. If you have a well-posed portfolio optimization problem set up in a Portfolio object and you use `plotFrontier`, you will get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is currently 10 and is maintained in the hidden property `defaultNumPorts`). This example illustrates a typical use of `plotFrontier` to create a new plot:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
plotFrontier(p);
```

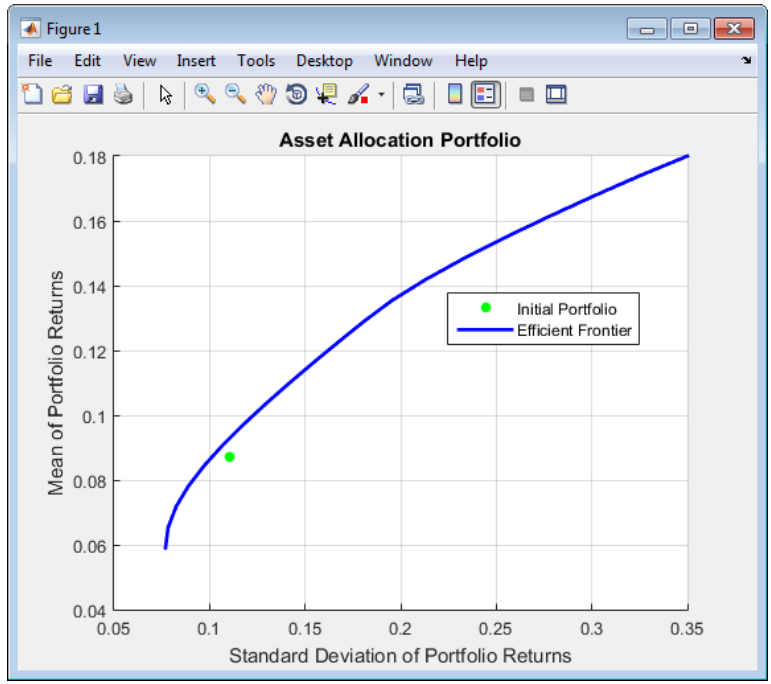


The `Name` property appears as the title of the efficient frontier plot if you set it in the Portfolio object. Without an explicit name, the title on the plot would be "Efficient Frontier." If you want to obtain a specific number of portfolios along the efficient frontier, use `plotFrontier` with the number of portfolios that you want. Suppose that you have the Portfolio object from the previous example and you want to plot 20 portfolios along the efficient frontier and to obtain 20 risk and return values for each portfolio:

```
[prsk, pret] = plotFrontier(p, 20);
display([pret, prsk]);
```

```
ans =

    0.0590    0.0769
    0.0654    0.0784
    0.0718    0.0825
    0.0781    0.0890
    0.0845    0.0973
    0.0909    0.1071
    0.0972    0.1179
    0.1036    0.1296
```

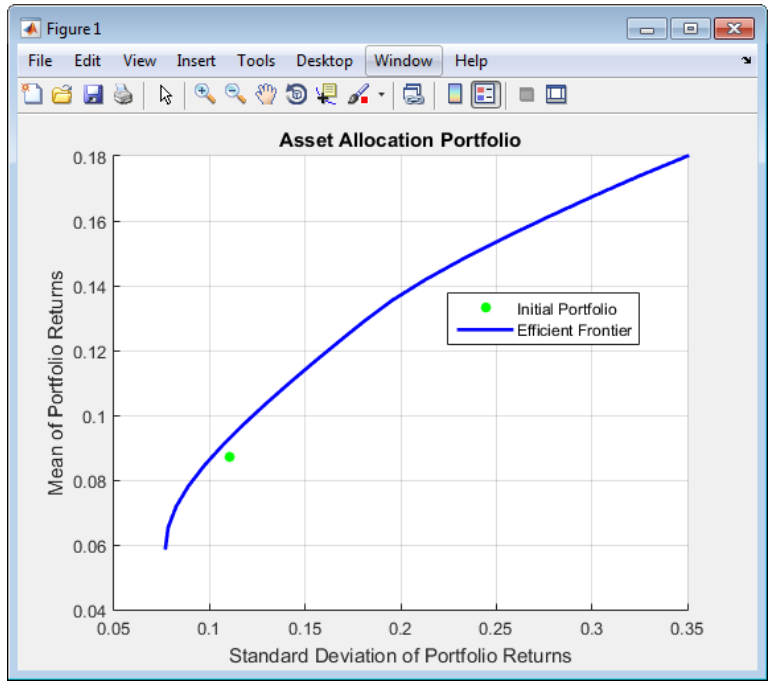
```
    0.1100    0.1418
    0.1163    0.1545
    0.1227    0.1676
    0.1291    0.1810
    0.1354    0.1955
    0.1418    0.2128
    0.1482    0.2323
    0.1545    0.2535
    0.1609    0.2760
    0.1673    0.2995
    0.1736    0.3239
    0.1800    0.3500
```



Plotting Existing Efficient Portfolios

If you already have efficient portfolios from any of the "estimateFrontier" functions (see “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109), pass them into `plotFrontier` directly to plot the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p, 20);
plotFrontier(p, pwgt);
```

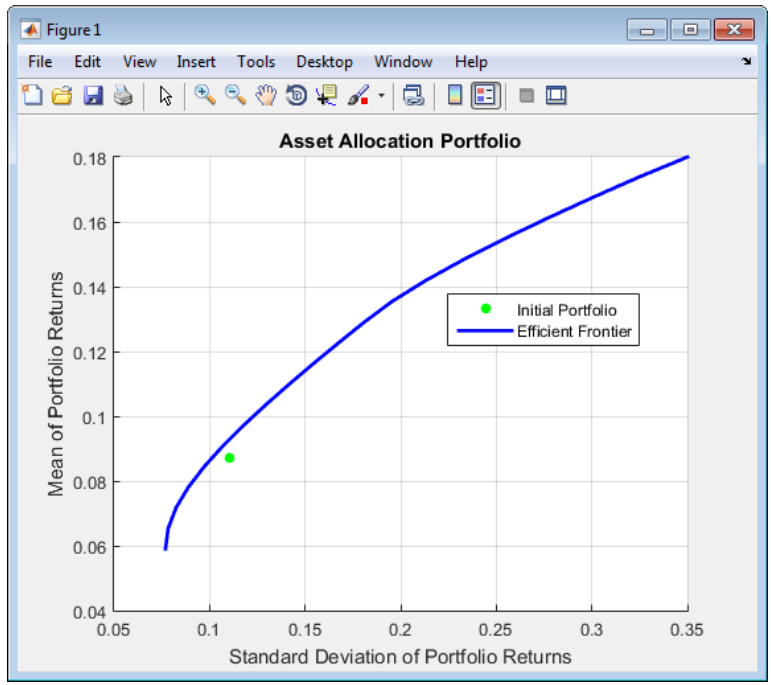


Plotting Existing Efficient Portfolio Risks and Returns

If you already have efficient portfolio risks and returns, you can use the interface to `plotFrontier` to pass them into `plotFrontier` to obtain a plot of the efficient frontier:

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

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
[prsk, pret] = estimatePortMoments(p, p.estimateFrontier(20));
plotFrontier(p, prsk, pret);
```



See Also

`Portfolio` | `estimatePortMoments` | `estimatePortReturn` | `plotFrontier`

Related Examples

- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109

- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-138
- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Postprocessing Results to Set Up Tradable Portfolios

After obtaining efficient portfolios or estimates for expected portfolio risks and returns, use your results to set up trades to move toward an efficient portfolio. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

Setting Up Tradable Portfolios

Suppose that you set up a portfolio optimization problem and obtained portfolios on the efficient frontier. Use the `dataset` object from Statistics and Machine Learning Toolbox™ to form a blotter that lists your portfolios with the names for each asset. For example, suppose that you want to obtain five portfolios along the efficient frontier. You can set up a blotter with weights multiplied by 100 to view the allocations for each portfolio:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

 p = Portfolio('InitPort', pwgt0);
 p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');
 p = setAssetMoments(p, m, C);
 p = setDefaultConstraints(p);
 pwgt = estimateFrontier(p, 5);

 pnames = cell(1,5);
   for i = 1:5
       pnames{i} = sprintf('Port%d',i);
   end

 Blotter = dataset([{100*pwgt},pnames],'obsnames',p.AssetList);
 display(Blotter);
```

```
Blotter =

                        Port1     Port2     Port3     Port4     Port5
    Bonds               88.906    51.216    13.525         0       0
    Large-Cap Equities  3.6875    24.387    45.086    27.479       0
    Small-Cap Equities  4.0425    7.7088    11.375    13.759       0
    Emerging Equities    3.364    16.689    30.014    58.762     100
```

This result indicates that you would invest primarily in bonds at the minimum-risk/minimum-return end of the efficient frontier (`Port1`), and that you would invest completely in emerging equity at the maximum-risk/maximum-return end of the efficient frontier (`Port5`). You can also select a particular efficient portfolio, for example, suppose

that you want a portfolio with 15% risk and you add purchase and sale weights outputs obtained from the “estimateFrontier” functions to set up a trade blotter:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = Portfolio('InitPort', pwgt0);
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, 0.15);

Blotter = dataset([{100*[pwgt0, pwgt, pbuy, psell]}, ...
     {'Initial','Weight', 'Purchases','Sales'}],'obsnames',p.AssetList);

display(Blotter);
```

```
Blotter =

                          Initial    Weight    Purchases    Sales
    Bonds                 30         20.299         0       9.7007
    Large-Cap Equities    30         41.366    11.366            0
    Small-Cap Equities    20         10.716         0       9.2838
    Emerging Equities     10         27.619    17.619            0
```

If you have prices for each asset (in this example, they can be ETFs), add them to your blotter and then use the tools of the `dataset` object to obtain shares and shares to be traded. For an example, see “Asset Allocation Case Study” on page 4-175.

See Also

`Portfolio` | `checkFeasibility` | `estimateAssetMoments`

Related Examples

- “Troubleshooting Portfolio Optimization Results” on page 4-141
- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129

- “Asset Allocation Case Study” on page 4-175
- “Portfolio Optimization Examples” on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Troubleshooting Portfolio Optimization Results

Portfolio Object Destroyed When Modifying

If a Portfolio object is destroyed when modifying, remember to pass an existing object into the `Portfolio` function if you want to modify it, otherwise it creates a new object. See “Creating the Portfolio Object” on page 4-28 for details.

Optimization Fails with “Bad Pivot” Message

If the optimization fails with a "bad pivot" message from `lcprog`, try a larger value for `tolpiv` which is a tolerance for pivot selection in the `lcprog` algorithm (try `1.0e-7`, for example) or try the `interior-point-convex` version of `quadprog`. For details, see “Choosing and Controlling the Solver for Mean-Variance Portfolio Optimization” on page 4-126, the help header for `lcprog`, and `quadprog`.

Speed of Optimization

Although it is difficult to characterize when one algorithm is faster than the other, the default solver, `lcprog` is generally faster for smaller problems and the `quadprog` solver is generally faster for larger problems. If one solver seems to take too much time, try the other solver. To change solvers, use `setSolver`.

Matrix Incompatibility and "Non-Conformable" Errors

If you get matrix incompatibility or "non-conformable" errors, the representation of data in the tools follows a specific set of basic rules described in “Conventions for Representation of Data” on page 4-26.

Missing Data Estimation Fails

If asset return data has missing or `NaN` values, the `estimateAssetMoments` function with the `'missingdata'` flag set to `true` may fail with either too many iterations or a singular covariance. To correct this problem, consider this:

- If you have asset return data with no missing or `NaN` values, you can compute a covariance matrix that may be singular without difficulties. If you have missing or

`NaN` values in your data, the supported missing data feature requires that your covariance matrix must be positive-definite, that is, nonsingular.

- `estimateAssetMoments` uses default settings for the missing data estimation procedure that might not be appropriate for all problems.

In either case, you might want to estimate the moments of asset returns separately with either the ECM estimation functions such as `ecmnmle` or with your own functions.

mv_optim_transform Errors

If you obtain optimization errors such as:

```
Error using mv_optim_transform (line 233)
Portfolio set appears to be either empty or unbounded. Check constraints.

Error in Portfolio/estimateFrontier (line 63)
    [A, b, f0, f, H, g, lb] = mv_optim_transform(obj);
```

or

```
Error using mv_optim_transform (line 238)
Cannot obtain finite lower bounds for specified portfolio set.

Error in Portfolio/estimateFrontier (line 63)
    [A, b, f0, f, H, g, lb] = mv_optim_transform(obj);
```

Since the portfolio optimization tools require a bounded portfolio set, these errors (and similar errors) can occur if your portfolio set is either empty and, if nonempty, unbounded. Specifically, the portfolio optimization algorithm requires that your portfolio set have at least a finite lower bound. The best way to deal with these problems is to use the validation functions in “Validate the Portfolio Problem for Portfolio Object” on page 4-104. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover or tracking-error and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

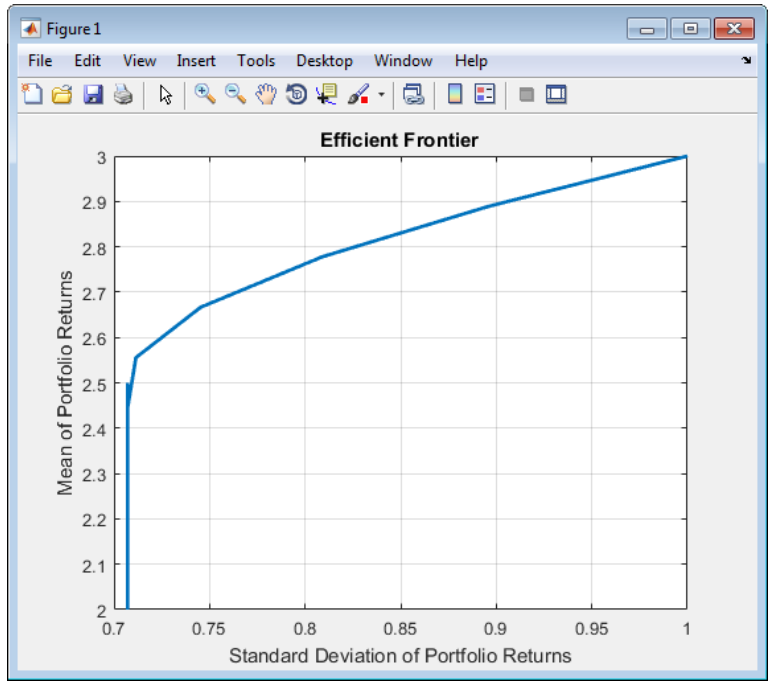
If you obtain efficient portfolios that do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between `0` and `1` and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

Efficient Frontiers Do Not Make Sense

If you obtain efficient frontiers that do not seem to make sense, this can happen for some cases of mean and covariance of asset returns. It is possible for some mean-variance portfolio optimization problems to have difficulties at the endpoints of the efficient frontier. It is generally rare for standard problems but can occur with, for example, unusual combinations of turnover constraints and transaction costs. In most cases, the workaround of setting the hidden property `enforcePareto` produces a single portfolio for the entire efficient frontier, where any other solutions are not Pareto optimal (which is what efficient portfolios must be).

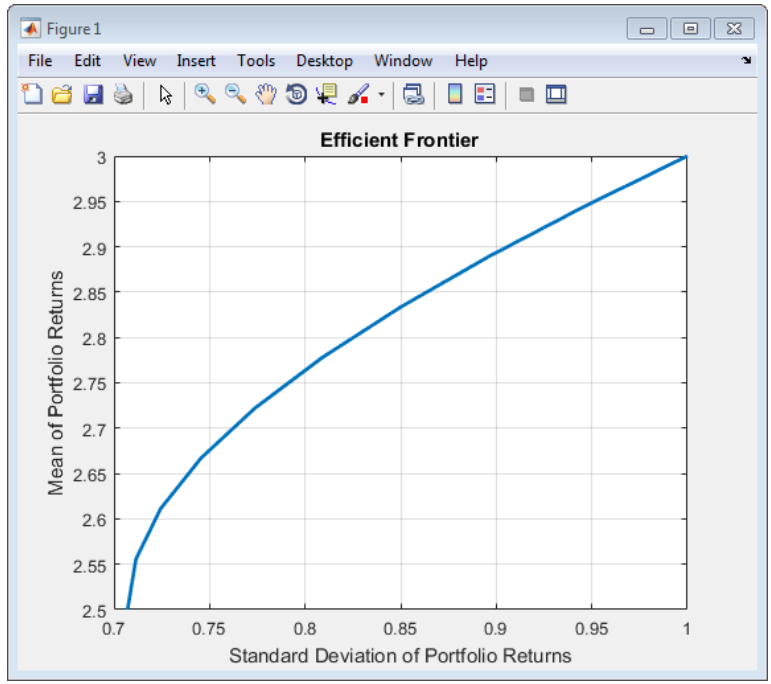
An example of a portfolio optimization problem that has difficulties at the endpoints of the efficient frontier is this standard mean-variance portfolio problem (long-only with a budget constraint) with the following mean and covariance of asset returns:

```
m = [ 1; 2; 3 ];
C = [ 1 1 0; 1 1 0; 0 0 1 ];

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

plotFrontier(p);
```



To work around this problem, set the hidden Portfolio object property for `enforcePareto`. This property instructs the optimizer to perform extra steps to ensure a Pareto-optimal solution. This slows down the solver, but guarantees a Pareto-optimal solution.

```
p.enforcePareto = true;
plotFrontier(p);
```



See Also

`Portfolio` | `checkFeasibility` | `estimateAssetMoments`

Related Examples

- "Postprocessing Results to Set Up Tradable Portfolios" on page 4-138
- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Optimization Examples

The following sequence of examples highlights features of the `Portfolio` object in the Financial Toolbox™. Specifically, the examples show how to set up mean-variance portfolio optimization problems that focus on the two-fund theorem, the impact of transaction costs and turnover constraints, how to obtain portfolios that maximize the Sharpe ratio, and how to set up two popular hedge-fund strategies - dollar-neutral and 130-30 portfolios.

Set up the Data

Every example works with moments for monthly total returns of a universe of 30 "blue-chip" stocks. Although derived from real data, these data are for illustrative purposes and are not meant to be representative of specific assets or of market performance. The data are contained in the file `BlueChipStockMoments.mat` with a list of asset identifiers in the variable `AssetList`, a mean and covariance of asset returns in the variables `AssetMean` and `AssetCovar`, and the mean and variance of cash and market returns in the variables `CashMean`, `CashVar`, `MarketMean`, and `MarketVar`. Since most of the analysis requires the use of the standard deviation of asset returns as the proxy for risk, cash and market variances are converted into standard deviations.

```
load BlueChipStockMoments

mret = MarketMean;
mrsk = sqrt(MarketVar);
cret = CashMean;
crsk = sqrt(CashVar);
```

Create a Portfolio Object

The first step is to create a "standard" `Portfolio` object with the `Portfolio` constructor and to incorporate the list of assets, the risk-free rate, and the moments of asset returns into the object.

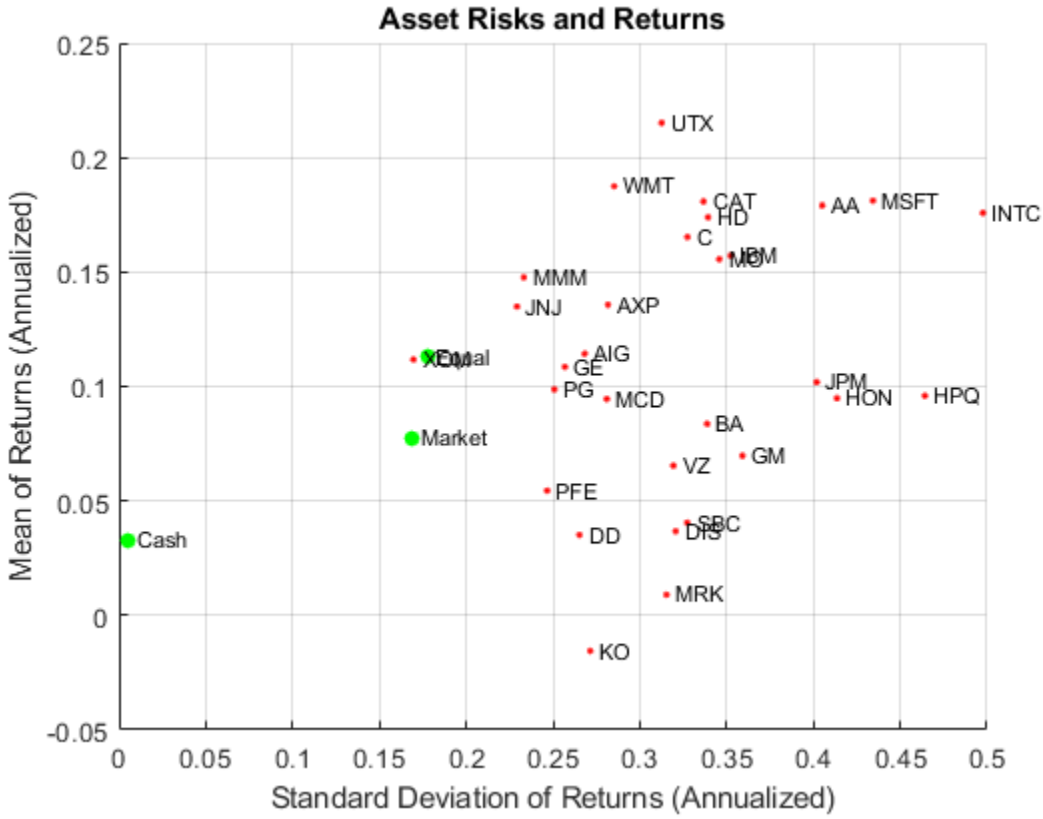
```
p = Portfolio('AssetList', AssetList, 'RiskFreeRate', CashMean);
p = setAssetMoments(p, AssetMean, AssetCovar);
```

To provide a basis for comparison, set up an equal-weight portfolio and make it the initial portfolio in the `Portfolio` object. Keep in mind that the hedged portfolios to be constructed later will require a different initial portfolio. Once the initial portfolio is created, the `estimatePortMoments` function estimates the mean and standard deviation of equal-weight portfolio returns.

```
p = setInitPort(p, 1/p.NumAssets);
[ersk, eret] = estimatePortMoments(p, p.InitPort);
```

A specialized "helper" function `portfolioexamples_plot` makes it possible to plot all results to be developed here. This first plot shows the distribution of individual assets according to their means and standard deviations of returns. In addition, the equal-weight, market, and cash portfolios are plotted on the same plot. Note that the plot function converts monthly total returns into annualized total returns.

```
clf;
portfolioexamples_plot('Asset Risks and Returns', ...
    {'scatter', mrsk, mret, {'Market'}}, ...
    {'scatter', crsk, cret, {'Cash'}}, ...
    {'scatter', ersk, eret, {'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```

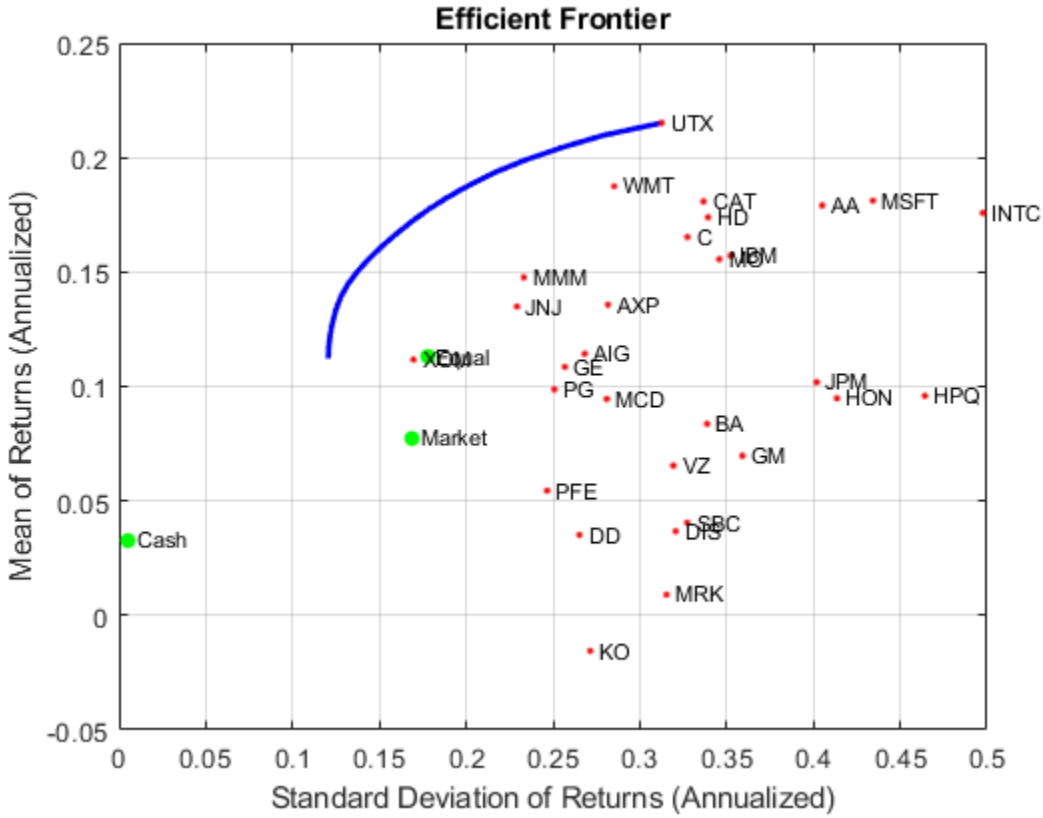


Set up a Portfolio Optimization Problem

Set up a "standard" or default mean-variance portfolio optimization problem with the `setDefaultConstraints` function that requires fully-invested long-only portfolios (non-negative weights that must sum to `1`). Given this initial problem, estimate the efficient frontier with the functions `estimateFrontier` and `estimatePortMoments`, where `estimateFrontier` estimates efficient portfolios and `estimatePortMoments` estimates risks and returns for portfolios. The next figure overlays the efficient frontier on the previous plot.

```
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p, 20);
```

```
[prsk, pret] = estimatePortMoments(p, pwgt);

% Plot efficient frontier

clf;
portfolioexamples_plot('Efficient Frontier', ...
    {'line', prsk, pret}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



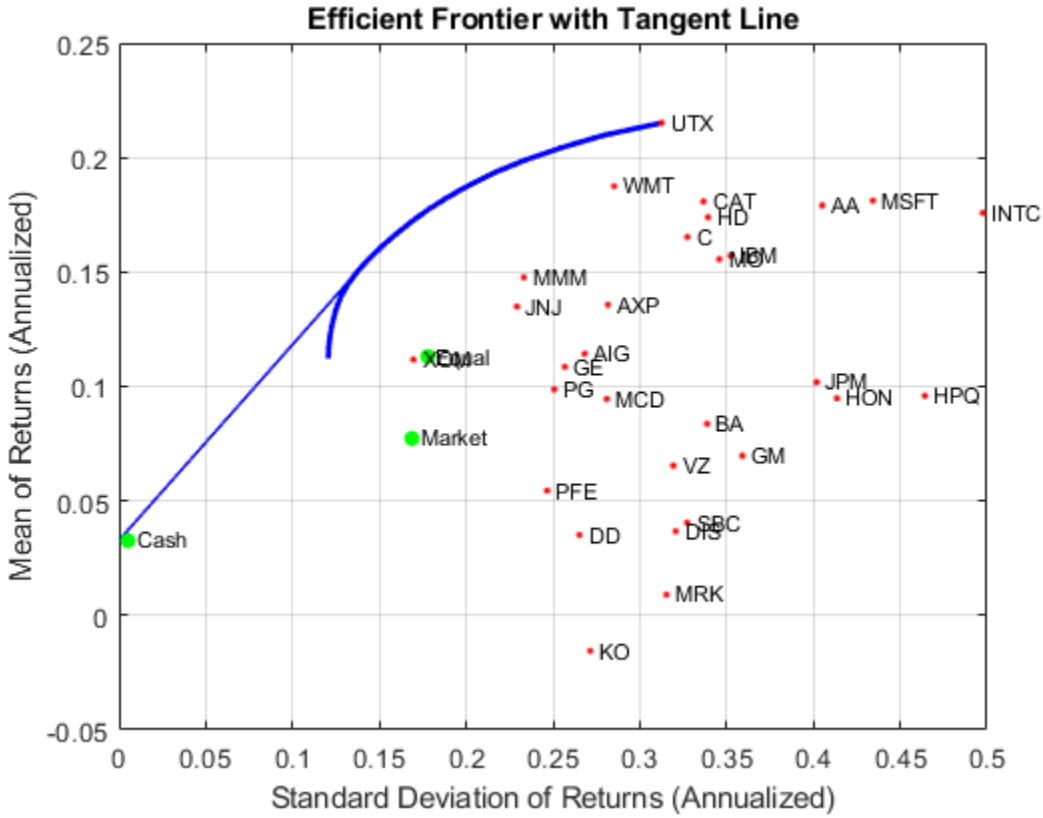
Illustrate the Tangent Line to the Efficient Frontier

Tobin's mutual fund theorem (Tobin 1958) says that the portfolio allocation problem can be viewed as a decision to allocate between a riskless asset and a risky portfolio. In the

mean-variance framework, cash can serve as a proxy for a riskless asset and an efficient portfolio on the efficient frontier serves as the risky portfolio such that any allocation between cash and this portfolio dominates all other portfolios on the efficient frontier. This portfolio is called a *tangency portfolio* because it is located at the point on the efficient frontier where a tangent line that originates at the riskless asset touches the efficient frontier.

Given that the `Portfolio` object already has the risk-free rate, obtain the tangent line by creating a copy of the `Portfolio` object with a budget constraint that permits allocation between 0% and 100% in cash. Since the `Portfolio` object is a value object, it is easy to create a copy by assigning the output of either the constructor or set functions to a new instance of the `Portfolio` object. The plot shows the efficient frontier with Tobin's allocations that form the tangent line to the efficient frontier.

```
q = setBudget(p, 0, 1);

qwgt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

% Plot efficient frontier with tangent line (0 to 1 cash)

clf;
portfolioexamples_plot('Efficient Frontier with Tangent Line', ...
    {'line', prsk, pret}, ...
    {'line', qrsk, qret, [], [], 1}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



Note that cash actually has a small risk so that the tangent line does not pass through the cash asset.

Obtain Range of Risks and Returns

To obtain efficient portfolios with target values of either risk or return, it is necessary to obtain the range of risks and returns among all portfolios on the efficient frontier. This can be accomplished with the `estimateFrontierLimits` function.

```
[rsk, ret] = estimatePortMoments(p, estimateFrontierLimits(p));

display(rsk);
```

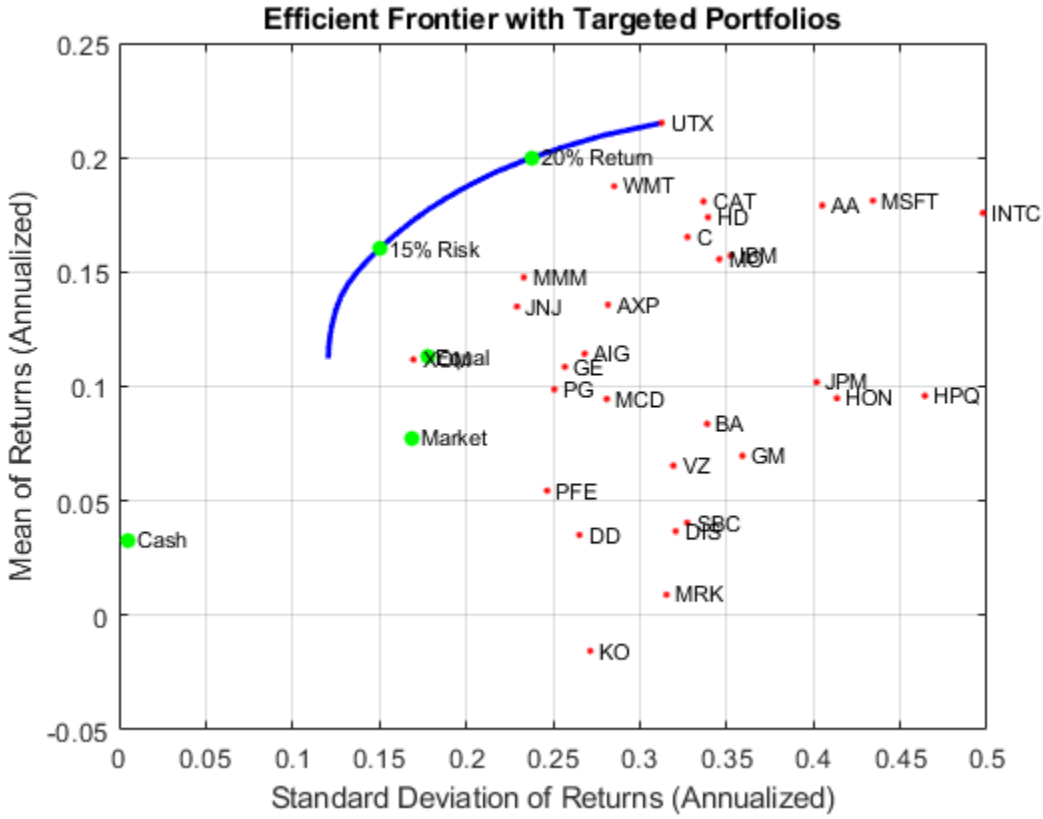
```
rsk =

    0.0348
    0.0903


display(ret);

ret =

    0.0094
    0.0179
```

The range of monthly portfolio returns is between 0.9% and 1.8% and the range for portfolio risks is between 3.5% and 9.0%. In annualized terms, the range of portfolio returns is 11.2% to 21.5% and the range of portfolio risks is 12.1% to 31.3%.

Find a Portfolio with a Targeted Return and Targeted Risk

Given the range of risks and returns, it is possible to locate specific portfolios on the efficient frontier that have target values for return and risk using the functions `estimateFrontierByReturn` and `estimateFrontierByRisk`.

```
TargetReturn = 0.20;            % input target annualized return and risk here
TargetRisk = 0.15;

% Obtain portfolios with targeted return and risk

awgt = estimateFrontierByReturn(p, TargetReturn/12);
[arsk, aret] = estimatePortMoments(p, awgt);

bwgt = estimateFrontierByRisk(p, TargetRisk/sqrt(12));
[brsk, bret] = estimatePortMoments(p, bwgt);

% Plot efficient frontier with targeted portfolios

clf;
portfolioexamples_plot('Efficient Frontier with Targeted Portfolios', ...
    {'line', prsk, pret}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', arsk, aret, {sprintf('%g%% Return',100*TargetReturn)}}, ...
    {'scatter', brsk, bret, {sprintf('%g%% Risk',100*TargetRisk)}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



To see what these targeted portfolios look like, use the `dataset` object to set up "blotters" that contain the portfolio weights and asset names (which are obtained from the `Portfolio` object).

```
aBlotter = dataset({100*awgt(awgt > 0),'Weight'}, 'obsnames', p.AssetList(awgt > 0));

displayPortfolio(sprintf('Portfolio with %g%% Target Return', 100*TargetReturn), aBlott
```

```
Portfolio with 20% Target Return
           Weight
    CAT     1.1445
    INTC   0.17452
    MO      9.6521
    MSFT   0.85862
```

```
    UTX      56.918
    WMT      31.253
```

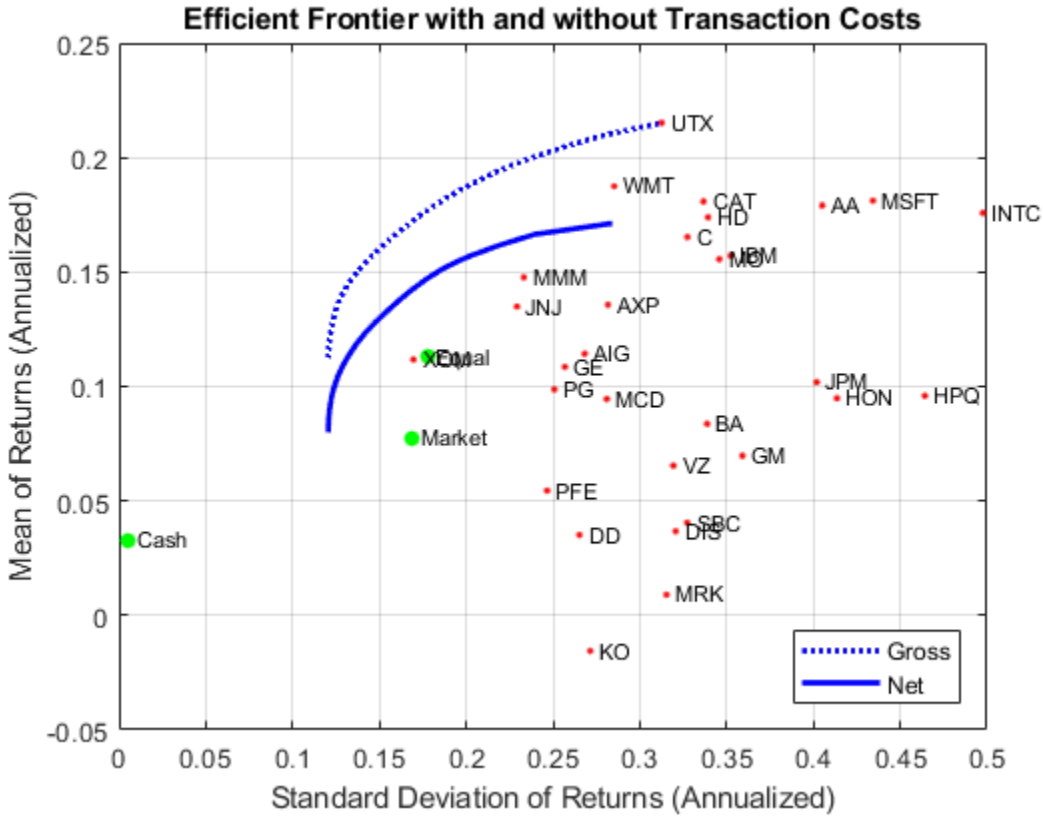
```
bBlotter = dataset({100*bwgt(bwgt > 0),'Weight'}, 'obsnames', p.AssetList(bwgt > 0));

displayPortfolio(sprintf('Portfolio with %g%% Target Risk', 100*TargetRisk), bBlotter,
```

```
Portfolio with 15% Target Risk
           Weight
    INTC    2.2585
    JNJ     9.2162
    MMM     16.603
    MO      15.388
    MSFT    4.4467
    PG       4.086
    UTX     10.281
    WMT     25.031
    XOM      12.69
```

Transactions Costs

The `Portfolio` object makes it possible to account for transaction costs as part of the optimization problem. Although individual costs can be set for each asset, use the scalar expansion features of the Portfolio object's functions to set up uniform transaction costs across all assets and compare efficient frontiers with gross versus net portfolio returns.

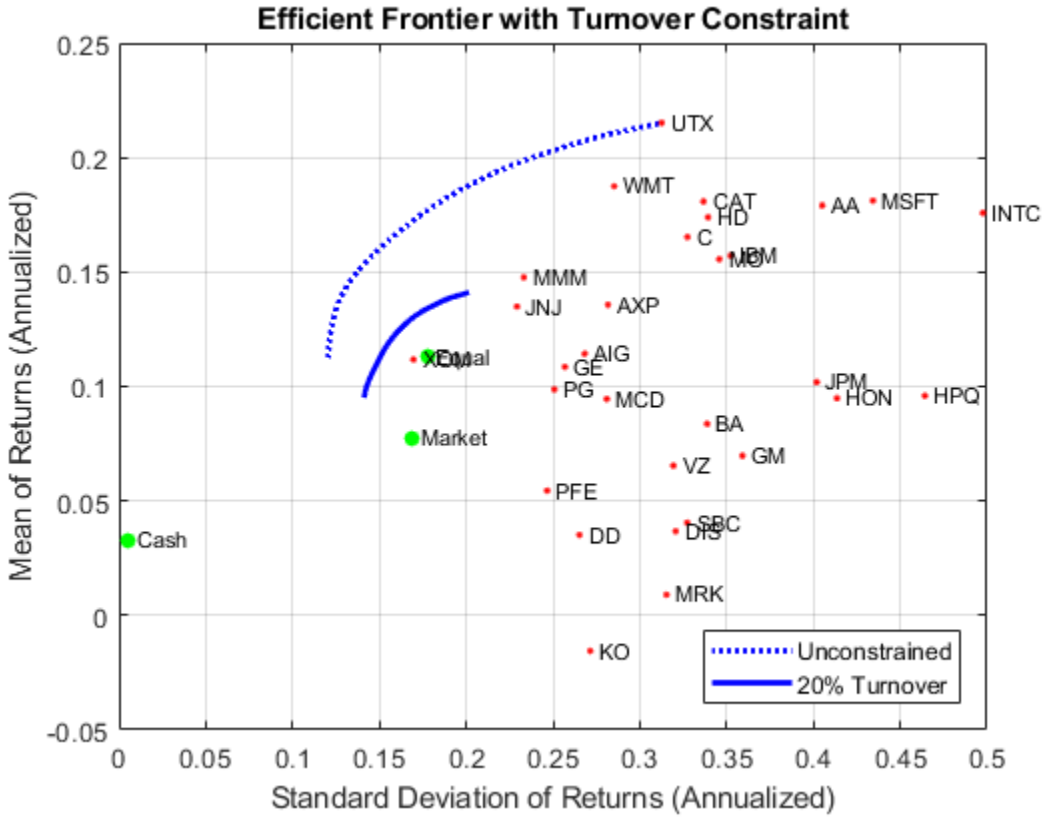
```
BuyCost = 0.0020;
SellCost = 0.0020;

q = setCosts(p, BuyCost, SellCost);

qwgt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

% Plot efficient frontiers with gross and net returns

clf;
portfolioexamples_plot('Efficient Frontier with and without Transaction Costs', ...
    {'line', prsk, pret, {'Gross'}, ':b'}, ...
    {'line', qrsk, qret, {'Net'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



Turnover Constraint

In addition to transaction costs, the `Portfolio` object can handle turnover constraints. The following example demonstrates that a turnover constraint produces an efficient frontier in the neighborhood of an initial portfolio that may restrict trading. Moreover, the introduction of a turnover constraint often implies that multiple trades may be necessary to shift from an initial portfolio to an unconstrained efficient frontier. Consequently, the turnover constraint introduces a form of time diversification that can spread trades out over multiple time periods. In this example, note that the sum of purchases and sales from the `estimateFrontier` function confirms that the turnover constraint has been satisfied.

```
BuyCost = 0.0020;
SellCost = 0.0020;
Turnover = 0.2;

q = setCosts(p, BuyCost, SellCost);
q = setTurnover(q, Turnover);

[qwgt, qbuy, qsell] = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

% Plot efficient frontier with turnover constraint

clf;
portfolioexamples_plot('Efficient Frontier with Turnover Constraint', ...
    {'line', prsk, pret, {'Unconstrained'}, ':b'}, ...
    {'line', qrsk, qret, {sprintf('%g%% Turnover', 100*Turnover)}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



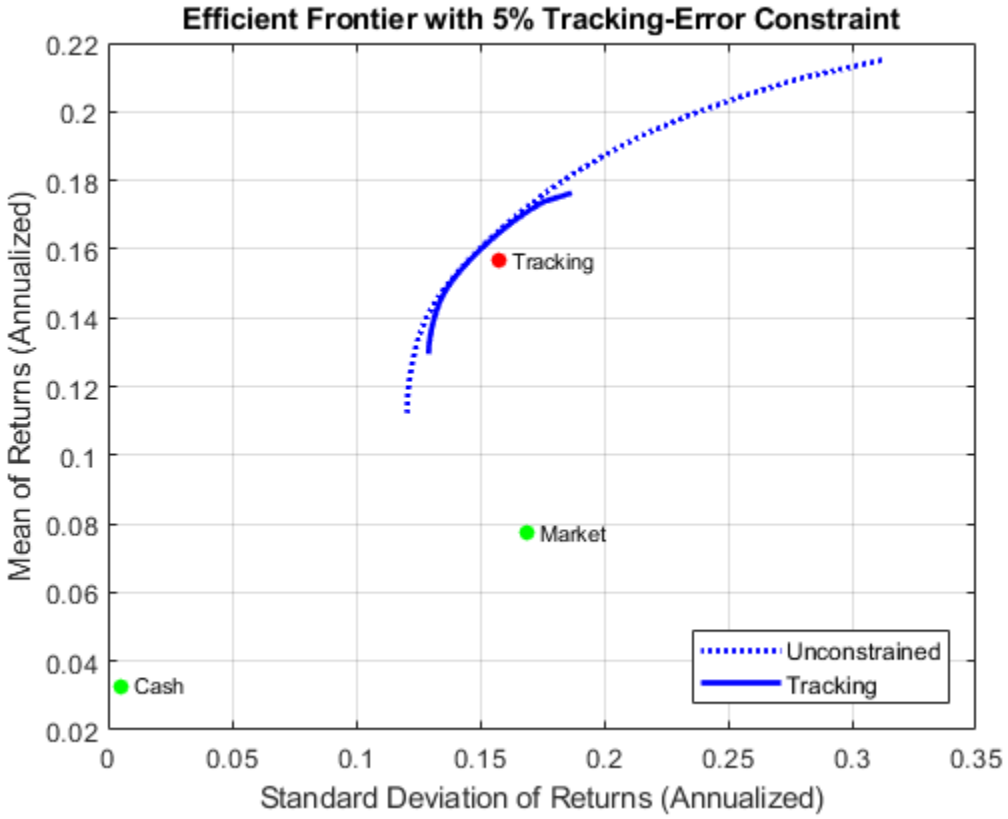

```
displaySumOfTransactions(Turnover, qbuy, qsell)
```

```
Sum of Purchases by Portfolio along Efficient Frontier (Max. Turnover 20%)
20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000
Sum of Sales by Portfolio along Efficient Frontier (Max. Turnover 20%)
20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000
```

Tracking-Error Constraint

The `Portfolio` object can handle tracking-error constraints, where tracking-error is the relative risk of a portfolio compared with a tracking portfolio. In this example, a sub-collection of nine assets forms an equally-weighted tracking portfolio. The goal is to find efficient portfolios with tracking errors that are within 5% of this tracking portfolio.

```
ii = [15, 16, 20, 21, 23, 25, 27, 29, 30];        % indexes of assets to include in tra

TrackingError = 0.05/sqrt(12);
TrackingPort = zeros(30, 1);
TrackingPort(ii) = 1;
TrackingPort = (1/sum(TrackingPort))*TrackingPort;

q = setTrackingError(p, TrackingError, TrackingPort);

qwgt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

[trsk, tret] = estimatePortMoments(q, TrackingPort);

% Plot efficient frontier with tracking-error constraint

clf;
portfolioexamples_plot('Efficient Frontier with 5% Tracking-Error Constraint', ...
    {'line', prsk, pret, {'Unconstrained'}, ':b'}, ...
    {'line', qrsk, qret, {'Tracking'}}, ...
    {'scatter', [mrsk, crsk], [mret, cret], {'Market', 'Cash'}}, ...
    {'scatter', trsk, tret, {'Tracking'}, 'r'});
```

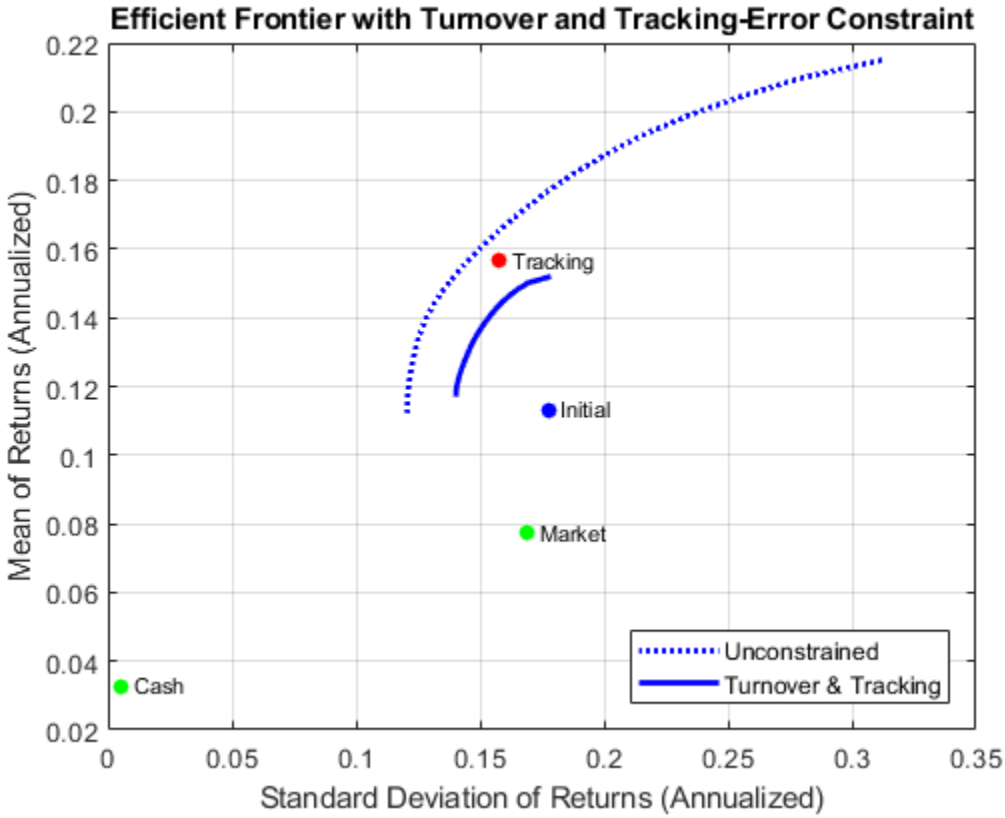


Combined Turnover and Tracking-Error Constraints

This example illustrates the interactions that can occur with combined constraints. In this case, both a turnover constraint relative to an initial equal-weight portfolio and a tracking-error constraint relative to a tracking portfolio must be satisfied. The turnover constraint has maximum 30% turnover and the tracking-error constraint has maximum 5% tracking error. Note that the turnover to get from the initial portfolio to the tracking portfolio is 70% so that an upper bound of 30% turnover means that the efficient frontier will lie somewhere between the initial portfolio and the tracking portfolio.

```
Turnover = 0.3;
InitPort = (1/q.NumAssets)*ones(q.NumAssets, 1);

ii = [15, 16, 20, 21, 23, 25, 27, 29, 30];        % indexes of assets to include in tra
```

```
TrackingError = 0.05/sqrt(12);
TrackingPort = zeros(30, 1);
TrackingPort(ii) = 1;
TrackingPort = (1/sum(TrackingPort))*TrackingPort;

q = setTurnover(q, Turnover, InitPort);

qwgt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

[trsk, tret] = estimatePortMoments(q, TrackingPort);
[ersk, eret] = estimatePortMoments(q, InitPort);

% Plot efficient frontier with combined turnover and tracking-error constraint

clf;
portfolioexamples_plot('Efficient Frontier with Turnover and Tracking-Error Constraint'
    {'line', prsk, pret, {'Unconstrained'}, ':b'}, ...
    {'line', qrsk, qret, {'Turnover & Tracking'}}, ...
    {'scatter', [mrsk, crsk], [mret, cret], {'Market', 'Cash'}}, ...
    {'scatter', trsk, tret, {'Tracking'}, 'r'}, ...
    {'scatter', ersk, eret, {'Initial'}, 'b'});
```

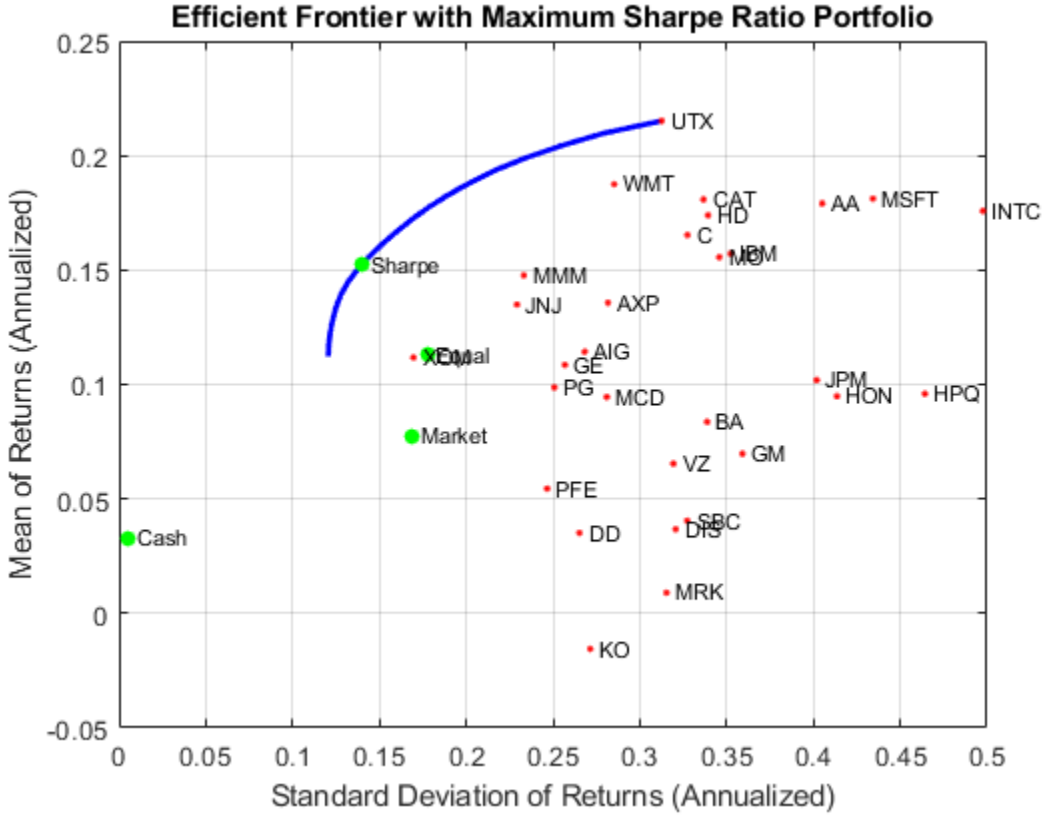


Maximize the Sharpe Ratio

The Sharpe ratio (Sharpe 1966) is a measure of return-to-risk that plays an important role in portfolio analysis. Specifically, a portfolio that maximizes the Sharpe ratio is also the tangency portfolio on the efficient frontier from the mutual fund theorem. The maximum Sharpe ratio portfolio is located on the efficient frontier with the function `estimateMaxSharpeRatio` and the `dataset` object is used to list the assets in this portfolio.

```
p = setInitPort(p, 0);

swgt = estimateMaxSharpeRatio(p);
[srsk, sret] = estimatePortMoments(p, swgt);
```

```
% Plot efficient frontier with portfolio that attains maximum Sharpe ratio

clf;
portfolioexamples_plot('Efficient Frontier with Maximum Sharpe Ratio Portfolio', ...
    {'line', prsk, pret}, ...
    {'scatter', srsk, sret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```




```
% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio

Blotter = dataset({100*swgt(swgt > 0),'Weight'}, 'obsnames', AssetList(swgt > 0));
```

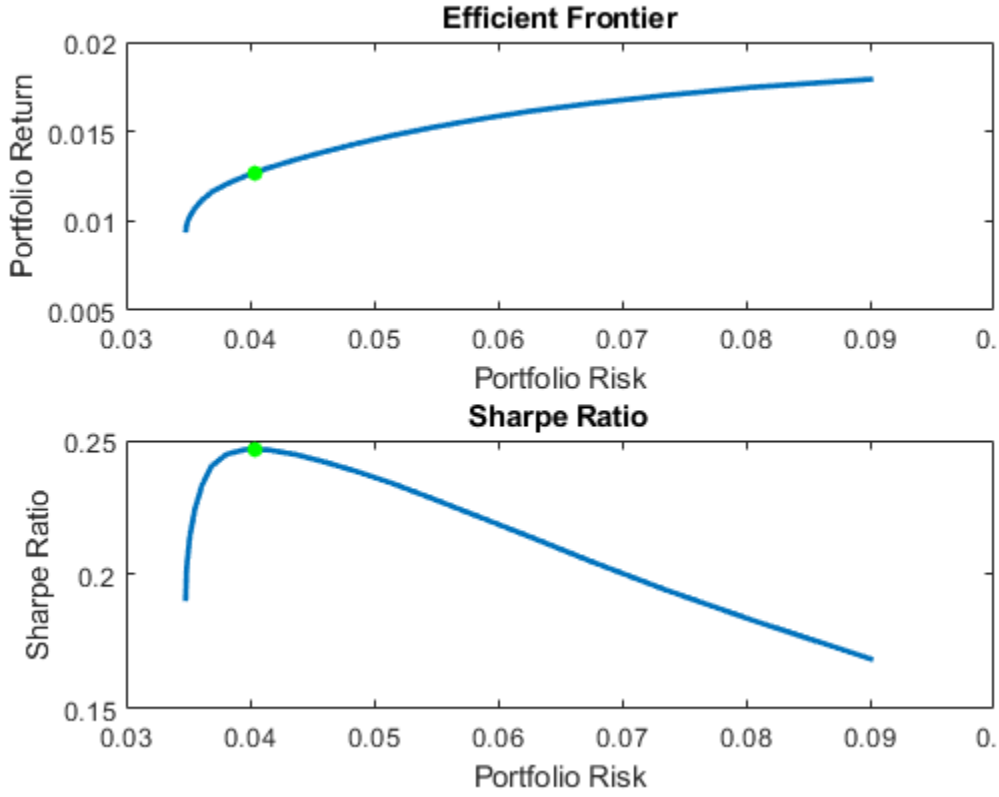
```
displayPortfolio('Portfolio with Maximum Sharpe Ratio', Blotter, false);
```

```
Portfolio with Maximum Sharpe Ratio
           Weight
    INTC    2.6638
    JNJ     9.0044
    MMM     15.502
    MO      13.996
    MSFT    4.4777
    PG      7.4588
    UTX     6.0056
    WMT     22.051
    XOM     18.841
```

Confirm that Maximum Sharpe Ratio is a Maximum

The following plot demonstrates that this portfolio (which is located at the dot on the plots) indeed maximizes the Sharpe ratio among all portfolios on the efficient frontier.

```
psratio = (pret - p.RiskFreeRate) ./ prsk;
ssratio = (sret - p.RiskFreeRate) / srsk;

clf;
subplot(2,1,1);
plot(prsk, pret, 'LineWidth', 2);
hold on
scatter(srsk, sret, 'g', 'filled');
title('\bfEfficient Frontier');
xlabel('Portfolio Risk');
ylabel('Portfolio Return');
hold off

subplot(2,1,2);
plot(prsk, psratio, 'LineWidth', 2);
hold on
scatter(srsk, ssratio, 'g', 'filled');
title('\bfSharpe Ratio');
xlabel('Portfolio Risk');
ylabel('Sharpe Ratio');
hold off
```

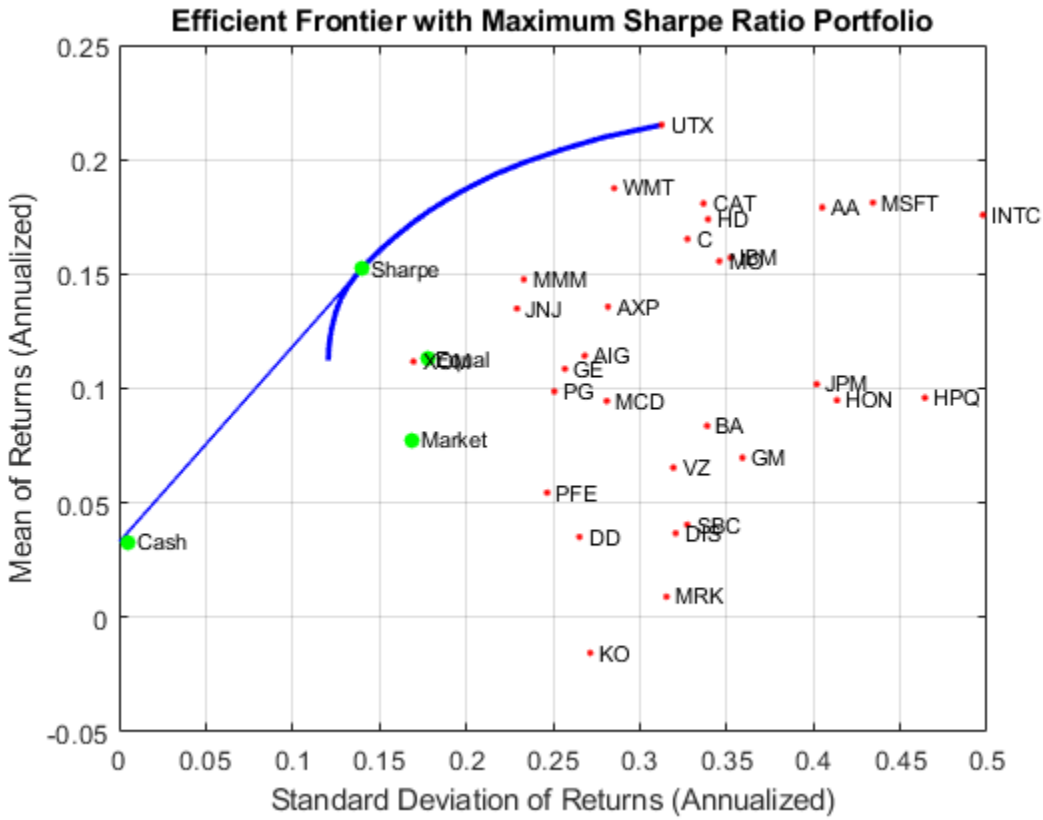


Illustrate that Sharpe is the Tangent Portfolio

The next plot demonstrates that the portfolio that maximizes the Sharpe ratio is also a tangency portfolio (in this case, the budget constraint is opened up to permit between 0% and 100% in cash).

```
q = setBudget(p, 0, 1);

qwgt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

% Plot that shows Sharpe ratio portfolio is the tangency portfolio

clf;
```

```
portfolioexamples_plot('Efficient Frontier with Maximum Sharpe Ratio Portfolio', ...
    {'line', prsk, pret}, ...
    {'line', qrsk, qret, [], [], 1}, ...
    {'scatter', srsk, sret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```

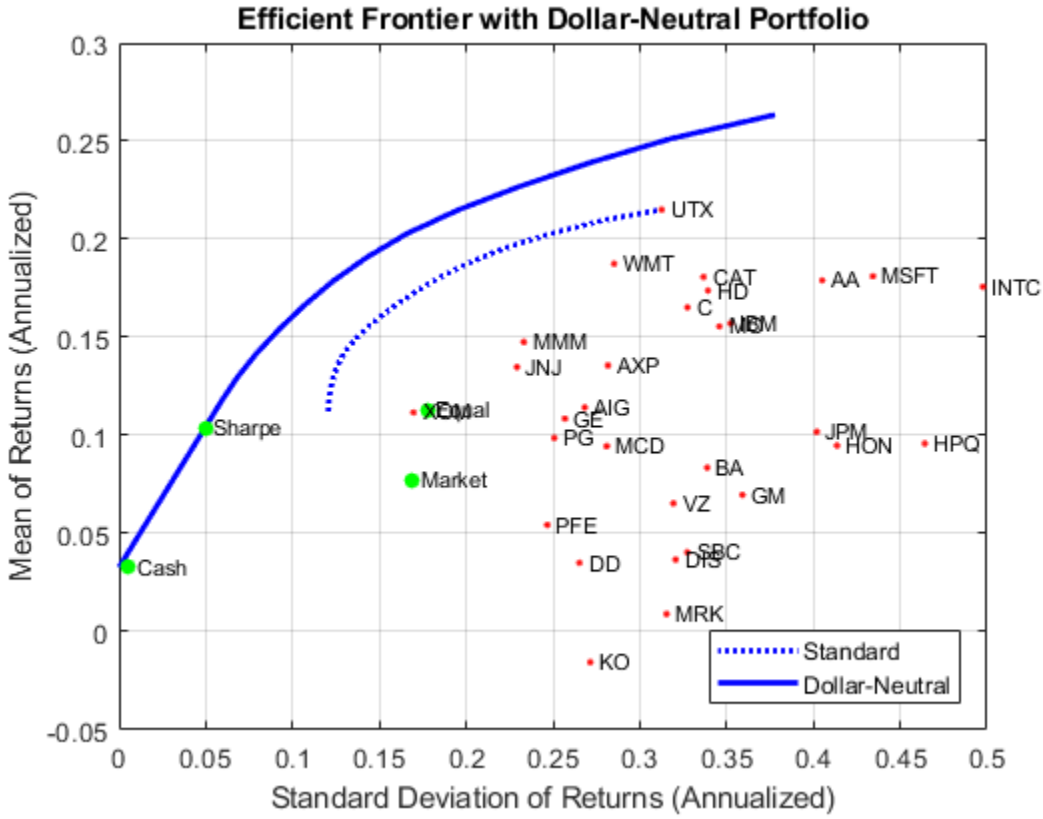


Dollar-Neutral Hedge-Fund Structure

To illustrate how to use the portfolio optimization tools in hedge fund management, two popular strategies with dollar-neutral and 130-30 portfolios are examined. The dollar-neutral strategy invests equally in long and short positions such that the net portfolio position is `0`. Such a portfolio is said to be "dollar-neutral."

To set up a dollar-neutral portfolio, start with the "standard" portfolio problem and set the maximum exposure in long and short positions in the variable `Exposure`. The bounds for individual asset weights are plus or minus `Exposure`. Since the net position must be dollar-neutral, the budget constraint is `0` and the initial portfolio must be `0`. Finally, the one-way turnover constraints provide the necessary long and short restrictions to prevent "double-counting" of long and short positions. The blotter shows the portfolio weights for the dollar-neutral portfolio that maximizes the Sharpe ratio. The long and short positions are obtained from the buy and sell trades relative to the initial portfolio.

```
Exposure = 1;

q = setBounds(p, -Exposure, Exposure);
q = setBudget(q, 0, 0);
q = setOneWayTurnover(q, Exposure, Exposure, 0);

[qwgt, qlong, qshort] = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

[qswgt, qslong, qsshort] = estimateMaxSharpeRatio(q);
[qsrsk, qsret] = estimatePortMoments(q, qswgt);

% Plot efficient frontier for a dollar-neutral fund structure with tangency portfolio

clf;
portfolioexamples_plot('Efficient Frontier with Dollar-Neutral Portfolio', ...
    {'line', prsk, pret, {'Standard'}, 'b:'}, ...
    {'line', qrsk, qret, {'Dollar-Neutral'}, 'b'}, ...
    {'scatter', qsrsk, qsret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```




```
% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio

Blotter = dataset({100*qswgt(abs(qswgt) > 1.0e-4), 'Weight'}, ...
    {100*qslong(abs(qswgt) > 1.0e-4), 'Long'}, ...
    {100*qsshort(abs(qswgt) > 1.0e-4), 'Short'}, ...
    'obsnames', AssetList(abs(qswgt) > 1.0e-4));

displayPortfolio('Dollar-Neutral Portfolio with Maximum Sharpe Ratio', Blotter, true, '
```

```
Dollar-Neutral Portfolio with Maximum Sharpe Ratio
           Weight     Long       Short
    AA      0.53991    0.53991         0
    AIG      3.2253     3.2253         0
```

```
    AXP      0.98473    0.98473          0
    BA        -3.709          0      3.709
    C         14.859     14.859          0
    CAT       3.9539     3.9539          0
    DD       -19.168          0     19.168
    DIS      -5.1186          0     5.1186
    GE       -3.8391          0     3.8391
    GM       -3.9487          0     3.9487
    HD        1.1683     1.1683          0
    HON      -1.5227          0     1.5227
    HPQ      0.10515    0.10515          0
    IBM      -8.5514          0     8.5514
    INTC      1.8775     1.8775          0
    JNJ       1.4534     1.4534          0
    JPM      -2.6816          0     2.6816
    KO       -15.074          0     15.074
    MCD       4.1492     4.1492          0
    MMM       8.0643     8.0643          0
    MO        4.3354     4.3354          0
    MRK       3.9762     3.9762          0
    MSFT      4.3262     4.3262          0
    PFE      -9.6523          0     9.6523
    PG        1.7502     1.7502          0
    SBC      -5.5761          0     5.5761
    UTX       6.0968     6.0968          0
    VZ       -2.5871          0     2.5871
    WMT      0.90033    0.90033          0
    XOM       19.662     19.662          0

Confirm Dollar-Neutral Portfolio
  (Net, Long, Short)
0.0000 81.4282 81.4282
```

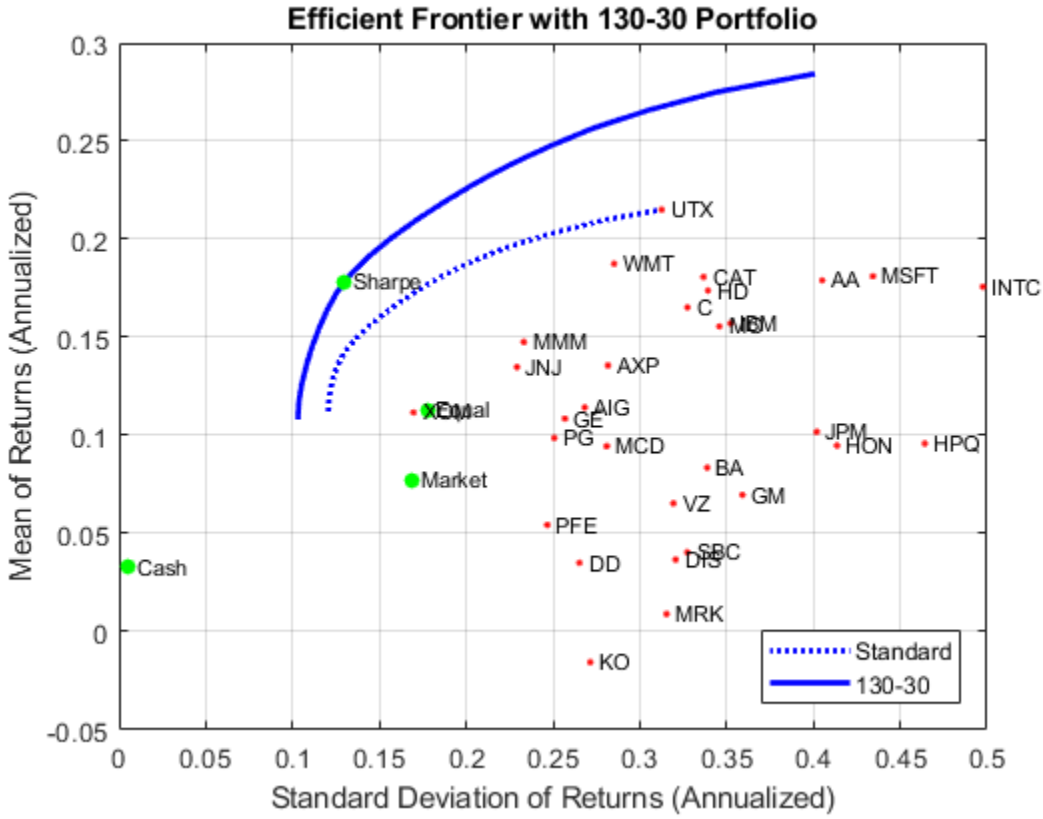
130/30 Fund Structure

Finally, the turnover constraints can be used to set up a 130-30 portfolio structure, which is a structure with a net long position but permits leverage with long and short positions up to a maximum amount of leverage. In the case of a 130-30 portfolio, the leverage is 30%.

To set up a 130-30 portfolio, start with the "standard" portfolio problem and set the maximum value for leverage in the variable `Leverage`. The bounds for individual asset weights range between `-Leverage` and `1 + Leverage`. Since the net position must be long, the budget constraint is `1` and, once again, the initial portfolio is `0`. Finally, the one-

way turnover constraints provide the necessary long and short restrictions to prevent "double-counting" of long and short positions. The blotter shows the portfolio weights for the 130-30 portfolio that maximizes the Sharpe ratio. The long and short positions are obtained from the buy and sell trades relative to the initial portfolio.

```
Leverage = 0.3;

q = setBounds(p, -Leverage, 1 + Leverage);
q = setBudget(q, 1, 1);
q = setOneWayTurnover(q, 1 + Leverage, Leverage);

[qwgt, qbuy, qsell] = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);

[qswgt, qslong, qsshort] = estimateMaxSharpeRatio(q);
[qsrsk, qsret] = estimatePortMoments(q, qswgt);

% Plot efficient frontier for a 130-30 fund structure with tangency portfolio

clf;
portfolioexamples_plot(sprintf('Efficient Frontier with %g-%g Portfolio', ...
    100*(1 + Leverage),100*Leverage), ...
    {'line', prsk, pret, {'Standard'}, 'b:'}, ...
    {'line', qrsk, qret, {'130-30'}, 'b'}, ...
    {'scatter', qsrsk, qsret, {'Sharpe'}}, ...
    {'scatter', [mrsk, crsk, ersk], [mret, cret, eret], {'Market', 'Cash', 'Equal'}}, .
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```




```
% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio

Blotter = dataset({100*qswgt(abs(qswgt) > 1.0e-4), 'Weight'}, ...
    {100*qslong(abs(qswgt) > 1.0e-4), 'Long'}, ...
    {100*qsshort(abs(qswgt) > 1.0e-4), 'Short'}, ...
    'obsnames', AssetList(abs(qswgt) > 1.0e-4));

displayPortfolio(sprintf('%g-%g Portfolio with Maximum Sharpe Ratio', 100*(1 + Leverage

130-30 Portfolio with Maximum Sharpe Ratio
            Weight      Long      Short
    DD       -9.5565         0     9.5565
    HON      -6.0245         0     6.0245
```

```
    INTC      4.0335    4.0335          0
    JNJ       7.1234    7.1234          0
    JPM      -0.44583         0    0.44583
    KO        -13.646         0     13.646
    MMM        20.908    20.908          0
    MO         14.433    14.433          0
    MSFT       4.5592    4.5592          0
    PG         17.243    17.243          0
    SBC      -0.32712         0    0.32712
    UTX        5.3584    5.3584          0
    WMT        21.018    21.018          0
    XOM        35.323    35.323          0

Confirm 130-30 Portfolio
  (Net, Long, Short)
100.0000 130.0000 30.0000
```

References

1. R. C. Grinold and R. N. Kahn (2000), *Active Portfolio Management*, 2nd ed.
2. H. M. Markowitz (1952), "Portfolio Selection," *Journal of Finance*, Vol. 1, No. 1, pp. 77-91.
3. J. Lintner (1965), "The Valuation of Risk Assets and the Selection of Risky Investments in Stock Portfolios and Capital Budgets," *Review of Economics and Statistics*, Vol. 47, No. 1, pp. 13-37.
4. H. M. Markowitz (1959), *Portfolio Selection: Efficient Diversification of Investments*, John Wiley & Sons, Inc.
5. W. F. Sharpe (1966), "Mutual Fund Performance," *Journal of Business*, Vol. 39, No. 1, Part 2, pp. 119-138.
6. J. Tobin (1958), "Liquidity Preference as Behavior Towards Risk," *Review of Economic Studies*, Vol. 25, No.1, pp. 65-86.
7. J. L. Treynor and F. Black (1973), "How to Use Security Analysis to Improve Portfolio Selection," *Journal of Business*, Vol. 46, No. 1, pp. 68-86.

Utility Functions

```
function displaySumOfTransactions(Turnover, qbuy, qsell)
fprintf('Sum of Purchases by Portfolio along Efficient Frontier (Max. Turnover %g%%)\n'
    100*Turnover);
fprintf('%.4f ', 100*sum(qbuy)), sprintf('\n\n');
fprintf('\n')
```

```
fprintf('Sum of Sales by Portfolio along Efficient Frontier (Max. Turnover %g%%)\n', ..
    100*Turnover);
fprintf('%.4f ', 100*sum(qsell));
end

function displayPortfolio(Description, Blotter, LongShortFlag, portfolioType)
fprintf('%s\n', Description);
disp(Blotter);
if (LongShortFlag)
    fprintf('Confirm %s Portfolio\n', portfolioType);
    fprintf('  (Net, Long, Short)\n');
    fprintf('%.4f ' , [ sum(Blotter.Weight), sum(Blotter.Long), sum(Blotter.Short) ]);
end
end
```

See Also

`Portfolio` | `addGroups` | `estimateAssetMoments` | `estimateBounds` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortRisk` | `plotFrontier` | `setAssetMoments` | `setBounds`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-138

More About

- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)

- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Asset Allocation Case Study

This example shows how to set up a basic asset allocation problem that uses mean-variance portfolio optimization to estimate efficient portfolios.

Step 1. Defining the portfolio problem.

Suppose that you want to manage an asset allocation fund with four asset classes: bonds, large-cap equities, small-cap equities, and emerging equities. The fund is long-only with no borrowing or leverage, should have no more than 85% of the portfolio in equities, and no more than 35% of the portfolio in emerging equities. The cost to trade the first three assets is 10 basis points annualized and the cost to trade emerging equities is four times higher. Finally, you want to ensure that average turnover is no more than 15%. To solve this problem, you will set up a basic mean-variance portfolio optimization problem and then slowly introduce the various constraints on the problem to get to a solution.

To set up the portfolio optimization problem, start with basic definitions of known quantities associated with the structure of this problem. Each asset class is assumed to have a tradeable asset with a real-time price. Such assets can be, for example, exchange-traded funds (ETFs). The initial portfolio with holdings in each asset that has a total of $7.5 million along with an additional cash position of $60,000. These basic quantities and the costs to trade are set up in the following variables with asset names in the cell array `Asset`, current prices in the vector `Price` , current portfolio holdings in the vector `Holding`, and transaction costs in the vector `UnitCost`.

To analyze this portfolio, you can set up a blotter in a `dataset` object to help track prices, holdings, weights, and so forth. In particular, you can compute the initial portfolio weights and maintain them in a new blotter field called `InitPort`.

```
Asset = { 'Bonds', 'Large-Cap Equities', 'Small-Cap Equities', 'Emerging Equities' };
Price = [ 52.4; 122.7; 35.2; 46.9 ];
Holding = [ 42938; 24449; 42612; 15991 ];
UnitCost = [ 0.001; 0.001; 0.001; 0.004 ];

Blotter = dataset({Price, 'Price'}, {Holding, 'InitHolding'},'obsnames',Asset);
Wealth = sum(Blotter.Price .* Blotter.InitHolding);
Blotter.InitPort = (1/Wealth)*(Blotter.Price .* Blotter.InitHolding);
Blotter.UnitCost = UnitCost;
disp(Blotter);
```

```
                         Price    InitHolding    InitPort    UnitCost
    Bonds                 52.4    42938          0.3         0.001
```

```
    Large-Cap Equities    122.7    24449          0.4          0.001
    Small-Cap Equities     35.2    42612          0.2          0.001
    Emerging Equities      46.9    15991          0.1          0.004
```

Step 2. Simulating asset prices.

Since this is a hypothetical example, to simulate asset prices from a given mean and covariance of annual asset total returns for the asset classes, the `portsim` function is used to create asset returns with the desired mean and covariance. Specifically, `portsim` is used to simulate five years of monthly total returns and then plotted to show the log of the simulated total return prices

The mean and covariance of annual asset total returns are maintained in the variables `AssetMean` and `AssetCovar`. The simulated asset total return prices (which are compounded total returns) are maintained in the variable `Y`. All initial asset total return prices are normalized to `1` in this example.

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

X = portsim(AssetMean'/12, AssetCovar/12, 60); % monthly total returns for 5 years (60
[Y, T] = ret2tick(X, [], 1/12);                % form total return prices

plot(T, log(Y));
title('\bfSimulated Asset Class Total Return Prices');
xlabel('Year');
ylabel('Log Total Return Price');
legend(Asset,'Location','best');
```

Step 3. Setting up the Portfolio object.

To explore portfolios on the efficient frontier, set up a Portfolio object using these specifications:

- Portfolio weights are nonnegative and sum to `1`.
- Equity allocation is no more than 85% of the portfolio.
- Emerging equity is no more than 35% of the portfolio.

These specifications are incorporated into the Portfolio object `p` in the following sequence of using functions that starts with using the `Portfolio` function.

1. The specification of the initial portfolio from `Blotter` gives the number of assets in your universe so you do not need to specify the `NumAssets` property directly. Next, set up default constraints (long-only with a budget constraint). In addition, set up the group constraint that imposes an upper bound on equities in the portfolio (equities are identified in the group matrix with `1`'s) and the upper bound constraint on emerging equities. Although you could have set the upper bound on emerging equities using the `setBounds` function, notice how the `addGroups` function is used to set up this constraint.
2. To have a fully specified mean-variance portfolio optimization problem, you must specify the mean and covariance of asset returns. Since starting with these moments in the variables `AssetMean` and `AssetCovar`, you can use the `setAssetMoments` function to enter these variables into your Portfolio object (remember that you are assuming that your raw data are monthly returns which is why you divide your annual input moments by 12 to get monthly returns).
3. Use the total return prices with the `estimateAssetMoments` function with a specification that your data in `Y` are prices, and not returns, to estimate asset return moments for your Portfolio object.
4. Although the returns in your Portfolio object are in units of monthly returns, and since subsequent costs are annualized, it is convenient to specify them as annualized total returns with this direct transformation of the `AssetMean` and `AssetCovar` properties of your Portfolio object `p`.
5. Display the Portfolio object `p`.

```
p = Portfolio('Name', 'Asset Allocation Portfolio', ...
'AssetList', Asset, 'InitPort', Blotter.InitPort);

p = setDefaultConstraints(p);
p = setGroups(p, [ 0, 1, 1, 1 ], [], 0.85);
p = addGroups(p, [ 0, 0, 0, 1 ], [], 0.35);

p = setAssetMoments(p, AssetMean/12, AssetCovar/12);
p = estimateAssetMoments(p, Y, 'DataFormat', 'Prices');

p.AssetMean = 12*p.AssetMean;
p.AssetCovar = 12*p.AssetCovar;

display(p);

p =
  Portfolio with properties:
```

```
         BuyCost: []
        SellCost: []
    RiskFreeRate: []
       AssetMean: [4x1 double]
      AssetCovar: [4x4 double]
   TrackingError: []
    TrackingPort: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
            Name: 'Asset Allocation Portfolio'
       NumAssets: 4
       AssetList: {1x4 cell}
        InitPort: [4x1 double]
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: [2x4 double]
      LowerGroup: []
      UpperGroup: [2x1 double]
          GroupA: []
          GroupB: []
     LowerRatio: []
     UpperRatio: []
```

Step 4. Validate the portfolio problem.

An important step in portfolio optimization is to validate that the portfolio problem is feasible and the main test is to ensure that the set of portfolios is nonempty and bounded. Use the `estimateBounds` function to determine the bounds for the portfolio set. In this case, since both `lb` and `ub` are finite, the set is bounded.

```
[lb, ub] = estimateBounds(p);
display([lb, ub]);
```

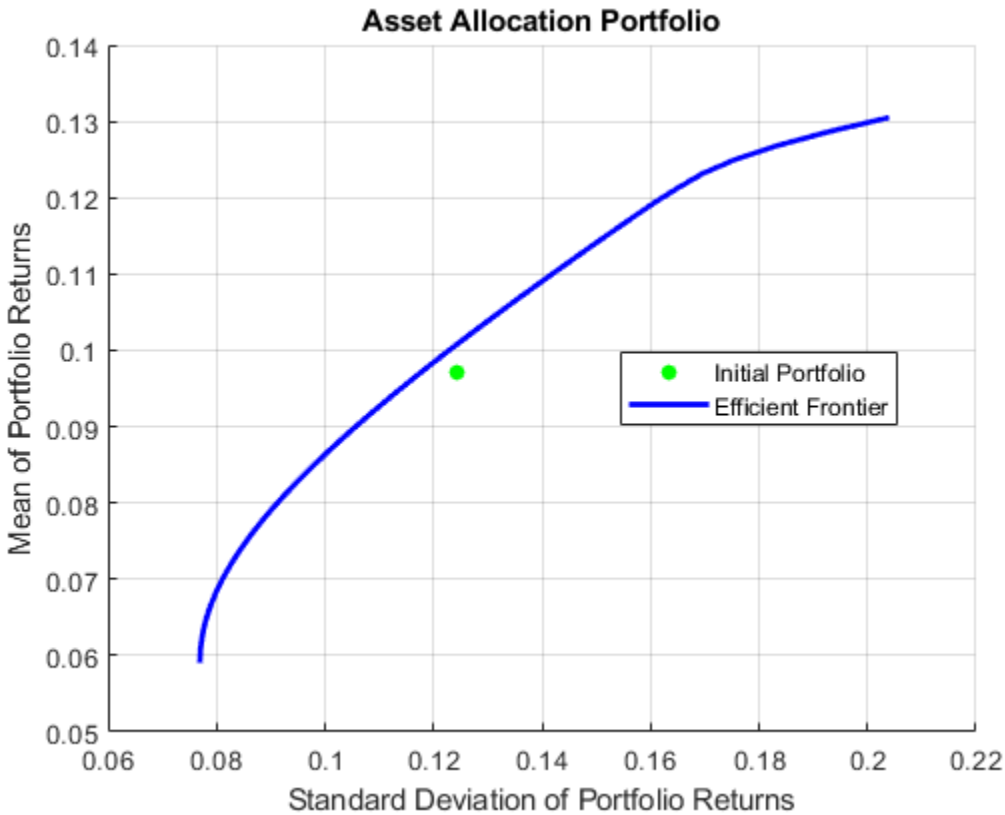
```
    0.1500    1.0000
    0.0000    0.8500
```

```
    0.0000    0.8500
    0.0000    0.3500
```

Step 5. Plotting the efficient frontier.

Given the constructed Portfolio object, use the `plotFrontier` function to view the efficient frontier. Instead of using the default of 10 portfolios along the frontier, you can display the frontier with 40 portfolios. Notice gross efficient portfolio returns fall between approximately 6% and 16% per years.

```
plotFrontier(p, 40);
```



Step 6. Evaluating gross vs. net portfolio returns.

The Portfolio object `p` does not include transaction costs so that the portfolio optimization problem specified in `p` uses gross portfolio return as the return proxy. To handle net returns, create a second Portfolio object `q` that includes transaction costs.

```
q = setCosts(p, UnitCost, UnitCost);
display(q);
```

```
q =
  Portfolio with properties:

          BuyCost: [4x1 double]
         SellCost: [4x1 double]
     RiskFreeRate: []
        AssetMean: [4x1 double]
       AssetCovar: [4x4 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: 'Asset Allocation Portfolio'
        NumAssets: 4
        AssetList: {1x4 cell}
         InitPort: [4x1 double]
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: [2x4 double]
       LowerGroup: []
       UpperGroup: [2x1 double]
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
```

Step 7. Analyzing descriptive properties of the Portfolio structures.

To be more concrete about the ranges of efficient portfolio returns and risks, use the `estimateFrontierLimits` function to obtain portfolios at the endpoints of the efficient frontier. Given these portfolios, compute their moments using the `estimatePortMoments` function. The following code generates a table that lists the risk and return of the initial portfolio as well as the gross and net moments of portfolio returns for the portfolios at the endpoints of the efficient frontier:

```
[prsk0, pret0] = estimatePortMoments(p, p.InitPort);

pret = estimatePortReturn(p, p.estimateFrontierLimits);
qret = estimatePortReturn(q, q.estimateFrontierLimits);

displayReturns(pret0, pret, qret);

Annualized Portfolio Returns ...
                                    Gross       Net
Initial Portfolio Return             9.70 %    9.70 %
Minimum Efficient Portfolio Return   5.90 %    5.77 %
Maximum Efficient Portfolio Return  13.05 %   12.86 %
```

The results shows that the cost to trade ranges from 14 to 19 basis points to get from the current portfolio to the efficient portfolios at the endpoints of the efficient frontier (these costs are the difference between gross and net portfolio returns.) In addition, notice that the maximum efficient portfolio return (13%) is less than the maximum asset return (18%) due to the constraints on equity allocations.

Step 8. Obtaining a Portfolio at the specified return level on the efficient frontier.

A common approach to select efficient portfolios is to pick a portfolio that has a desired fraction of the range of expected portfolio returns. To obtain the portfolio that is 30% of the range from the minimum to maximum return on the efficient frontier, obtain the range of net returns in `qret` using the Portfolio object `q` and interpolate to obtain a 30% level with the `interp1` function to obtain a portfolio `qwgt`.

```
Level = 0.3;

qret = estimatePortReturn(q, q.estimateFrontierLimits);
qwgt = estimateFrontierByReturn(q, interp1([0, 1], qret, Level));
[qrsk, qret] = estimatePortMoments(q, qwgt);

displayReturnLevel(Level, qret, qrsk);
```

```
Portfolio at 30% return level on efficient frontier ...
    Return       Risk
      7.90       9.09


display(qwgt);

qwgt =

    0.6252
    0.1856
    0.0695
    0.1198
```

The target portfolio that is 30% of the range from minimum to maximum net returns has a return of 7.9% and a risk of 9.1%.

Step 9. Obtaining a Portfolio at the specified risk levels on the efficient frontier.

Although you could accept this result, suppose that you want to target values for portfolio risk. Specifically, suppose that you have a conservative target risk of 10%, a moderate target risk of 15%, and an aggressive target risk of 20% and you want to obtain portfolios that satisfy each risk target. Use the `estimateFrontierByRisk` function to obtain targeted risks specified in the variable `TargetRisk`. The resultant three efficient portfolios are obtained in `qwgt`.

```
TargetRisk = [ 0.10; 0.15; 0.20 ];
qwgt = estimateFrontierByRisk(q, TargetRisk);
display(qwgt);

qwgt =

    0.5407    0.2020    0.1500
    0.2332    0.4000    0.0318
    0.0788    0.1280    0.4682
    0.1474    0.2700    0.3500
```

Use the `estimatePortRisk` function to compute the portfolio risks for the three portfolios to confirm that the target risks have been attained:

```
display(estimatePortRisk(q, qwgt));
```

```
    0.1000
    0.1500
    0.2000
```

Suppose that you want to shift from the current portfolio to the moderate portfolio. You can estimate the purchases and sales to get to this portfolio:

```
[qwgt, qbuy, qsell] = estimateFrontierByRisk(q, 0.15);
```

If you average the purchases and sales for this portfolio, you can see that the average turnover is 17%, which is greater than the target of 15%:

```
disp(sum(qbuy + qsell)/2)
```

```
    0.1700
```

Since you also want to ensure that average turnover is no more than 15%, you can add the average turnover constraint to the `Portfolio` object:

```
q = setTurnover(q, 0.15);
[qwgt, qbuy, qsell] = estimateFrontierByRisk(q, 0.15);
```

You can enter the estimated efficient portfolio with purchases and sales into the Blotter:

```
qbuy(abs(qbuy) < 1.0e-5) = 0;
qsell(abs(qsell) < 1.0e-5) = 0;  % zero out near 0 trade weights

Blotter.Port = qwgt;
Blotter.Buy = qbuy;
Blotter.Sell = qsell;

display(Blotter);
```

```
Blotter =
                         Price    InitHolding    InitPort    UnitCost
    Bonds                 52.4    42938          0.3         0.001
    Large-Cap Equities   122.7    24449          0.4         0.001
    Small-Cap Equities    35.2    42612          0.2         0.001
    Emerging Equities     46.9    15991          0.1         0.004


                         Port       Buy     Sell
```

```
    Bonds                 0.18787       0     0.11213
    Large-Cap Equities        0.4       0           0
    Small-Cap Equities    0.16213       0    0.037871
    Emerging Equities        0.25    0.15           0
```

The `Buy` and `Sell` elements of the Blotter are changes in portfolio weights that must be converted into changes in portfolio holdings to determine the trades. Since you are working with net portfolio returns, you must first compute the cost to trade from your initial portfolio to the new portfolio. This can be accomplished as follows:

```
TotalCost = Wealth * sum(Blotter.UnitCost .* (Blotter.Buy + Blotter.Sell))
```

```
TotalCost = 5.6248e+03
```

The cost to trade is $5,625, so that, in general, you would have to adjust your initial wealth accordingly before setting up your new portfolio weights. However, to keep the analysis simple, note that you have sufficient cash ($60,0000) set aside to pay the trading costs and that you will not touch the cash position to build up any positions in your portfolio. Thus, you can populate your blotter with the new portfolio holdings and the trades to get to the new portfolio without making any changes in your total invested wealth. First, compute portfolio holding:

```
Blotter.Holding = Wealth * (Blotter.Port ./ Blotter.Price);
```

Compute number of shares to `Buy` and `Sell` in your Blotter:

```
Blotter.BuyShare = Wealth * (Blotter.Buy ./ Blotter.Price);
Blotter.SellShare = Wealth * (Blotter.Sell ./ Blotter.Price);
```

Notice how you used an add-hoc truncation rule to obtain unit numbers of shares to buy and sell. Clean up the blotter by removing the unit costs and the buy and sell portfolio weights:

```
Blotter.Buy = [];
Blotter.Sell = [];
Blotter.UnitCost = [];
```

Step 10. Displaying the final results.

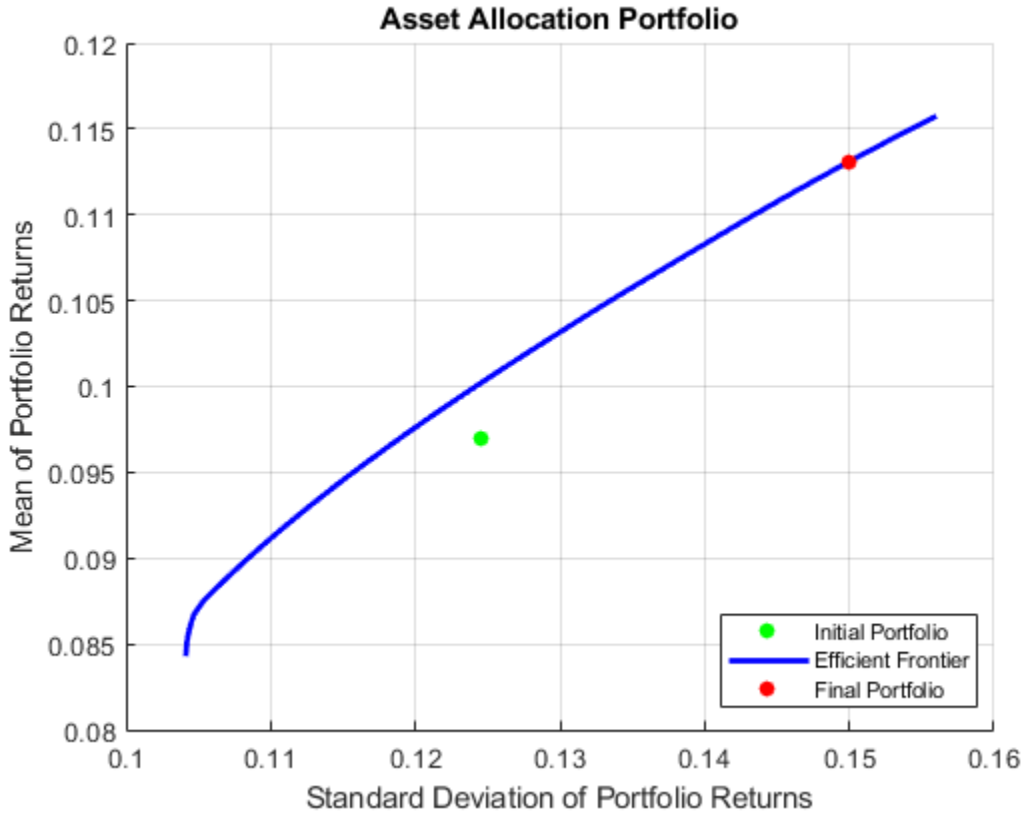
The final result is a blotter that contains proposed trades to get from your current portfolio to a moderate-risk portfolio. To make the trade, you would need to sell 16,049 shares of your bond asset and 8,069 shares of your small-cap equity asset and would need to purchase 23,986 shares of your emerging equities asset.

```
display(Blotter);
```

```
Blotter =
                          Price    InitHolding    InitPort    Port       Holding
    Bonds                  52.4    42938          0.3         0.18787    26889
    Large-Cap Equities    122.7    24449          0.4             0.4    24449
    Small-Cap Equities     35.2    42612          0.2         0.16213    34543
    Emerging Equities      46.9    15991          0.1            0.25    39977


                          BuyShare    SellShare
    Bonds                        0        16049
    Large-Cap Equities           0            0
    Small-Cap Equities           0       8068.8
    Emerging Equities        23986            0
```

The final plot uses the plotFrontier function to display the efficient frontier and the initial portfolio for the fully specified portfolio optimization problem. It also adds the location of the moderate-risk or final portfolio on the efficient frontier.

```
plotFrontier(q, 40);
hold on
scatter(estimatePortRisk(q, qwgt), estimatePortReturn(q, qwgt), 'filled', 'r');
h = legend('Initial Portfolio', 'Efficient Frontier', 'Final Portfolio', 'location', 'b
set(h, 'Fontsize', 8);
hold off
```



Utility Functions

```
function displayReturns(pret0, pret, qret)
fprintf('Annualized Portfolio Returns ...\n');
fprintf('                                    %6s    %6s\n','Gross','Net');
fprintf('Initial Portfolio Return            %6.2f %%  %6.2f %%\n',100*pret0,100*pret0);
fprintf('Minimum Efficient Portfolio Return %6.2f %%  %6.2f %%\n',100*pret(1),100*qret(
fprintf('Maximum Efficient Portfolio Return %6.2f %%  %6.2f %%\n',100*pret(2),100*qret(
end

function displayReturnLevel(Level, qret, qrsk)
fprintf('Portfolio at %g%% return level on efficient frontier ...\n',100*Level);
fprintf('%10s %10s\n','Return','Risk');
```

```
fprintf('%10.2f %10.2f\n',100*qret,100*qrsk);
end
```

See Also

`Portfolio` | `addGroups` | `estimateAssetMoments` | `estimateBounds` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortRisk` | `plotFrontier` | `setAssetMoments` | `setBounds`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-138

More About

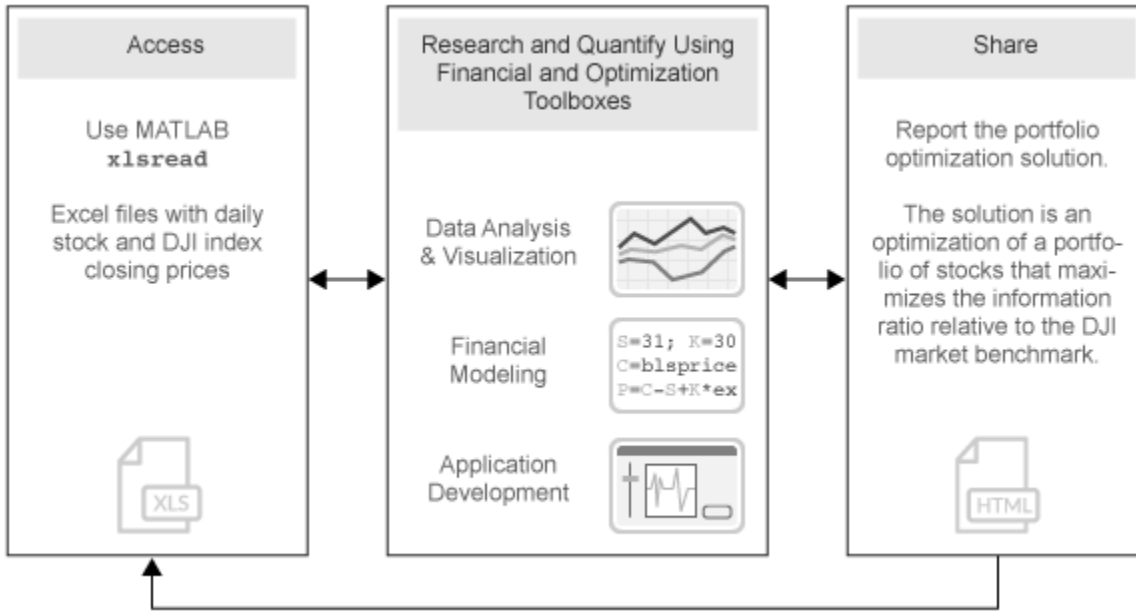
- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Optimization Against a Benchmark

This example shows how to perform portfolio optimization using the `Portfolio` object in Financial Toolbox™. The example, in particular, demonstrates optimizing a portfolio to maximize the information ratio relative to a market benchmark.



Import historical data using MATLAB®

Import historical prices for the asset universe and the Dow Jones Industrial Average (DJI) market benchmark. The data is imported into a table from a Microsoft® Excel® spreadsheet using the MATLAB® `readtable` function.

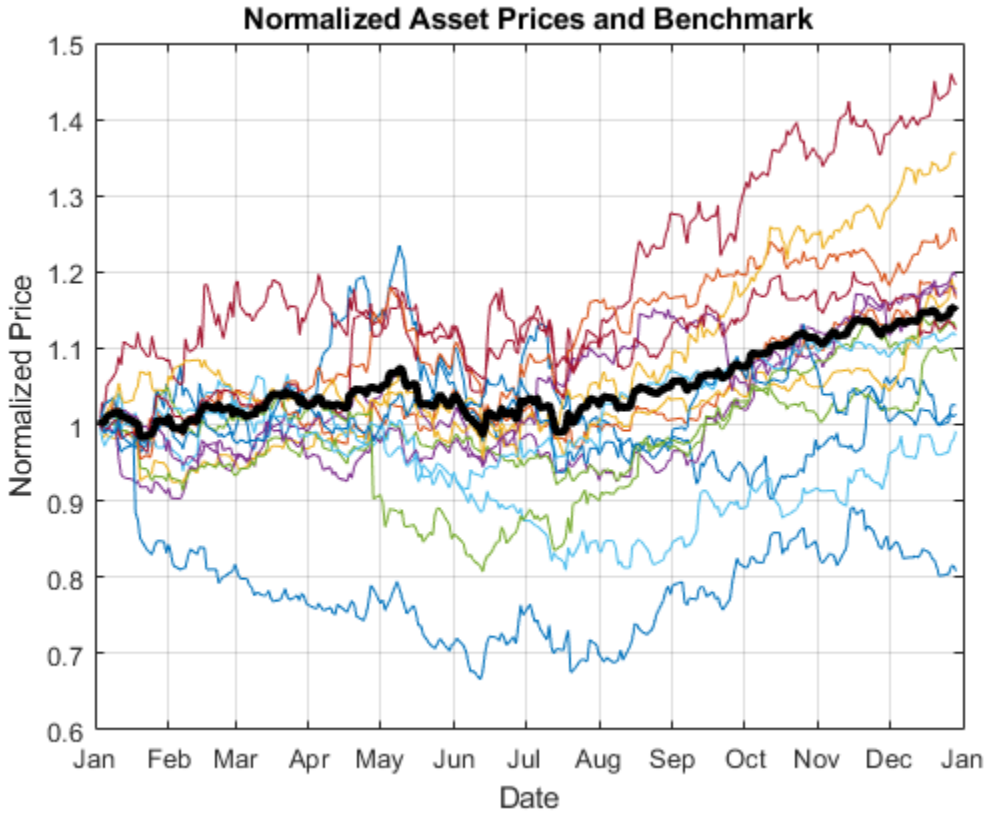
```
data = readtable('dowPortfolio.xlsx');
disp(data(1:10, :));
```

Dates	DJI	AA	AIG	AXP	BA	C	CAT	DD
1/3/2006	10847	28.72	68.41	51.53	68.63	45.26	55.86	40.68
1/4/2006	10880	28.89	68.51	51.03	69.34	44.42	57.29	40.46

```
    1/5/2006     10882    29.12     68.6    51.57    68.53    44.65    57.29    40.38
    1/6/2006     10959    29.02    68.89    51.75    67.57    44.65    58.43    40.55
    1/9/2006     11012    29.37    68.57    53.04    67.01    44.43    59.49    40.32
   1/10/2006     11012    28.44    69.18    52.88    67.33    44.57    59.25     40.2
   1/11/2006     11043    28.05     69.6    52.59     68.3    44.98    59.28    38.87
   1/12/2006     10962    27.68    69.04     52.6     67.9    45.02    60.13    38.02
   1/13/2006     10960    27.81    68.84     52.5     67.7    44.92    60.24    37.86
   1/17/2006     10896    27.97    67.84    52.03    66.93    44.47    60.85    37.75
```

Separate the asset names, asset prices, and DJI benchmark prices from the dataset array. The visualization shows the evolution of all the asset prices normalized to start at unity.

```
dates = datenum(data.Dates);
benchPrice = data.DJI;
assetNames = data.Properties.VariableNames(3:2:end);
assetPrice = table2array(data(:, assetNames) );

assetP = assetPrice./assetPrice(1, :);
benchmarkP = benchPrice / benchPrice(1);

figure;
plot(dates,assetP);
hold on;
plot(dates,benchmarkP,'LineWidth',3,'Color','k');
hold off;
xlabel('Date');
ylabel('Normalized Price');
title('Normalized Asset Prices and Benchmark');
datetick('x');
grid on;
```



The bold line indicates the DJIA market benchmark.

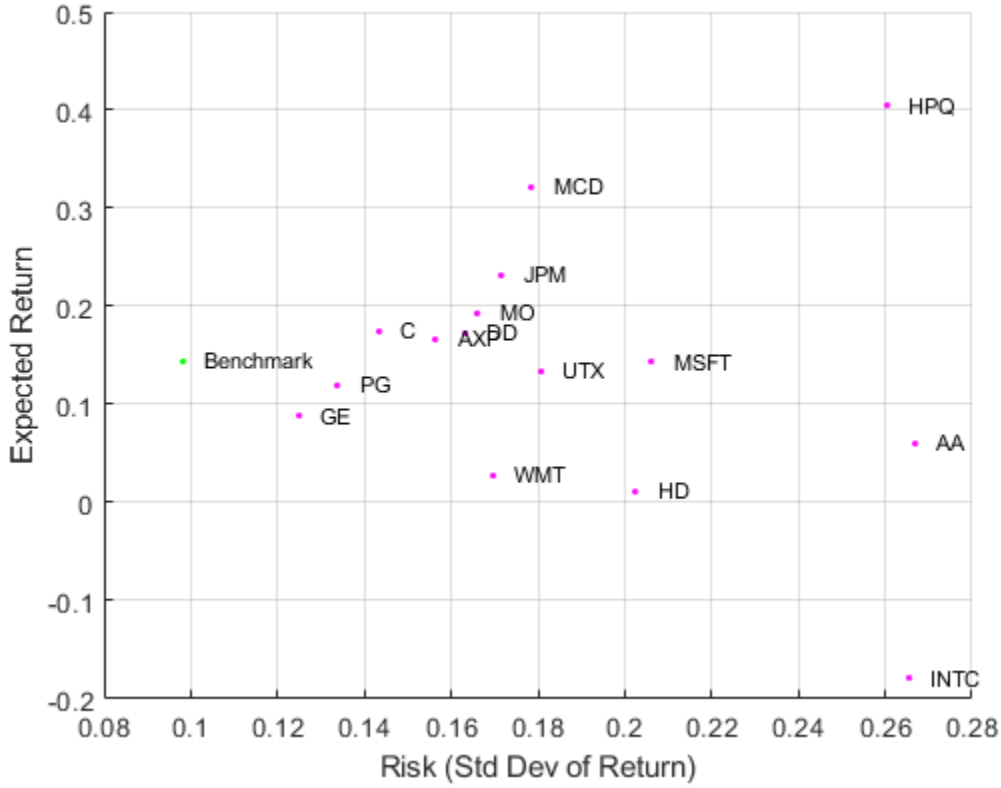
Compute returns and risk-adjusted returns

Calculate the return series from the price series and compute asset moments (historical returns and standard deviations). The visualization shows a scatter plot of the risk-return characteristics of all the assets and the DJI market benchmark.

```
benchReturn = tick2ret(benchPrice);
assetReturn = tick2ret(assetPrice);
activReturn = assetReturn - benchReturn;
```

Calculate historical statistics and plot the risk returns.

```
benchRetn = mean(benchReturn);
benchRisk =  std(benchReturn);
assetRetn = mean(assetReturn);
assetRisk =  std(assetReturn);
scale = 252;

assetRiskR = sqrt(scale) * assetRisk;
benchRiskR = sqrt(scale) * benchRisk;
assetReturnR = scale * assetRetn;
benchReturnR = scale * benchRetn;

figure;
scatter(assetRiskR, assetReturnR, 6, 'm', 'Filled');
hold on
scatter(benchRiskR, benchReturnR, 6, 'g', 'Filled');
for k = 1:length(assetNames)
    text(assetRiskR(k) + 0.005, assetReturnR(k), assetNames{k}, 'FontSize', 8);
end
text(benchRiskR + 0.005, benchReturnR, 'Benchmark', 'Fontsize', 8);
hold off;

xlabel('Risk (Std Dev of Return)');
ylabel('Expected Return');
grid on;
```



Set up a portfolio optimization

Set up a portfolio optimization problem by populating the object. In this example, the expected returns and covariances of the assets in the portfolio are set to their historical values..

```
p = Portfolio('AssetList',assetNames)
```

```
p =
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
```

```
       AssetMean: []
      AssetCovar: []
   TrackingError: []
    TrackingPort: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
            Name: []
       NumAssets: 15
       AssetList: {1x15 cell}
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: []
      UpperBound: []
     LowerBudget: []
     UpperBudget: []
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Set up default portfolio constraints (all weights sum to `1`, no shorting, and 100% investment in risky assets).

```
p = setDefaultConstraints(p)
```

```
p =
  Portfolio with properties:

          BuyCost: []
         SellCost: []
    RiskFreeRate: []
       AssetMean: []
      AssetCovar: []
   TrackingError: []
    TrackingPort: []
        Turnover: []
     BuyTurnover: []
```

```
    SellTurnover: []
            Name: []
       NumAssets: 15
       AssetList: {1x15 cell}
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [15x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Add asset returns and covariance to the `Portfolio` object.

```
pAct = estimateAssetMoments(p,activReturn,'missingdata',false)
```

```
pAct =
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [15x1 double]
       AssetCovar: [15x15 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 15
        AssetList: {1x15 cell}
         InitPort: []
      AInequality: []
      bInequality: []
```

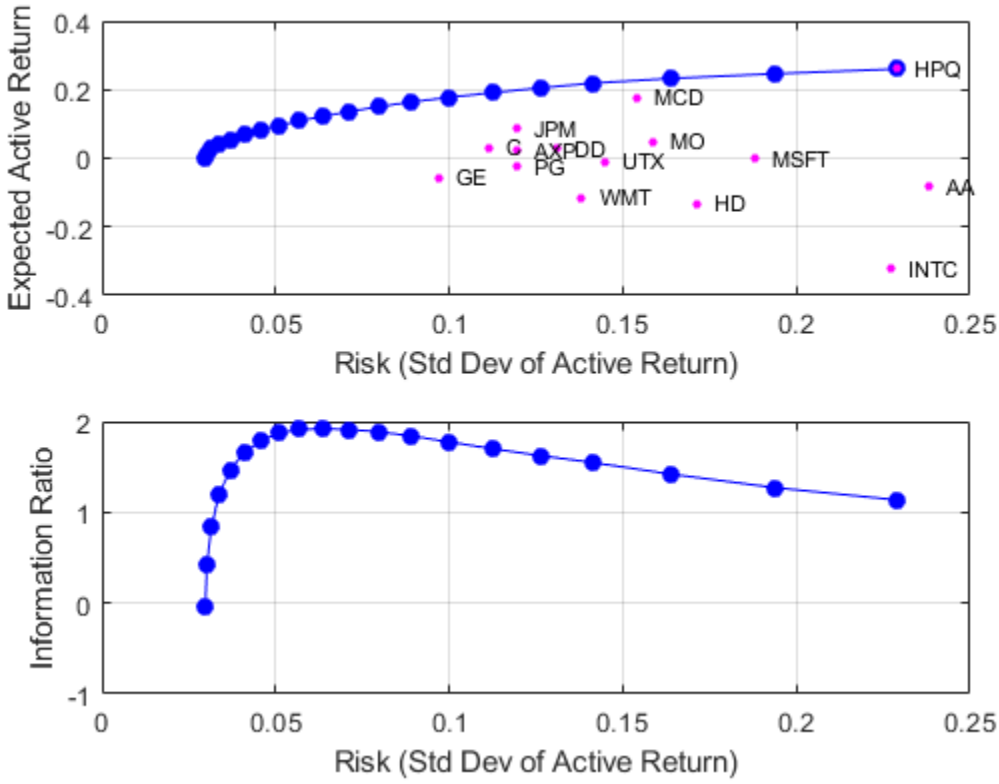
```
        AEquality: []
        bEquality: []
       LowerBound: [15x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
```

Compute the efficient frontier using the `Portfolio` object

Compute the mean-variance efficient frontier of 20 optimal portfolios. Visualize the frontier over the risk-return characteristics of the individual assets. Furthermore, calculate and visualize the information ratio for each portfolio along the frontier.

```
wAct = estimateFrontier(pAct, 20); % Estimate weights
[portRiskAct, portRetnAct] = estimatePortMoments(pAct, wAct); % Get risk and return

if isa(pAct,'Portfolio')
    % Extract asset moments & names
    [assetReturnP, assetCovarP] = getAssetMoments(pAct);
    assetRiskP = sqrt(diag(assetCovarP));
    assetNames = pAct.AssetList;
else
    assetNames = pAct;
end

% Rescale
assetRiskT = sqrt(scale) * assetRiskP;
portRiskT  = sqrt(scale) *  portRiskAct;
assetReturnT = scale * assetReturnP;
portReturnT = scale *  portRetnAct;

figure;
subplot(2,1,1);
plot(portRiskT, portReturnT, 'bo-', 'MarkerFaceColor', 'b');
hold on;

scatter(assetRiskT, assetReturnT, 12, 'm', 'Filled');
```

```
hold on;
for k = 1:length(assetNames)
    text(assetRiskT(k) + 0.005, assetReturnT(k), assetNames{k}, 'FontSize', 8);
end

hold off;

xlabel('Risk (Std Dev of Active Return)');
ylabel('Expected Active Return');
grid on;

subplot(2,1,2);
plot(portRiskT, portReturnT./portRiskT, 'bo-', 'MarkerFaceColor', 'b');
xlabel('Risk (Std Dev of Active Return)');
ylabel('Information Ratio');
grid on;
```



Perform information ratio maximization using Optimization Toolbox™

Run a hybrid optimization to find the portfolio along the frontier with the maximum information ratio. The information ratio is the ratio of relative return to relative risk (known as "tracking error"). Whereas the Sharpe ratio looks at returns relative to a riskless asset, the information ratio is based on returns relative to a risky benchmark, in this case the DJI benchmark. This is done by running an optimization that finds the optimal return constraint for which the portfolio optimization problem returns the maximum information ratio portfolio. The portfolio optimization functions are called from an objective function `infoRatioTargetReturn` that is optimized by the Optimization Toolbox™ function `fminbnd`. The utility function `infoRatioTargetReturn` calculates a minimum (active) risk portfolio given a target active return.

The `infoRatioTargetReturn` utility function is called as an objective function in an optimization routine (`fminbnd`) that seeks to find the target return that maximizes the information ratio and minimizes a negative information ratio.

```
objFun = @(targetReturn) -infoRatioTargetReturn(targetReturn,pAct);
options = optimset('TolX',1.0e-8);
[optPortRetn, ~, exitflag] = fminbnd(objFun,0,max(portRetnAct),options);
```

Get weights, information ratio, and risk return for the optimal portfolio.

```
[optInfoRatio,optWts] = infoRatioTargetReturn(optPortRetn,pAct);
optPortRisk = estimatePortRisk(pAct,optWts)
```

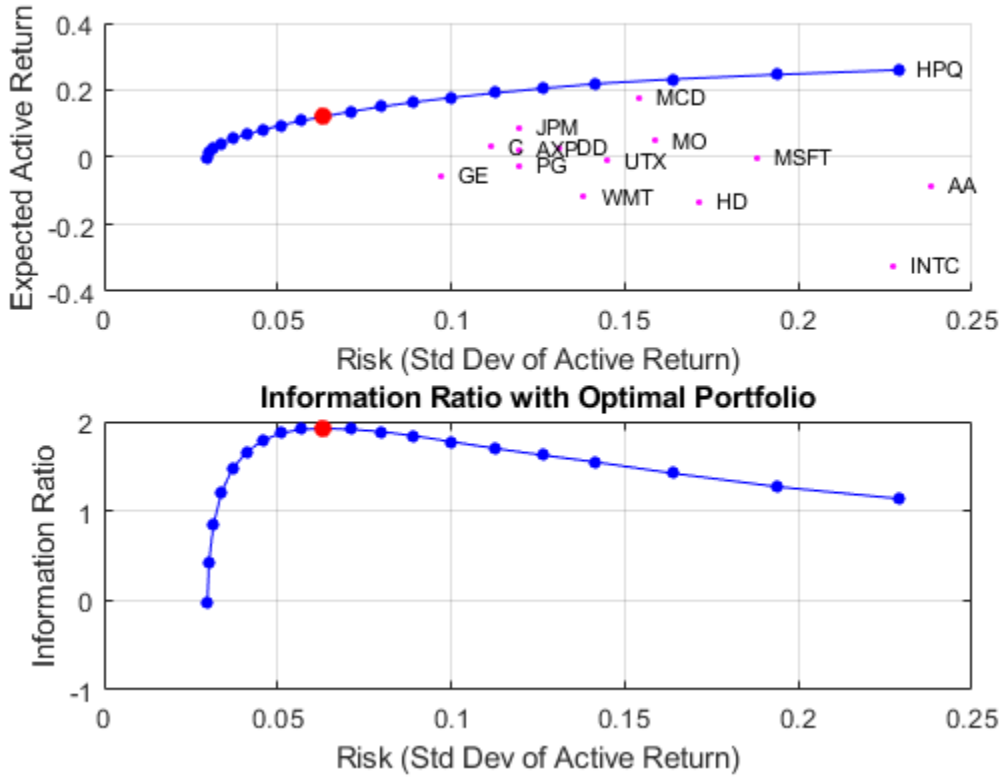
```
optPortRisk = 0.0040
```

Plot the optimal portfolio

Verify that the portfolio found is indeed the maximum information-ratio portfolio.

```
if isa(pAct,'Portfolio')
    % Extract asset moments & names
    [assetReturnQ,assetCovarQ] = getAssetMoments(pAct);
    assetRiskQ = sqrt(diag(assetCovarQ));
    assetNamesQ = pAct.AssetList;
else
    assetNamesQ = pAct;
end

% Rescale
  assetRiskS = sqrt(scale) * assetRiskQ;
   portRiskS = sqrt(scale) *  portRiskAct;
optPortRiskS = sqrt(scale) * optPortRisk;
  assetReturnS = scale * assetReturnQ;
   portReturnS = scale *  portRetnAct;
optPortReturnS = scale * optPortRetn;

figure;
subplot(2,1,1);

scatter(assetRiskS, assetReturnS, 6, 'm', 'Filled');
hold on
for k = 1:length(assetNames)
    text(assetRiskS(k) + 0.005,assetReturnS(k),assetNames{k},'FontSize',8);
end
plot(portRiskS,portReturnS,'bo-','MarkerSize',4,'MarkerFaceColor','b');
```

```
plot(optPortRiskS,optPortReturnS,'ro-','MarkerFaceColor','r');
hold off;

xlabel('Risk (Std Dev of Active Return)');
ylabel('Expected Active Return');
grid on;

subplot(2,1,2);
plot(portRiskS,portReturnS./portRiskS,'bo-','MarkerSize',4,'MarkerFaceColor','b');
hold on
plot(optPortRiskS,optPortReturnS./optPortRiskS,'ro-','MarkerFaceColor','r');
hold off;

xlabel('Risk (Std Dev of Active Return)');
ylabel('Information Ratio');
title('Information Ratio with Optimal Portfolio');
grid on;
```



Display the portfolio optimization solution

Display the portfolio optimization solution.

```
assetIndx = optWts > .001;
results = table(assetNames(assetIndx)', optWts(assetIndx)*100, 'VariableNames',{'Asset'
disp('Maximum Information Ratio Portfolio:');
```

```
Maximum Information Ratio Portfolio:
```

```
disp(results);
```

```
    Asset      Weight
    ______     ______
```

```
    'AA'       1.539
    'AXP'     0.3555
    'C'       9.6533
    'DD'      4.0684
    'HPQ'     17.698
    'JPM'     21.565
    'MCD'     26.736
    'MO'      13.648
    'MSFT'    2.6858
    'UTX'     2.0509
```

```
fprintf('Max. Info Ratio portfolio has expected active return %0.2f%%\n', optPortRetn*2
```

```
Max. Info Ratio portfolio has expected active return 12.14%
```

```
fprintf('Max. Info Ratio portfolio has expected tracking error of %0.2f%%\n', optPortRi
```

```
Max. Info Ratio portfolio has expected tracking error of 6.32%
```

Utility Function

```
function [infoRatio,wts] = infoRatioTargetReturn(targetReturn,portObj)
% Calculate information ratio for a target-return portfolio along the
% efficient frontier
wts = estimateFrontierByReturn(portObj,targetReturn);
portRiskAct = estimatePortRisk(portObj,wts);
infoRatio = targetReturn/portRiskAct;
end
```

See Also

`Portfolio` | `fminbnd` | `inforatio`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Working with Portfolio Constraints Using Defaults” on page 4-67
- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129

- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-138
- “Portfolio Optimization Examples” on page 4-147
- “Information Ratio” on page 7-8

More About

- “Performance Metrics Overview” on page 7-2
- “Portfolio Object” on page 4-23
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-21

External Websites

- Using MATLAB to Optimize Portfolios with Financial Toolbox (33 min 24 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

CVaR Portfolio Optimization Tools

Portfolio Optimization Theory

In this section...

Portfolio Optimization Problems

Portfolio optimization problems involve identifying portfolios that satisfy three criteria:

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

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

The set of feasible portfolios is necessarily a nonempty, closed, and bounded set. The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices. The proxy for return is a function that characterizes either the gross or net benefits associated with portfolio choices. The terms “risk” and “risk proxy” and “return” and “return proxy” are interchangeable. The fundamental insight of Markowitz (see “Portfolio Optimization” on page A-7) is that the goal of the portfolio choice problem is to seek minimum risk for a given level of return and to seek maximum return for a given level of risk. Portfolios satisfying these criteria are efficient portfolios and the graph of the risks and returns of these portfolios forms a curve called the efficient frontier.

Portfolio Problem Specification

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

- Proxy for portfolio return (μ)

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

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

- The Portfolio object (`Portfolio`) supports mean-variance portfolio optimization (see Markowitz [46], [47] at "Portfolio Optimization" on page A-7). This object has either gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set.
- The PortfolioCVaR object (`PortfolioCVaR`) implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at "Portfolio Optimization" on page A-7), which is generally referred to as CVaR portfolio optimization. CVaR portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses conditional value-at-risk of portfolio returns as the risk proxy.
- The PortfolioMAD object (`PortfolioMAD`) implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7), which is referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

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

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

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

and (sample) covariance of asset returns

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

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

Gross Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

Net Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

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

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

Risk Proxy

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

Variance

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

$$\sum(x) = x^T Cx$$

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

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

Although the risk proxy in mean-variance portfolio optimization is the variance of portfolio returns, the square root, which is the standard deviation of portfolio returns, is often reported and displayed. Moreover, this quantity is often called the "risk" of the portfolio. For details, see Markowitz ("Portfolio Optimization" on page A-7).

Conditional Value-at-Risk

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

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

where:

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

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

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

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

The value-at-risk is defined as

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

An alternative formulation for CVaR has the form:

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

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

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

To describe the probability distribution of returns, the `PortfolioCVaR` object takes a finite sample of return scenarios y_s, with $s = 1,\ldots,S$. Each y_s is an n vector that contains the returns for each of the n assets under the scenario s. This sample of S scenarios is stored as a scenario matrix of size S-by-n. Then, the risk proxy for CVaR portfolio optimization, for a given portfolio $x \in X$ and $\alpha \in (0,1)$, is computed as

$$\sum(x) = VaR_\alpha(x) + \frac{1}{(1-\alpha)S}\sum_{s=1}^{S} \max\left\{0, -y_s^T x - VaR_\alpha(x)\right\}$$

The value-at-risk, $VaR_\alpha(x)$, is estimated whenever the CVaR is estimated. The loss function is $f(x, y_s) = -y_s^T x$, which is the portfolio loss under scenario s.

Under this definition, VaR and CVaR are sample estimators for VaR and CVaR based on the given scenarios. Better scenario samples yield more reliable estimates of VaR and CVaR.

For more information, see Rockafellar and Uryasev [48], [49], and Cornuejols and Tütüncü, [51], at "Portfolio Optimization" on page A-7.

Mean Absolute-Deviation

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

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

where:

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

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

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

such that

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

For more information, see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7.

See Also

`PortfolioCVaR`

Related Examples

- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
- "Default Portfolio Problem" on page 5-18
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Portfolio Set for Optimization Using PortfolioCVaR Object

The final element for a complete specification of a portfolio optimization problem is the

set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset R^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

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

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

Linear Inequality Constraints

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

$$A_I x \le b_I$$

where:

x is the portfolio (n vector).

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

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

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

Portfolio object properties to specify linear inequality constraints are:

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

The default is to ignore these constraints.

Linear Equality Constraints

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

$$A_E x = b_E$$

where:

x is the portfolio (n vector).

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

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

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

Portfolio object properties to specify linear equality constraints are:

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

The default is to ignore these constraints.

Bound Constraints

Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Since every portfolio set must be bounded, it is

often a good practice, albeit not necessary, to set explicit bounds for the portfolio problem. To obtain explicit bounds for a given portfolio set, use the `estimateBounds` function. Bound constraints take the form

$$l_B \le x \le u_B$$

where:

x is the portfolio (n vector).

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

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

n is the number of assets in the universe.

Portfolio object properties to specify bound constraints are:

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

The default is to ignore these constraints.

The default portfolio optimization problem (see "Default Portfolio Problem" on page 5-18) has l_B = `0` with u_B set implicitly through a budget constraint.

Budget Constraints

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

$$l_S \le 1^T x \le u_S$$

where:

x is the portfolio (n vector).

`1` is the vector of ones (n vector).

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

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

n is the number of assets in the universe.

Portfolio object properties to specify budget constraints are:

- `LowerBudget` for l_S
- `UpperBudget` for u_S
- `NumAssets` for n

The default is to ignore this constraint.

The default portfolio optimization problem (see "Default Portfolio Problem" on page 5-18) has $l_S = u_S = 1$, which means that the portfolio weights sum to `1`. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0–100% invested in cash. If cash is to be a portfolio choice, set `RiskFreeRate` (r_0) to a suitable value (see "Portfolio Problem Specification" on page 5-3 and "Working with a Riskless Asset" on page 5-56).

Group Constraints

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

$$l_G \le Gx \le u_G$$

where:

x is the portfolio (n vector).

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

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

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

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

Portfolio object properties to specify group constraints are:

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

The default is to ignore these constraints.

Group Ratio Constraints

Group ratio constraints are specialized linear constraints that enforce relationships among groups of assets. The constraints take the form
$l_{Ri}(G_B x)_i \le (G_A x)_i \le u_{Ri}(G_B x)_i$

for $i = 1,..., n_R$ where:

x is the portfolio (n vector).

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

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

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

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

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

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

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

Portfolio object properties to specify group ratio constraints are:

- `GroupA` for G_A
- `GroupB` for G_B
- `LowerRatio` for l_R

- `UpperRatio` for u_R
- `NumAssets` for n

The default is to ignore these constraints.

Average Turnover Constraints

Turnover constraint is a linear absolute value constraint that ensures estimated optimal portfolios differ from an initial portfolio by no more than a specified amount. Although portfolio turnover is defined in many ways, the turnover constraints implemented in Financial Toolbox computes portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

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

where:

x is the portfolio (n vector).

`1` is the vector of ones (n vector).

x_0 is the initial portfolio (n vector).

τ is the upper bound for turnover (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

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

The default is to ignore this constraint.

One-way Turnover Constraints

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

$$1^T \times \max\{0, x - x_0\} \le \tau_B$$

$$1^T \times \max\{0, x_0 - x\} \le \tau_S$$

where:

x is the portfolio (n vector)

`1` is the vector of ones (n vector).

x_0 is the Initial portfolio (n vector).

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

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

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

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

The default is to ignore this constraint.

Note The average turnover constraint (see "Average Turnover Constraints" on page 5-15) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

See Also

`PortfolioCVaR`

Related Examples

- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63

More About

- "PortfolioCVaR Object" on page 5-22
- "Default Portfolio Problem" on page 5-18
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Default Portfolio Problem

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

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

See Also

`PortfolioCVaR`

Related Examples

- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

PortfolioCVaR Object Workflow

The PortfolioCVaR object workflow for creating and modeling a CVaR portfolio is:

1. Create a CVaR Portfolio.

   Create a `PortfolioCVaR` object for conditional value-at-risk (CVaR) portfolio optimization. For more information, see “Creating the PortfolioCVaR Object” on page 5-27.

2. Define asset returns and scenarios.

   Evaluate scenarios for portfolio asset returns, including assets with missing data and financial time series data. For more information, see “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44.

3. Specify the CVaR Portfolio Constraints.

   Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, and turnover constraints. For more information, see “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63.

4. Validate the CVaR Portfolio.

   Identify errors for the portfolio specification. For more information, see “Validate the CVaR Portfolio Problem” on page 5-96.

5. Estimate the efficient portfolios and frontiers.

   Analyze the efficient portfolios and efficient frontiers for a CVaR portfolio. For more information, see “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101 and “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119.

6. Postprocess the results.

   Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see “Postprocessing Results to Set Up Tradable Portfolios” on page 5-130.

See Also

More About

- “Portfolio Optimization Theory” on page 5-3

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

PortfolioCVaR Object

In this section...

PortfolioCVaR Object Properties and Functions

The PortfolioCVaR object implements conditional value-at-risk (CVaR) portfolio optimization. Every property and function of the PortfolioCVaR object is public, although some properties and functions are hidden. See`PortfolioCVaR` for the properties and functions of a PortfolioCVaR object. The PortfolioCVaR object is a value object where every instance of the object is a distinct version of the object. Since the PortfolioCVaR object is also a MATLAB object, it inherits the default functions associated with MATLAB objects.

Working with PortfolioCVaR Objects

The PortfolioCVaR object and its functions are an interface for conditional value-at-risk portfolio optimization. So, almost everything you do with the PortfolioCVaR object can be done using the functions. The basic workflow is:

1. Design your portfolio problem.
2. Use the `PortfolioCVaR` function to create the PortfolioCVaR object or use the various set functions to set up your portfolio problem.
3. Use estimate functions to solve your portfolio problem.

In addition, functions are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a PortfolioCVaR object, you can

save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of CVaR portfolio optimization, means that you have either scenarios, data, or moments for asset returns, a probability level, and a collection of constraints on your portfolios, use the `PortfolioCVaR` function to set the properties for the PortfolioCVaR object.

The`PortfolioCVaR` function lets you create an object from scratch or update an existing object. Since the PortfolioCVaR object is a value object, it is easy to create a basic object, then use functions to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see "Creating the PortfolioCVaR Object" on page 5-27.

Setting and Getting Properties

You can set properties of a PortfolioCVaR object using either the `PortfolioCVaR` function or various set functions.

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

The `PortfolioCVaR` function supports setting properties with name-value pair arguments such that each argument name is a property and each value is the value to assign to that property. For example, to set the `LowerBound`, `Budget`, and `ProbabilityLevel` properties in an existing PortfolioCVaR object `p`, use the syntax:

```
p = PortfolioCVaR(p,'LowerBound', 0, 'Budget', 'ProbabilityLevel', 0.95);
```

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

- The `PortfolioCVaR` function and set functions try to determine the dimensions of your problem with either explicit or implicit inputs.
- The `PortfolioCVaR` function and set functions try to resolve ambiguities with default choices.

- The `PortfolioCVaR` function and set functions perform scalar expansion on arrays when possible.
- The CVaR functions try to diagnose and warn about problems.

Displaying PortfolioCVaR Objects

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

Saving and Loading PortfolioCVaR Objects

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

Estimating Efficient Portfolios and Frontiers

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

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

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

Arrays of PortfolioCVaR Objects

Although all functions associated with a PortfolioCVaR object are designed to work on a scalar PortfolioCVaR object, the array capabilities of MATLAB enables you to set up and

work with arrays of PortfolioCVaR objects. The easiest way to do this is with the `repmat` function. For example, to create a 3-by-2 array of PortfolioCVaR objects:

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

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

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

This example calls the `PortfolioCVaR` function for the (`i`,`j`) element of a matrix of PortfolioCVaR objects in the variable `p`.

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

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

and, once set, you can access these bounds with

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

PortfolioCVaR object functions work on only one PortfolioCVaR object at a time.

Subclassing PortfolioCVaR Objects

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

Conventions for Representation of Data

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

- Asset returns or prices for scenarios are in matrix form with samples for a given asset going down the rows and assets going across the columns. In the case of prices, the earliest dates must be at the top of the matrix, with increasing dates going down.
- Portfolios are in vector or matrix form with weights for a given portfolio going down the rows and distinct portfolios going across the columns.
- Constraints on portfolios are formed in such a way that a portfolio is a column vector.
- Portfolio risks and returns are either scalars or column vectors (for multiple portfolio risks and returns).

See Also

`PortfolioCVaR`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63

More About

- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Creating the PortfolioCVaR Object

In this section...

To create a fully specified CVaR portfolio optimization problem, instantiate the PortfolioCVaR object using the `PortfolioCVaR` function. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow” on page 5-20.

Syntax

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

```
p = PortfolioCVaR;
```

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

The `PortfolioCVaR` function also accepts collections of argument name-value pair arguments for properties and their values. The `PortfolioCVaR` function accepts inputs for public properties with the general syntax:

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

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

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

Input argument names are not case-sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 5-32). The `PortfolioCVaR` function tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo

various scalar or matrix expansion operations that simplify the overall process to formulate a problem. In addition, a PortfolioCVaR object is a value object so that, given portfolio `p`, the following code creates two objects, `p` and `q`, that are distinct:

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

PortfolioCVaR Problem Sufficiency

A CVaR portfolio optimization problem is completely specified with the PortfolioCVaR object if the following three conditions are met:

- You must specify a collection of asset returns or prices known as scenarios such that all scenarios are finite asset returns or prices. These scenarios are meant to be samples from the underlying probability distribution of asset returns. This condition can be satisfied by the `setScenarios` function or with several canned scenario simulation functions.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded. You can satisfy this condition using an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed and several tools, such as the `estimateBounds` function, provide ways to ensure that your problem is properly formulated.
- You must specify a probability level to locate the level of tail loss above which the conditional value-at-risk is to be minimized. This condition can be satisfied by the `setProbabilityLevel` function.

  Although the general sufficient conditions for CVaR portfolio optimization go beyond the first three conditions, the PortfolioCVaR object handles all these additional conditions.

PortfolioCVaR Function Examples

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

```
p = PortfolioCVaR;
disp(p);

PortfolioCVaR with properties:
```

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

The approaches listed provide a way to set up a portfolio optimization problem with the `PortfolioCVaR` function. The custom set functions offer additional ways to set and modify collections of properties in the PortfolioCVaR object.

Using the PortfolioCVaR Function for a Single-Step Setup

You can use the`PortfolioCVaR` function to directly set up a “standard” portfolio optimization problem. Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified as follows:

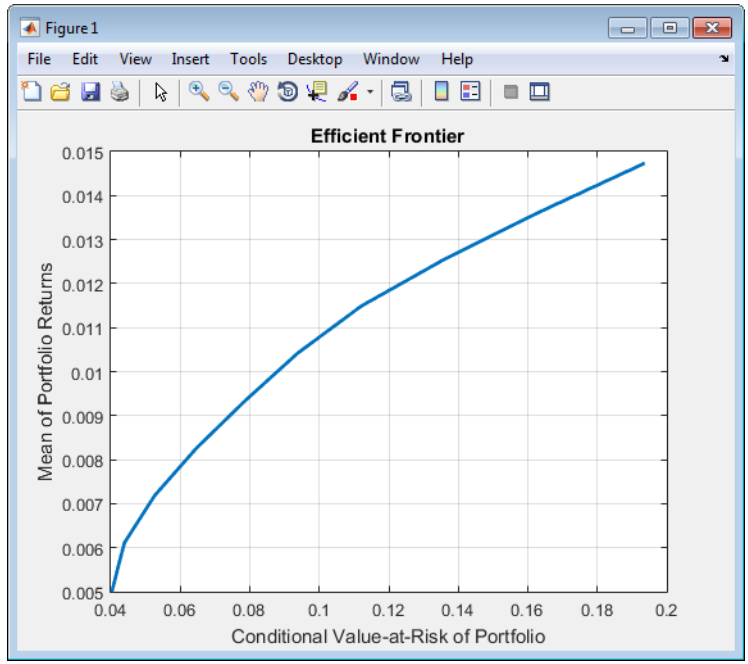
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```
p = PortfolioCVaR('Scenarios', AssetScenarios, ...
'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1, ...
'ProbabilityLevel', 0.95);
```

The `LowerBound` property value undergoes scalar expansion since `AssetScenarios` provides the dimensions of the problem.

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

```
p.plotFrontier;
```

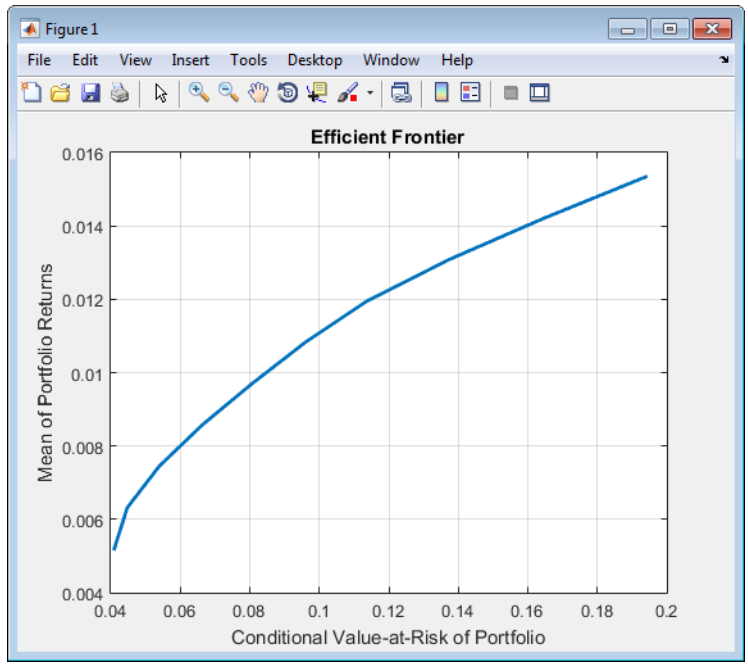


Using the PortfolioCVaR Function with a Sequence of Steps

An alternative way to accomplish the same task of setting up a “standard” CVaR portfolio optimization problem, given `AssetScenarios` variable is:

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

```
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = PortfolioCVaR(p, 'LowerBound', 0);
p = PortfolioCVaR(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```



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

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

```
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = PortfolioCVaR(p, 'LowerBound', zeros(size(m)));
p = PortfolioCVaR(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setProbabilityLevel(p, 0.95);
p = setScenarios(p, AssetScenarios);
```

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

Shortcuts for Property Names

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

Shortcuts for Property Names

Shortcut Argument Name	Equivalent Argument / Property Name
`ae`	`AEquality`
`ai`	`AInequality`
`assetnames` or `assets`	`AssetList`
`be`	`bEquality`
`bi`	`bInequality`
`budget`	`UpperBudget` and `LowerBudget`
`group`	`GroupMatrix`
`lb`	`LowerBound`
`n` or `num`	`NumAssets`
`level`, `problevel`, or `plevel`	`ProbabilityLevel`
`rfr`	`RiskFreeRate`
`scenario` or `assetscenarios`	`Scenarios`
`ub`	`UpperBound`

For example, this call to the `PortfolioCVaR` function uses these shortcuts for properties:

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('scenario', AssetScenarios, 'lb', 0, 'budget', 1, 'plevel', 0.95);
plotFrontier(p);
```

Direct Setting of Portfolio Object Properties

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

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;

p = setScenarios(p, AssetScenarios);
p.ProbabilityLevel = 0.95;

p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));

plotFrontier(p);
```

Note Scenarios cannot be assigned directly to a PortfolioCVaR object. Scenarios must always be set through either the `PortfolioCVaR` function, the `setScenarios` function, or any of the scenario simulation functions.

See Also

`PortfolioCVaR` | `estimateBounds`

Related Examples

- "Common Operations on the PortfolioCVaR Object" on page 5-36
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Common Operations on the PortfolioCVaR Object

In this section...

Naming a PortfolioCVaR Object

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

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

```
Asset Allocation Fund
```

Configuring the Assets in the Asset Universe

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

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

```
4
```

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

Setting Up a List of Asset Identifiers

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

Setting Up Asset Lists Using the PortfolioCVaR Function

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

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

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

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

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

```
5
```

Setting Up Asset Lists Using the setAssetList Function

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

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

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

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

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

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

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

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

```
p = PortfolioCVaR('numassets',5);
p = setAssetList(p);
disp(p.AssetList);

'Asset1'    'Asset2'    'Asset3'    'Asset4'    'Asset5'
```

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

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

Truncating and Padding Asset Lists

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

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

Warning: Input list of assets has 2 too few identifiers. Padding with numbered assets.
> In PortfolioCVaR.setAssetList at 118
    '921937835'    '922908769'    '922042775'    'UnknownCUSIP4'    'UnknownCUSIP5'
```

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

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

Warning: AssetList has 1 too many identifiers. Using first 4 assets.
> In PortfolioCVaR.checkarguments at 399
  In PortfolioCVaR.PortfolioCVaR>PortfolioCVaR.PortfolioCVaR at 195
    'AGG'    'EEM'    'MDY'    'SPY'
```

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

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

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

See Also

`PortfolioCVaR` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort`

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 5-41
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Setting Up an Initial or Current Portfolio

In many applications, creating a new optimal portfolio requires comparing the new portfolio with an initial or current portfolio to form lists of purchases and sales. The PortfolioCVaR object property `InitPort` lets you identify an initial or current portfolio. The initial portfolio also plays an essential role if you have either transaction costs or turnover constraints. The initial portfolio need not be feasible within the constraints of the problem. This can happen if the weights in a portfolio have shifted such that some constraints become violated. To check if your initial portfolio is feasible, use the `checkFeasibility` function described in "Validating CVaR Portfolios" on page 5-98. Suppose that you have an initial portfolio in `x0`, then use the `PortfolioCVaR` function to set up an initial portfolio:

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

```
  0.3000
  0.2000
  0.2000
       0
```

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

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

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

To clear an initial portfolio from your PortfolioCVaR object, use either the `PortfolioCVaR` function or the `setInitPort` function with an empty input for the `InitPort` property. If transaction costs or turnover constraints are set, it is not possible

to clear the InitPort property in this way. In this case, to clear InitPort, first clear the dependent properties and then clear theInitPort property.

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

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

```
p = PortfolioCVaR;
p = setInitPort(p, 1/4, 4);
disp(p.InitPort);
```

```
 0.2500
 0.2500
 0.2500
 0.2500
```

PortfolioCVaR object functions that work with either transaction costs or turnover constraints also depend on the InitPort property. So, the set functions for transaction costs or turnover constraints permit the assignment of a value for the InitPort property as part of their implementation. For details, see “Working with Average Turnover Constraints Using PortfolioCVaR Object” on page 5-88, “Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-92, and “Working with Transaction Costs” on page 5-58. If either transaction costs or turnover constraints are used, then the InitPort property must have a nonempty value. Absent a specific value assigned through the PortfolioCVaR function or various set functions, the PortfolioCVaR object sets InitPort to 0 and warns if BuyCost, SellCost, or Turnover properties are set. This example shows what happens if you specify an average turnover constraint with an initial portfolio:

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

```
    0.3000
    0.2000
    0.2000
         0
```

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

```
p = PortfolioCVaR('Turnover', 0.3);
disp(p.InitPort);
```

```
Warning: InitPort and NumAssets are empty and either transaction costs or turnover constraints specified.
Will set NumAssets = 1 and InitPort = 0.
> In PortfolioCVaR.checkarguments at 322
  In PortfolioCVaR.PortfolioCVaR>PortfolioCVaR.PortfolioCVaR at 195
     0
```

See Also

`PortfolioCVaR` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Common Operations on the PortfolioCVaR Object” on page 5-36
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Asset Returns and Scenarios Using PortfolioCVaR Object

In this section...

How Stochastic Optimization Works

The CVaR of a portfolio is a conditional expectation. (For the definition of the CVaR function, see "Risk Proxy" on page 5-6.) Therefore, the CVaR portfolio optimization problem is a stochastic optimization problem. Given a sample of scenarios, the conditional expectation that defines the sample CVaR of the portfolio can be expressed as a finite sum, a weighted average of losses. The weights of the losses depend on their relative magnitude; for a confidence level α, only the worst $(1 - \alpha) \times 100\%$ losses get a positive weight. As a function of the portfolio weights, the CVaR of the portfolio is a convex function (see [48], [49] Rockafellar & Uryasev at "Portfolio Optimization" on page A-7). It is also a nonsmooth function, but its edges are less sharp as the sample size increases.

There are reformulations of the CVaR portfolio optimization problem (see [48], [49] at Rockafellar & Uryasev) that result in a linear programming problem, which can be solved either with standard linear programming techniques or with stochastic programming solvers. The PortfolioCVaR object, however, does not reformulate the problem in such a manner. The PortfolioCVaR object computes the CVaR as a nonlinear function. The convexity of the CVaR, as a function of the portfolio weights and the dull edges when the number of scenarios is large, make the CVaR portfolio optimization problem tractable, in practice, for certain nonlinear programming solvers, such as `fmincon` from Optimization Toolbox. The problem can also be solved using a cutting-

plane method (see Kelley [45] at "Portfolio Optimization" on page A-7). For more information, see Algorithms section of `setSolver`. To learn more about the workflow when using PortfolioCVaR objects, see "PortfolioCVaR Object Workflow" on page 5-20.

What Are Scenarios?

Since conditional value-at-risk portfolio optimization works with scenarios of asset returns to perform the optimization, several ways exist to specify and simulate scenarios. In many applications with CVaR portfolio optimization, asset returns may have distinctly nonnormal probability distributions with either multiple modes, binning of returns, truncation of distributions, and so forth. In other applications, asset returns are modeled as the result of various simulation methods that might include Monte-Carlo simulation, quasi-random simulation, and so forth. Often, the underlying probability distribution for risk factors may be multivariate normal but the resultant transformations are sufficiently nonlinear to result in distinctively nonnormal asset returns.

For example, this occurs with bonds and derivatives. In the case of bonds with a nonzero probability of default, such scenarios would likely include asset returns that are −100% to indicate default and some values slightly greater than −100% to indicate recovery rates.

Although the PortfolioCVaR object has functions to simulate multivariate normal scenarios from either data or moments (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`), the usual approach is to specify scenarios directly from your own simulation functions. These scenarios are entered directly as a matrix with a sample for all assets across each row of the matrix and with samples for an asset down each column of the matrix. The architecture of the CVaR portfolio optimization tools references the scenarios through a function handle so scenarios that have been set cannot be accessed directly as a property of the PortfolioCVaR object.

Setting Scenarios Using the PortfolioCVaR Function

Suppose that you have a matrix of scenarios in the `AssetScenarios` variable. The scenarios are set through the `PortfolioCVaR` function with:

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

```
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('Scenarios', AssetScenarios);

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

```
4

20000
```

Notice that the PortfolioCVaR object determines and fixes the number of assets in `NumAssets` and the number of scenarios in `NumScenarios` based on the scenario's matrix. You can change the number of scenarios by calling the `PortfolioCVaR` function with a different scenario matrix. However, once the `NumAssets` property has been set in the object, you cannot enter a scenario matrix with a different number of assets. The `getScenarios` function lets you recover scenarios from a PortfolioCVaR object. You can also obtain the mean and covariance of your scenarios using `estimateScenarioMoments`.

Although not recommended for the casual user, an alternative way exists to recover scenarios by working with the function handle that points to scenarios in the PortfolioCVaR object. To access some or all the scenarios from a PortfolioCVaR object, the hidden property `localScenarioHandle` is a function handle that points to a function to obtain scenarios that have already been set. To get scenarios directly from a PortfolioCVaR object `p`, use

```
scenarios = p.localScenarioHandle([], []);
```

and to obtain a subset of scenarios from rows `startrow` to `endrow`, use

```
scenarios = p.localScenarioHandle(startrow, endrow);
```

where $1 \le$ `startrow` $\le$ `endrow` $\le$ `numScenarios`.

Setting Scenarios Using the setScenarios Function

You can also set scenarios using `setScenarios`. For example, given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set:

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);

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

```
4

20000
```

Estimating the Mean and Covariance of Scenarios

The `estimateScenarioMoments` function obtains estimates for the mean and covariance of scenarios in a PortfolioCVaR object.

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```
p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
[mean, covar] = estimateScenarioMoments(p)

mean =

    0.0043
    0.0085
    0.0098
    0.0153


covar =

    0.0005    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0049    0.0029
    0.0000    0.0010    0.0029    0.0102
```

Simulating Normal Scenarios

As a convenience, the two functions (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`) exist to simulate scenarios from data or moments under an assumption that they are distributed as multivariate normal random asset returns.

Simulating Normal Scenarios from Returns or Prices

Given either return or price data, use the function `simulateNormalScenariosByData` to simulate multivariate normal scenarios. Either returns or prices are stored as matrices with samples going down the rows and assets going across the columns. In addition, returns or prices can be stored in a financial time series `fints` object (see "Simulating Normal Scenarios from Time Series Data" on page 5-52). To illustrate using `simulateNormalScenariosByData`, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with `portsim`. The default behavior of `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by `portsim` in the variable *X*, a price series is created in the variable *Y*:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
```

```
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);
Y = ret2tick(X);
```

Note Portfolio optimization requires that you use total returns and not just price returns. So, "returns" should be total returns and "prices" should be total return prices.

Given asset returns and prices in variables *X* and *Y* from above, this sequence of examples demonstrates equivalent ways to simulate multivariate normal scenarios for the PortfolioCVaR object. Assume a PortfolioCVaR object created in `p` that uses the asset returns in *X* uses `simulateNormalScenariosByData`:

```
p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, X, 20000);

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0043
    0.0083
    0.0102
    0.1507


passetcovar =

    0.0053    0.0003    0.0002    0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0049    0.0028
    0.0000    0.0010    0.0028    0.0101
```

The moments that you obtain from this simulation will likely differ from the moments listed here because the scenarios are random samples from the estimated multivariate normal probability distribution of the input returns *X*.

The default behavior of `simulateNormalScenariosByData` is to work with asset returns. If, instead, you have asset prices as in the variable *Y*, `simulateNormalScenariosByData` accepts a name-value pair argument name

'DataFormat' with a corresponding value set to 'prices' to indicate that the input to the function is in the form of asset prices and not returns (the default value for the 'DataFormat' argument is 'returns'). This example simulates scenarios with the asset price data in *Y* for the PortfolioCVaR object q:

```
p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =

    0.0043
    0.0084
    0.0094
    0.1490


passetcovar =

    0.0054    0.0004    0.0001   -0.0000
    0.0004    0.0024    0.0016    0.0009
    0.0001    0.0016    0.0048    0.0028
   -0.0000    0.0009    0.0028    0.0100
```

Simulating Normal Scenarios with Missing Data

Often when working with multiple assets, you have missing data indicated by NaN values in your return or price data. Although "Multivariate Normal Regression" on page 9-2 goes into detail about regression with missing data, the simulateNormalScenariosByData function has a name-value pair argument name 'MissingData' that indicates with a Boolean value whether to use the missing data capabilities of Financial Toolbox. The default value for 'MissingData' is false which removes all samples with NaN values. If, however, 'MissingData' is set to true, simulateNormalScenariosByData uses the ECM algorithm to estimate asset moments. This example shows how this works on price data with missing values:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];
```

```
X = portsim(m', C, 120);
Y = ret2tick(X);
Y(1:20,1) = NaN;
Y(1:12,4) = NaN;
```

Notice that the prices above in `Y` have missing values in the first and fourth series.

```
p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

q = PortfolioCVaR;
q = simulateNormalScenariosByData(q, Y, 20000, 'dataformat', 'prices', 'missingdata', true);

[passetmean, passetcovar] = estimateScenarioMoments(p)
[qassetmean, qassetcovar] = estimateScenarioMoments(q)
```

```
passetmean =

    0.0020
    0.0074
    0.0078
    0.1476


passetcovar =

    0.0055    0.0003   -0.0001   -0.0003
    0.0003    0.0024    0.0019    0.0012
   -0.0001    0.0019    0.0050    0.0028
   -0.0003    0.0012    0.0028    0.0101


qassetmean =

    0.0024
    0.0085
    0.0106
    0.1482


qassetcovar =

    0.0071    0.0004   -0.0001   -0.0004
    0.0004    0.0032    0.0022    0.0012
   -0.0001    0.0022    0.0063    0.0034
   -0.0004    0.0012    0.0034    0.0127
```

The first PortfolioCVaR object, `p`, contains scenarios obtained from price data in `Y` where `NaN` values are discarded and the second PortfolioCVaR object, `q`, contains scenarios obtained from price data in `Y` that accommodate missing values. Each time you run this example, you get different estimates for the moments in `p` and `q`.

Simulating Normal Scenarios from Time Series Data

The `simulateNormalScenariosByData` function also accepts asset returns or prices stored in financial time series (`fints`) objects. The function implicitly works with matrices of data or data in a `fints` object using the same rules for whether the data are returns or prices. To illustrate, use `fints` to create the `fints` object `Xfts` that contains asset returns generated with `fints` (see "Estimating Asset Moments from Prices or Returns" on page 4-52) and add series labels:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);

d = (datenum('31-jan-2001'):datenum('31-dec-2010'))';
Xfts = fints(d, zeros(numel(d),4), {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'});
Xfts = tomonthly(Xfts);

Xfts.Bonds = X(:,1);
Xfts.LargeCap = X(:,2);
Xfts.SmallCap = X(:,3);
Xfts.Emerging = X(:,4);

p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, Xfts, 20000);

[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =

    0.0044
    0.0082
    0.0102
    0.1504


passetcovar =

    0.0054    0.0004    0.0002   -0.0000
    0.0004    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0047    0.0027
   -0.0000    0.0010    0.0027    0.0102
```

The name-value inputs `'DataFormat'` to handle return or price data and`'MissingData'` to ignore or use samples with missing values also work for `fints` data. In addition, `simulateNormalScenariosByData` extracts asset names or

identifiers from a `fints` object if the argument name `'GetAssetList'` is set to `true` (the default value is `false`). If the `'GetAssetList'` value is `true`, the identifiers are used to set the `AssetList` property of the PortfolioCVaR object. Thus, repeating the formation of the PortfolioCVaR object `q` from the previous example with the `'GetAssetList'` flag set to `true` extracts the series labels from the `fints` object:

```
p = simulateNormalScenariosByData(p, Xfts, 20000, 'getassetlist', true);
disp(p.AssetList)
```

```
'Bonds' 'LargeCap' 'SmallCap' 'Emerging'
```

If you set the`'GetAssetList'` flag set to `true` and your input data is in a matrix, `simulateNormalScenariosByData` uses the default labeling scheme from `setAssetList` as described in "Setting Up a List of Asset Identifiers" on page 5-37.

Simulating Normal Scenarios with Mean and Covariance

Given the mean and covariance of asset returns, use the `simulateNormalScenariosByMoments` function to simulate multivariate normal scenarios. The mean can be either a row or column vector and the covariance matrix must be a symmetric positive-semidefinite matrix. Various rules for scalar expansion apply. To illustrate using `simulateNormalScenariosByMoments`, start with moments in `m` and `C` and generate 20,000 scenarios:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

p = PortfolioCVaR;
p = simulateNormalScenariosByMoments(p, m, C, 20000);
[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =

    0.0049
    0.0083
    0.0101
    0.1503


passetcovar =
```

```
    0.0053    0.0003    0.0002   -0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0047    0.0028
   -0.0000    0.0010    0.0028    0.0101
```

`simulateNormalScenariosByMoments` performs scalar expansion on arguments for the moments of asset returns. If `NumAssets` has not already been set, a scalar argument is interpreted as a scalar with `NumAssets` set to `1`. `simulateNormalScenariosByMoments` provides an additional optional argument to specify the number of assets so that scalar expansion works with the correct number of assets. In addition, if either a scalar or vector is input for the covariance of asset returns, a diagonal matrix is formed such that a scalar expands along the diagonal and a vector becomes the diagonal.

See Also

`PortfolioCVaR` | `estimatePortVaR` | `setCosts` | `setProbabilityLevel` | `setScenarios` | `simulateNormalScenariosByData` | `simulateNormalScenariosByMoments`

Related Examples

- “Working with a Riskless Asset” on page 5-56
- “Working with Transaction Costs” on page 5-58
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3

- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with a Riskless Asset

The PortfolioCVaR object has a separate `RiskFreeRate` property that stores the rate of return of a riskless asset. Thus, you can separate your universe into a riskless asset and a collection of risky assets. For example, assume that your riskless asset has a return in the scalar variable `r0`, then the property for the `RiskFreeRate` is set using the `PortfolioCVaR` function:

```
r0 = 0.01/12;

p = PortfolioCVaR;
p = PortfolioCVaR('RiskFreeRate', r0);
disp(p.RiskFreeRate);

8.3333e-04
```

Note If your portfolio problem has a budget constraint such that your portfolio weights must sum to `1`, then the riskless asset is irrelevant.

See Also

`PortfolioCVaR` | `estimatePortVaR` | `setCosts` | `setProbabilityLevel` | `setScenarios` | `simulateNormalScenariosByData` | `simulateNormalScenariosByMoments`

Related Examples

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Working with Transaction Costs” on page 5-58
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Transaction Costs

The difference between net and gross portfolio returns is transaction costs. The net portfolio return proxy has distinct proportional costs to purchase and to sell assets which are maintained in the PortfolioCVaR object properties `BuyCost` and `SellCost`. Transaction costs are in units of total return and, as such, are proportional to the price of an asset so that they enter the model for net portfolio returns in return form. For example, suppose that you have a stock currently priced $40 and your usual transaction costs are 5 cents per share. Then the transaction cost for the stock is 0.05/40 = 0.00125 (as defined in "Net Portfolio Returns" on page 5-5). Costs are entered as positive values and credits are entered as negative values.

Setting Transaction Costs Using the PortfolioCVaR Function

To set up transaction costs, you must specify an initial or current portfolio in the `InitPort` property. If the initial portfolio is not set when you set up the transaction cost properties, `InitPort` is `0`. The properties for transaction costs can be set using the `PortfolioCVaR` function. For example, assume that purchase and sale transaction costs are in the variables `bc` and `sc` and an initial portfolio is in the variable `x0`, then transaction costs are set:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioCVaR('BuyCost', bc, 'SellCost', sc, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
     5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
```

```
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Setting Transaction Costs Using the setCosts Function

You can also set the properties for transaction costs using `setCosts`. Assume that you have the same costs and initial portfolio as in the previous example. Given a PortfolioCVaR object `p` with an initial portfolio already set, use `setCosts` to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioCVaR('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
    5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

    0.4000
    0.2000
```

```
    0.2000
    0.1000
    0.1000
```

You can also set up the initial portfolio's `InitPort` value as an optional argument to `setCosts` so that the following is an equivalent way to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioCVaR;
p = setCosts(p, bc, sc, x0);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
    5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Setting Transaction Costs with Scalar Expansion

Both the `PortfolioCVaR` function and `setCosts` function implement scalar expansion on the arguments for transaction costs and the initial portfolio. If the `NumAssets`

property is already set in the PortfolioCVaR object, scalar arguments for these properties are expanded to have the same value across all dimensions. In addition, `setCosts` lets you specify `NumAssets` as an optional final argument. For example, assume that you have an initial portfolio `x0` and you want to set common transaction costs on all assets in your universe. You can set these costs in any of these equivalent ways:

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioCVaR('InitPort', x0, 'BuyCost', 0.002, 'SellCost', 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setCosts(p, 0.002, 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioCVaR;
p = setCosts(p, 0.002, 0.002, x0);
```

To clear costs from your PortfolioCVaR object, use either the `PortfolioCVaR` function or `setCosts` with empty inputs for the properties to be cleared. For example, you can clear sales costs from the PortfolioCVaR object `p` in the previous example:

```
p = PortfolioCVaR(p, 'SellCost', []);
```

See Also

`PortfolioCVaR` | `estimatePortVaR` | `setCosts` | `setProbabilityLevel` | `setScenarios` | `simulateNormalScenariosByData` | `simulateNormalScenariosByMoments`

Related Examples

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Working with a Riskless Asset” on page 5-56
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96

- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with CVaR Portfolio Constraints Using Defaults

The final element for a complete specification of a portfolio optimization problem is the

set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset R^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or "default" portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to `1` (using the budget constraint). For information on the workflow when using PortfolioCVaR objects, see "PortfolioCVaR Object Workflow" on page 5-20.

Setting Default Constraints for Portfolio Weights Using PortfolioCVaR Object

The "default" CVaR portfolio problem has two constraints on portfolio weights:

- Portfolio weights must be nonnegative.
- Portfolio weights must sum to `1`.

Implicitly, these constraints imply that portfolio weights are no greater than `1`, although this is a superfluous constraint to impose on the problem.

Setting Default Constraints Using the PortfolioCVaR Function

Given a portfolio optimization problem with `NumAssets` = `20` assets, use the `PortfolioCVaR` function to set up a default problem and explicitly set bounds and budget constraints:

```
p = PortfolioCVaR('NumAssets', 20, 'LowerBound', 0, 'Budget', 1);
disp(p);
```

```
  PortfolioCVaR with properties:

            BuyCost: []
           SellCost: []
       RiskFreeRate: []
    ProbabilityLevel: []
```

```
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: []
                Name: []
           NumAssets: 20
           AssetList: []
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: [20x1 double]
          UpperBound: []
         LowerBudget: 1
         UpperBudget: 1
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
```

Setting Default Constraints Using the setDefaultConstraints Function

An alternative approach is to use the `setDefaultConstraints` function. If the number of assets is already known in a PortfolioCVaR object, use `setDefaultConstraints` with no arguments to set up the necessary bound and budget constraints. Suppose that you have 20 assets to set up the portfolio set for a default problem:

```
p = PortfolioCVaR('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);
```

```
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: []
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
```

```
       NumScenarios: []
               Name: []
          NumAssets: 20
          AssetList: []
           InitPort: []
        AInequality: []
        bInequality: []
          AEquality: []
          bEquality: []
         LowerBound: [20x1 double]
         UpperBound: []
        LowerBudget: 1
        UpperBudget: 1
        GroupMatrix: []
         LowerGroup: []
         UpperGroup: []
             GroupA: []
             GroupB: []
         LowerRatio: []
         UpperRatio: []
```

If the number of assets is unknown, `setDefaultConstraints` accepts `NumAssets` as an optional argument to form a portfolio set for a default problem. Suppose that you have 20 assets:

```
p = PortfolioCVaR;
p = setDefaultConstraints(p, 20);
disp(p);
```

```
 PortfolioCVaR with properties:

              BuyCost: []
             SellCost: []
         RiskFreeRate: []
    ProbabilityLevel: []
             Turnover: []
          BuyTurnover: []
         SellTurnover: []
         NumScenarios: []
                 Name: []
            NumAssets: 20
            AssetList: []
             InitPort: []
          AInequality: []
          bInequality: []
```

```
     AEquality: []
     bEquality: []
    LowerBound: [20x1 double]
    UpperBound: []
   LowerBudget: 1
   UpperBudget: 1
   GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
        GroupA: []
        GroupB: []
    LowerRatio: []
    UpperRatio: []
```

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Working with Bound Constraints Using PortfolioCVaR Object” on page 5-68
- “Working with Budget Constraints Using PortfolioCVaR Object” on page 5-71
- “Working with Group Constraints Using PortfolioCVaR Object” on page 5-74
- “Working with Group Ratio Constraints Using PortfolioCVaR Object” on page 5-78
- “Working with Linear Equality Constraints Using PortfolioCVaR Object” on page 5-82
- “Working with Linear Inequality Constraints Using PortfolioCVaR Object” on page 5-85
- “Working with Average Turnover Constraints Using PortfolioCVaR Object” on page 5-88
- “Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-92
- “Creating the PortfolioCVaR Object” on page 5-27
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101

- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Bound Constraints Using PortfolioCVaR Object

Bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “Bound Constraints” on page 5-11). Although every CVaR portfolio set must be bounded, it is not necessary to specify a CVaR portfolio set with explicit bound constraints. For example, you can create a CVaR portfolio set with an implicit upper bound constraint or a CVaR portfolio set with average turnover constraints. The bound constraints have properties `LowerBound` for the lower-bound constraint and `UpperBound` for the upper-bound constraint. Set default values for these constraints using the `setDefaultConstraints` function (see “Setting Default Constraints for Portfolio Weights Using PortfolioCVaR Object” on page 5-63).

Setting Bounds Using the PortfolioCVaR Function

The properties for bound constraints are set through the `PortfolioCVaR` function. Suppose that you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. The bound constraints for a balanced fund are set with:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioCVaR('LowerBound', lb, 'UpperBound', ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
2

 0.5000
 0.2500

 0.7500
 0.5000
```

To continue with this example, you must set up a budget constraint. For details, see “Working with Budget Constraints Using PortfolioCVaR Object” on page 5-71.

Setting Bounds Using the setBounds Function

You can also set the properties for bound constraints using `setBounds`. Suppose that you have a balanced fund with stocks that can range from 50% to 75% of your portfolio

and bonds that can range from 25% to 50% of your portfolio. Given a PortfolioCVaR object `p`, use `setBounds` to set the bound constraints:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioCVaR;
p = setBounds(p, lb, ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
  2

  0.5000
  0.2500

  0.7500
  0.5000
```

Setting Bounds Using the PortfolioCVaR Function or setBounds Function

Both the `PortfolioCVaR` function and `setBounds` function implement scalar expansion on either the `LowerBound` or `UpperBound` properties. If the `NumAssets` property is already set in the PortfolioCVaR object, scalar arguments for either property expand to have the same value across all dimensions. In addition, `setBounds` lets you specify `NumAssets` as an optional argument. Suppose that you have a universe of 500 assets and you want to set common bound constraints on all assets in your universe. Specifically, you are a long-only investor and want to hold no more than 5% of your portfolio in any single asset. You can set these bound constraints in any of these equivalent ways:

```
p = PortfolioCVaR('NumAssets', 500, 'LowerBound', 0, 'UpperBound', 0.05);
```

or

```
p = PortfolioCVaR('NumAssets', 500);
p = setBounds(p, 0, 0.05);
```

or

```
p = PortfolioCVaR;
p = setBounds(p, 0, 0.05, 500);
```

To clear bound constraints from your PortfolioCVaR object, use either the `PortfolioCVaR` function or `setBounds` with empty inputs for the properties to be cleared. For example, to clear the upper-bound constraint from the PortfolioCVaR object `p` in the previous example:

```
p = PortfolioCVaR(p, 'UpperBound', []);
```

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Budget Constraints Using PortfolioCVaR Object

The budget constraint is an optional linear constraint that maintains upper and lower bounds on the sum of portfolio weights (see "Budget Constraints" on page 5-12). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a CVaR portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to `1` using `setDefaultConstraints` (see "Setting Default Constraints Using the PortfolioCVaR Function" on page 5-63).

Setting Budget Constraints Using the PortfolioCVaR Function

The properties for the budget constraint can also be set using the `PortfolioCVaR` function. Suppose that you have an asset universe with many risky assets and a riskless asset and you want to ensure that your portfolio never holds more than 1% cash, that is, you want to ensure that you are 99–100% invested in risky assets. The budget constraint for this portfolio can be set with:

```
p = PortfolioCVaR('LowerBudget', 0.99, 'UpperBudget', 1);
disp(p.LowerBudget);
disp(p.UpperBudget);
```

```
 0.9900

1
```

Setting Budget Constraints Using the setBudget Function

You can also set the properties for a budget constraint using `setBudget`. Suppose that you have a fund that permits up to 10% leverage which means that your portfolio can be from 100% to 110% invested in risky assets. Given a PortfolioCVaR object `p`, use `setBudget` to set the budget constraints:

```
p = PortfolioCVaR;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);
disp(p.UpperBudget);
```

```
     1

     1.1000
```

If you were to continue with this example, then set the `RiskFreeRate` property to the borrowing rate to finance possible leveraged positions. For details on the `RiskFreeRate` property, see "Working with a Riskless Asset" on page 5-56. To clear either bound for the budget constraint from your PortfolioCVaR object, use either the `PortfolioCVaR` function or `setBudget` with empty inputs for the properties to be cleared. For example, clear the upper-budget constraint from the PortfolioCVaR object `p` in the previous example with:

```
p = PortfolioCVaR(p, 'UpperBudget', []);
```

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
- "Validate the CVaR Portfolio Problem" on page 5-96
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Group Constraints Using PortfolioCVaR Object

Group constraints are optional linear constraints that group assets together and enforce bounds on the group weights (see “Group Constraints” on page 5-13). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the PortfolioCVaR Function

The properties for group constraints are set through the `PortfolioCVaR` function. Suppose that you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio, then you can set group constraints:

```
G = [ 1 1 1 0 0 ];
p = PortfolioCVaR('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

The group matrix `G` can also be a logical matrix so that the following code achieves the same result.

```
G = [ true true true false false ];
p = PortfolioCVaR('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

Setting Group Constraints Using the setGroups and addGroups Functions

You can also set the properties for group constraints using `setGroups`. Suppose that you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a PortfolioCVaR object `p`, use `setGroups` to set the group constraints:

```
G = [ true true true false false ];
p = PortfolioCVaR;
p = setGroups(p, G, [], 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

In this example, you would set the `LowerGroup` property to be empty (`[]`).

Suppose that you want to add another group constraint to make odd-numbered assets constitute at least 20% of your portfolio. Set up an augmented group matrix and introduce infinite bounds for unconstrained group bounds or use the `addGroups` function to build up group constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioCVaR;
G = [ true true true false false ];    % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ];    % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
 5

1     1     1     0     0
1     0     1     0     1

  -Inf
-Inf

0.3000
0.2000
```

```
0.3000
  Inf
```

addGroups determines which bounds are unbounded so you only need to focus on the constraints that you want to set.

The PortfolioCVaR function, setGroups, and addGroups implement scalar expansion on either the LowerGroup or UpperGroup properties based on the dimension of the group matrix in the property GroupMatrix. Suppose that you have a universe of 30 assets with 6 asset classes such that assets 1–5, assets 6–12, assets 13–18, assets 19–22, assets 23–27, and assets 28–30 constitute each of your asset classes and you want each asset class to fall from 0% to 25% of your portfolio. Let the following group matrix define your groups and scalar expansion define the common bounds on each group:

```
p = PortfolioCVaR;
G = blkdiag(true(1,5), true(1,7), true(1,6), true(1,4), true(1,5), true(1,3));
p = setGroups(p, G, 0, 0.25);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
30

  Columns 1 through 16

     1     1     1     1     1     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     1     1     1     1     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     1     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0

  Columns 17 through 30

     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0
     1     1     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     1     1     1     1     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     1     1     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     1     1     1

     0
     0
     0
     0
     0
     0

    0.2500
    0.2500
    0.2500
    0.2500
    0.2500
    0.2500
```

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
- "Validate the CVaR Portfolio Problem" on page 5-96
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Group Ratio Constraints Using PortfolioCVaR Object

Group ratio constraints are optional linear constraints that maintain bounds on proportional relationships among groups of assets (see “Group Ratio Constraints” on page 5-14). Although the constraints are implemented as general constraints, the usual convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group compared to a comparison group fall within specified bounds. Group ratio constraints have properties:

- `GroupA` for the base membership matrix
- `GroupB` for the comparison membership matrix
- `LowerRatio` for the lower-bound constraint on the ratio of groups
- `UpperRatio` for the upper-bound constraint on the ratio of groups

Setting Group Ratio Constraints Using the PortfolioCVaR Function

The properties for group ratio constraints are set using `PortfolioCVaR` function. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose that you have six assets with three financial companies (assets 1–3) and three nonfinanical companies (assets 4–6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ];    % financial companies
GB = [ 0 0 0 1 1 1 ];    % nonfinancial companies
p = PortfolioCVaR('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioCVaR('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Setting Group Ratio Constraints Using the setGroupRatio and addGroupRatio Functions

You can also set the properties for group ratio constraints using `setGroupRatio`. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose that you have six assets with three financial companies (assets 1–3) and three nonfinanical companies (assets 4–6). Given a PortfolioCVaR object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose that you want to add another group ratio constraint to ensure that the weights in odd-numbered assets constitute at least 20% of the weights in nonfinancial assets your portfolio. You can set up augmented group ratio matrices and introduce infinite bounds for unconstrained group ratio bounds, or you can use the `addGroupRatio` function to build up group ratio constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioCVaR;
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ];   % odd-numbered companies
GB = [ false false false true true true ];   % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);
```

```
 6

1     1     1     0     0     0
1     0     1     0     1     0

0     0     0     1     1     1
0     0     0     1     1     1

  -Inf
0.2000

0.5000
  Inf
```

Notice that `addGroupRatio` determines which bounds are unbounded so you only need to focus on the constraints you want to set.

The `PortfolioCVaR` function, `setGroupRatio`, and `addGroupRatio` implement scalar expansion on either the `LowerRatio` or `UpperRatio` properties based on the dimension of the group matrices in `GroupA` and `GroupB` properties.

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Linear Equality Constraints Using PortfolioCVaR Object

Linear equality constraints are optional linear constraints that impose systems of equalities on portfolio weights (see “Linear Equality Constraints” on page 5-11). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `bEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the PortfolioCVaR Function

The properties for linear equality constraints are set using the `PortfolioCVaR` function. Suppose that you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. To set this constraint:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR('AEquality', A, 'bEquality', b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);
```

```
5

1     1     1     0     0

0.5000
```

Setting Linear Equality Constraints Using the setEquality and addEquality Functions

You can also set the properties for linear equality constraints using `setEquality`. Suppose that you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioCVaR object `p`, use `setEquality` to set the linear equality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setEquality(p, A, b);
disp(p.NumAssets);
```

```
disp(p.AEquality);
disp(p.bEquality);
```

```
5

1     1     1     0     0

0.5000
```

Suppose that you want to add another linear equality constraint to ensure that the last three assets also constitute 50% of your portfolio. You can set up an augmented system of linear equalities or use `addEquality` to build up linear equality constraints. For this example, create another system of equalities:

```
p = PortfolioCVaR;
A = [ 1 1 1 0 0 ];    % first equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % second equality constraint
b = 0.5;
p = addEquality(p, A, b);

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

```
5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000
```

The `PortfolioCVaR` function, `setEquality`, and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Linear Inequality Constraints Using PortfolioCVaR Object

Linear inequality constraints are optional linear constraints that impose systems of inequalities on portfolio weights (see “Linear Inequality Constraints” on page 5-10). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the PortfolioCVaR Function

The properties for linear inequality constraints are set using the `PortfolioCVaR` function. Suppose that you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. To set up these constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR('AInequality', A, 'bInequality', b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5

1     1     1     0     0

0.5000
```

Setting Linear Inequality Constraints Using the setInequality and addInequality Functions

You can also set the properties for linear inequality constraints using `setInequality`. Suppose that you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 50% of your portfolio. Given a PortfolioCVaR object `p`, use `setInequality` to set the linear inequality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setInequality(p, A, b);
disp(p.NumAssets);
```

```
disp(p.AInequality);
disp(p.bInequality);
```

```
5

1     1     1     0     0

0.5000
```

Suppose that you want to add another linear inequality constraint to ensure that the last three assets constitute at least 50% of your portfolio. You can set up an augmented system of linear inequalities or use the `addInequality` function to build up linear inequality constraints. For this example, create another system of inequalities:

```
p = PortfolioCVaR;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

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

```
5

1     1     1     0     0
0     0    -1    -1    -1

0.5000
-0.5000
```

The `PortfolioCVaR` function, `setInequality`, and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Average Turnover Constraints Using PortfolioCVaR Object

The turnover constraint is an optional linear absolute value constraint (see “Average Turnover Constraints” on page 5-15) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `PortfolioCVaR` function or the `setTurnover` function. The turnover constraint depends on an initial or current portfolio, which is assumed to be zero if not set when the turnover constraint is set. The turnover constraint has properties `Turnover`, for the upper bound on average turnover, and `InitPort`, for the portfolio against which turnover is computed.

Setting Average Turnover Constraints Using the PortfolioCVaR Function

The properties for the turnover constraints are set using the `PortfolioCVaR` function. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and you want to ensure that average turnover is no more than 30%. To set this turnover constraint:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('Turnover', 0.3, 'InitPort', x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

Note if the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see “Setting Up an Initial or Current Portfolio” on page 5-41).

Setting Average Turnover Constraints Using the setTurnover Function

You can also set properties for portfolio turnover using `setTurnover` to specify both the upper bound for average turnover and an initial portfolio. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that average turnover is no more than 30%. Given a PortfolioCVaR object `p`, use `setTurnover` to set the turnover constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setTurnover(p, 0.3);

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

```
10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR;
p = setTurnover(p, 0.3, x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10

0.3000

0.1200
0.0900
0.0800
```

```
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

`setTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the PortfolioCVaR object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setTurnover` lets you specify `NumAssets` as an optional argument. To clear turnover from your PortfolioCVaR object, use the `PortfolioCVaR` function or `setTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with One-Way Turnover Constraints Using PortfolioCVaR Object

One-way turnover constraints are optional constraints (see “One-way Turnover Constraints” on page 5-15) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the `PortfolioCVaR` function or the `setOneWayTurnover` function. One-way turnover constraints depend upon an initial or current portfolio, which is assumed to be zero if not set when the turnover constraints are set. One-way turnover constraints have properties `BuyTurnover`, for the upper bound on net purchases, `SellTurnover`, for the upper bound on net sales, and `InitPort`, for the portfolio against which turnover is computed.

Setting One-Way Turnover Constraints Using the PortfolioCVaR Function

The Properties for the one-way turnover constraints are set using the `PortfolioCVaR` function. Suppose that you have an initial portfolio with 10 assets in a variable `x0` and you want to ensure that turnover on purchases is no more than 30% and turnover on sales is no more than 20% of the initial portfolio. To set these turnover constraints:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('BuyTurnover', 0.3, 'SellTurnover', 0.2, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
    10

    0.3000

    0.2000

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
```

```
    0.0800
    0.1000
```

If the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see "Setting Up an Initial or Current Portfolio" on page 5-41).

Setting Turnover Constraints Using the setOneWayTurnover Function

You can also set properties for portfolio turnover using `setOneWayTurnover` to specify to the upper bounds for turnover on purchases (`BuyTurnover`) and sales (`SellTurnover`) and an initial portfolio. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that turnover on purchases is no more than 30% and that turnover on sales is no more than 20% of the initial portfolio. Given a PortfolioCVaR object `p`, use `setOneWayTurnover` to set the turnover constraints with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);

disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
    10

    0.3000

    0.2000

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR;
p = setOneWayTurnover(p, 0.3, 0.2, x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
    10

    0.3000

    0.2000

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

`setOneWayTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the PortfolioCVaR object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setOneWayTurnover` lets you specify `NumAssets` as an optional argument. To remove one-way turnover from your PortfolioCVaR object, use the`PortfolioCVaR` function or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioCVaR` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63

- “Validate the CVaR Portfolio Problem” on page 5-96
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Validate the CVaR Portfolio Problem

In this section...

Sometimes, you may want to validate either your inputs to, or outputs from, a portfolio optimization problem. Although most error checking that occurs during the problem setup phase catches most difficulties with a portfolio optimization problem, the processes to validate CVaR portfolio sets and portfolios are time consuming and are best done offline. So, the portfolio optimization tools have specialized functions to validate CVaR portfolio sets and portfolios. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow” on page 5-20.

Validating a CVaR Portfolio Set

Since it is necessary and sufficient that your CVaR portfolio set must be a nonempty, closed, and bounded set to have a valid portfolio optimization problem, the `estimateBounds` function lets you examine your portfolio set to determine if it is nonempty and, if nonempty, whether it is bounded. Suppose that you have the following CVaR portfolio set which is an empty set because the initial portfolio at `0` is too far from a portfolio that satisfies the budget and turnover constraint:

```
p = PortfolioCVaR('NumAssets', 3, 'Budget', 1);
p = setTurnover(p, 0.3, 0);
```

If a CVaR portfolio set is empty, `estimateBounds` returns `NaN` bounds and sets the `isbounded` flag to `[]`:

```
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =

   NaN
   NaN
   NaN

ub =

   NaN
```

```
   NaN
   NaN

isbounded =

     []
```

Suppose that you create an unbounded CVaR portfolio set as follows:

```
p = PortfolioCVaR('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

  -Inf
  -Inf

ub =

  1.0e-008 *

   -0.3712
       Inf

isbounded =

     0
```

In this case, `estimateBounds` returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Finally, suppose that you created a CVaR portfolio set that is both nonempty and bounded. `estimateBounds` not only validates the set, but also obtains tighter bounds which are useful if you are concerned with the actual range of portfolio choices for individual assets in your portfolio set:

```
p = PortfolioCVaR;
p = setBudget(p, 1,1);
p = setBounds(p, [ -0.1; 0.2; 0.3; 0.2 ], [ 0.5; 0.3; 0.9; 0.8 ]);

[lb, ub, isbounded] = estimateBounds(p)

lb =

   -0.1000
```

```
    0.2000
    0.3000
    0.2000

ub =

    0.3000
    0.3000
    0.7000
    0.6000

isbounded =

     1
```

In this example, all but the second asset has tighter upper bounds than the input upper bound implies.

Validating CVaR Portfolios

Given a CVaR portfolio set specified in a PortfolioCVaR object, you often want to check if specific portfolios are feasible with respect to the portfolio set. This can occur with, for example, initial portfolios and with portfolios obtained from other procedures. The `checkFeasibility` function determines whether a collection of portfolios is feasible. Suppose that you perform the following portfolio optimization and want to determine if the resultant efficient portfolios are feasible relative to a modified problem.

First, set up a problem in the PortfolioCVaR object `p`, estimate efficient portfolios in `pwgt`, and then confirm that these portfolios are feasible relative to the initial problem:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)

ans =

     1     1     1     1     1     1     1     1     1     1
```

Next, set up a different portfolio problem that starts with the initial problem with an additional a turnover constraint and an equally weighted initial portfolio:

```
q = setTurnover(p, 0.3, 0.25);
checkFeasibility(q, pwgt)

ans =

     0     0     0     1     1     0     0     0     0     0
```

In this case, only two of the 10 efficient portfolios from the initial problem are feasible relative to the new problem in PortfolioCVaR object `q`. Solving the second problem using `checkFeasibility` demonstrates that the efficient portfolio for PortfolioCVaR object `q` is feasible relative to the initial problem:

```
qwgt = estimateFrontier(q);
checkFeasibility(p, qwgt)

ans =

     1     1     1     1     1     1     1     1     1     1
```

See Also

`PortfolioCVaR` | `checkFeasibility` | `estimateBounds`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101

- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object

There are two ways to look at a portfolio optimization problem that depends on what you are trying to do. One goal is to estimate efficient portfolios and the other is to estimate efficient frontiers. This section focuses on the former goal and "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119 focuses on the latter goal. For information on the workflow when using PortfolioCVaR objects, see "PortfolioCVaR Object Workflow" on page 5-20.

Obtaining Portfolios Along the Entire Efficient Frontier

The most basic way to obtain optimal portfolios is to obtain points over the entire range of the efficient frontier. Given a portfolio optimization problem in a PortfolioCVaR object, the `estimateFrontier` function computes efficient portfolios spaced evenly according to the return proxy from the minimum to maximum return efficient portfolios. The number of portfolios estimated is controlled by the hidden property `defaultNumPorts` which is set to `10`. A different value for the number of portfolios estimated is specified as input to `estimateFrontier`. This example shows the default number of efficient portfolios over the entire range of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);
disp(pwgt);
```

```
Columns 1 through 8

    0.8670    0.7046    0.5421    0.3825    0.2236    0.0570    0.0000    0.0000
    0.0413    0.1193    0.1963    0.2667    0.3392    0.4159    0.3392    0.1753
    0.0488    0.0640    0.0811    0.1012    0.1169    0.1427    0.1568    0.1754
    0.0429    0.1120    0.1806    0.2496    0.3203    0.3844    0.5040    0.6493

  Columns 9 through 10

    0.0000    0.0000
```

```
    0.0230    0.0000
    0.1777    0.0000
    0.7993    1.0000
```

If you want only four portfolios in the previous example:

```
pwgt = estimateFrontier(p, 4);

disp(pwgt);

0.8670    0.3825    0.0000    0.0000
0.0413    0.2667    0.3392    0.0000
0.0488    0.1012    0.1568    0.0000
0.0429    0.2496    0.5040    1.0000
```

Starting from the initial portfolio, `estimateFrontier` also returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

  Columns 1 through 8

    0.8670    0.7046    0.5421    0.3825    0.2236    0.0570    0.0000    0.0000
    0.0413    0.1193    0.1963    0.2667    0.3392    0.4159    0.3392    0.1753
    0.0488    0.0640    0.0811    0.1012    0.1169    0.1427    0.1568    0.1754
    0.0429    0.1120    0.1806    0.2496    0.3203    0.3844    0.5040    0.6493

  Columns 9 through 10

    0.0000    0.0000
    0.0230    0.0000
    0.1777    0.0000
    0.7993    1.0000


pbuy =

  Columns 1 through 8

    0.5670    0.4046    0.2421    0.0825         0         0         0         0
         0         0         0         0    0.0392    0.1159    0.0392         0
         0         0         0         0         0         0         0         0
         0    0.0120    0.0806    0.1496    0.2203    0.2844    0.4040    0.5493

  Columns 9 through 10

         0         0
         0         0
```

```
         0         0
    0.6993    0.9000


psell =

  Columns 1 through 8

         0         0         0         0    0.0764    0.2430    0.3000    0.3000
    0.2587    0.1807    0.1037    0.0333         0         0         0    0.1247
    0.1512    0.1360    0.1189    0.0988    0.0831    0.0573    0.0432    0.0246
    0.0571         0         0         0         0         0         0         0

  Columns 9 through 10

    0.3000    0.3000
    0.2770    0.3000
    0.0223    0.2000
         0         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Obtaining Endpoints of the Efficient Frontier” on page 5-105
- “Obtaining Efficient Portfolios for Target Returns” on page 5-108
- “Obtaining Efficient Portfolios for Target Risks” on page 5-112
- “Obtaining CVaR Portfolio Risks and Returns” on page 5-119
- “Obtaining Portfolio Standard Deviation and VaR” on page 5-121
- “Plotting the Efficient Frontier for a PortfolioCVaR Object” on page 5-123
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-137

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Obtaining Endpoints of the Efficient Frontier

Often, you might be interested in the endpoint portfolios for the efficient frontier. Suppose that you want to determine the range of returns from minimum to maximum to refine a search for a portfolio with a specific target return. Use the `estimateFrontierLimits` function to obtain the endpoint portfolios:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);
pwgt = estimateFrontierLimits(p);

disp(pwgt);

0.8646    0.0000
0.0470    0.0000
0.0414    0.0000
0.0470    1.0000
```

Note The endpoints of the efficient frontier depend upon the Scenarios in the PortfolioCVaR object. If you change the Scenarios, you are likely to obtain different endpoints.

Starting from an initial portfolio, `estimateFrontierLimits` also returns purchases and sales to get from the initial portfolio to the endpoint portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierLimits(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.8624    0.0000
    0.0513    0.0000
    0.0452    0.0000
    0.0411    1.0000


pbuy =

    0.5624         0
         0         0
         0         0
         0    0.9000


psell =

         0    0.3000
    0.2487    0.3000
    0.1548    0.2000
    0.0589         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Troubleshooting CVaR Portfolio Optimization Results" on page 5-137

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Obtaining Efficient Portfolios for Target Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` function accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. For example, assume that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 12%:

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

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierByReturn(p, [0.07,  0.10, .12]);
display(pwgt);

pwgt =

    0.7526    0.3773    0.1306
    0.1047    0.3079    0.4348
    0.0662    0.1097    0.1426
    0.0765    0.2051    0.2920
```

Sometimes, you can request a return for which no efficient portfolio exists. Based on the previous example, suppose that you want a portfolio with a 4% return (which is the return of the first asset). A portfolio that is fully invested in the first asset, however, is inefficient. `estimateFrontierByReturn` warns if your target returns are outside the range of efficient portfolio returns and replaces it with the endpoint portfolio of the efficient frontier closest to your target return:

```
 pwgt = estimateFrontierByReturn(p, [0.04]);

Warning: One or more target return values are outside the feasible range [
0.066388, 0.178834 ].
    Will return portfolios associated with endpoints of the range for these values.
> In PortfolioCVaR.estimateFrontierByReturn at 93
```

The best way to avoid this situation is to bracket your target portfolio returns with `estimateFrontierLimits` and `estimatePortReturn` (see "Obtaining Endpoints of the Efficient Frontier" on page 5-105 and "Obtaining CVaR Portfolio Risks and Returns" on page 5-119).

```
pret = estimatePortReturn(p, p.estimateFrontierLimits);

display(pret);
```

```
pret =

    0.0664
    0.1788
```

This result indicates that efficient portfolios have returns that range from 6.5% to 17.8%. Note, your results for these examples may be different due to the random generation of scenarios.

If you have an initial portfolio, `estimateFrontierByReturn` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, to obtain purchases and sales with target returns of 7%, 10%, and 12%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByReturn(p, [0.07,  0.10, .12]);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.7526    0.3773    0.1306
    0.1047    0.3079    0.4348
    0.0662    0.1097    0.1426
    0.0765    0.2051    0.2920


pbuy =

    0.4526    0.0773         0
         0    0.0079    0.1348
         0         0         0
         0    0.1051    0.1920
```

```
psell =

         0         0    0.1694
    0.1953         0         0
    0.1338    0.0903    0.0574
    0.0235         0         0
```

If you do not have an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-137

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)

- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Obtaining Efficient Portfolios for Target Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` function accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Suppose that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

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

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);
```

```
pwgt =

    0.3594    0.2524    0.1543
    0.3164    0.3721    0.4248
    0.1044    0.1193    0.1298
    0.2199    0.2563    0.2910
```

Sometimes, you can request a risk for which no efficient portfolio exists. Based on the previous example, suppose that you want a portfolio with 6% risk (individual assets in this universe have risks ranging from 7% to 42.5%). It turns out that a portfolio with 6% risk cannot be formed with these four assets. `estimateFrontierByRisk` warns if your target risks are outside the range of efficient portfolio risks and replaces it with the endpoint of the efficient frontier closest to your target risk:

```
pwgt = estimateFrontierByRisk(p, 0.06)
```

```
Warning: One or more target risk values are outside the feasible range [
0.0735749, 0.436667 ].
    Will return portfolios associated with endpoints of the range for these values.
```

```
> In PortfolioCVaR.estimateFrontierByRisk at 80

pwgt =

    0.7899
    0.0856
    0.0545
    0.0700
```

The best way to avoid this situation is to bracket your target portfolio risks with `estimateFrontierLimits` and `estimatePortRisk` (see "Obtaining Endpoints of the Efficient Frontier" on page 5-105 and "Obtaining CVaR Portfolio Risks and Returns" on page 5-119).

```
prsk = estimatePortRisk(p, p.estimateFrontierLimits);

display(prsk);

prsk =

    0.0736
    0.4367
```

This result indicates that efficient portfolios have risks that range from 7% to 42.5%. Note, your results for these examples may be different due to the random generation of scenarios.

Starting with an initial portfolio, `estimateFrontierByRisk` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the example with target risks of 12%, 14%, and 16%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

    0.3594    0.2524    0.1543
    0.3164    0.3721    0.4248
    0.1044    0.1193    0.1298
    0.2199    0.2563    0.2910
```

```
pbuy =

    0.0594         0         0
    0.0164    0.0721    0.1248
         0         0         0
    0.1199    0.1563    0.1910


psell =

         0    0.0476    0.1457
         0         0         0
    0.0956    0.0807    0.0702
         0         0         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-137

More About

- “PortfolioCVaR Object” on page 5-22

- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Choosing and Controlling the Solver

When solving portfolio optimizations for a PortfolioCVaR object, all variations of `fmincon` from Optimization Toolbox are supported. Alternatively, you can use `'cuttingplane'`, a solver that implements Kelley's cutting plane method (see Kelley [45] at "Portfolio Optimization" on page A-7).

Unlike Optimization Toolbox which uses the `interior-point` algorithm as the default algorithm for `fmincon`, the portfolio optimization for a PortfolioCVaR object uses the `sqp` algorithm. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see "Constrained Nonlinear Optimization Algorithms" (Optimization Toolbox).

To modify `fmincon` options for CVaR portfolio optimizations, use `setSolver` to set the hidden properties `solverType` and `solverOptions` to specify and control the solver. (Note that you can see the default options by creating a dummy`PortfolioCVaR` object, using `p = PortfolioCVaR` and then type `p.solverOptions`.) Since these solver properties are hidden, you cannot set them using the `PortfolioCVaR` function. The default solver is `fmincon` with the `sqp` algorithm objective function, gradients turned on, and no displayed output, so you do not need to use `setSolver` to specify this.

If you want to specify additional options associated with the `fmincon` solver, `setSolver` accepts these options as name-value pair arguments. For example, if you want to use `fmincon` with the `trust-region-reflective` algorithm and with no displayed output, use `setSolver` with:

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'on');
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
on
```

Alternatively, `setSolver` accepts an `optimoptions` object as the second argument. For example, you can change the algorithm to `trust-region-reflective` with no displayed output as follows:

```
p = PortfolioCVaR;
options = optimoptions('fmincon','Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

The `'cuttingplane'` solver has options to control the number iterations and stopping tolerances. Moreover, this solver uses `linprog` as the master solver, and all `linprog` options are supported using `optimoptions` structures. All these options are set using `setSolver`.

For example, you can use `setSolver` to increase the number of iterations for `'cuttingplane'`:

```
p = PortfolioCVaR;
p = setSolver(p, 'cuttingplane', 'MaxIter', 2000);
display(p.solverType);
display(p.solverOptions);
```

```
cuttingplane
                MaxIter: 2000
                 AbsTol: 1.0000e-06
                 RelTol: 1.0000e-05
    MasterSolverOptions: [1x1 optim.options.Linprog]
```

To change the master solver algorithm to`'interior-point'`, with no display, use `setSolver` to modify `'MasterSolverOptions'`:

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','interior-point','Display','off');
p = setSolver(p,'cuttingplane','MasterSolverOptions',options);
display(p.solverType)
display(p.solverOptions)
display(p.solverOptions.MasterSolverOptions.Algorithm)
display(p.solverOptions.MasterSolverOptions.Display)
```

```
cuttingplane
                MaxIter: 1000
                 AbsTol: 1.0000e-06
                 RelTol: 1.0000e-05
    MasterSolverOptions: [1x1 optim.options.Linprog]

interior-point
off
```

See Also

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` |

`estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-137

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Estimate Efficient Frontiers for PortfolioCVaR Object

In this section...

Whereas “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101 focused on estimation of efficient portfolios, this section focuses on the estimation of efficient frontiers. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow” on page 5-20.

Obtaining CVaR Portfolio Risks and Returns

Given any portfolio and, in particular, efficient portfolios, the functions `estimatePortReturn` and `estimatePortRisk` provide estimates for the return (or return proxy), risk (or the risk proxy). Each function has the same input syntax but with different combinations of outputs. Suppose that you have this following portfolio optimization problem that gave you a collection of portfolios along the efficient frontier in `pwgt`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = setInitPort(p, pwgt0);
pwgt = estimateFrontier(p);
```

Note Remember that the risk proxy for CVaR portfolio optimization is CVaR.

Given `pwgt0` and `pwgt`, use the portfolio risk and return estimation functions to obtain risks and returns for your initial portfolio and the portfolios on the efficient frontier:

```
prsk0 = estimatePortRisk(p, pwgt0);
pret0 = estimatePortReturn(p, pwgt0);
prsk = estimatePortRisk(p, pwgt);
pret = estimatePortReturn(p, pwgt);
```

You obtain these risks and returns:

```
display(prsk0);
display(pret0);
display(prsk);
display(pret);
```

```
prsk0 =

    0.0591


pret0 =

    0.0067


prsk =

    0.0414
    0.0453
    0.0553
    0.0689
    0.0843
    0.1006
    0.1193
    0.1426
    0.1689
    0.1969


pret =

    0.0050
```

```
    0.0060
    0.0070
    0.0080
    0.0089
    0.0099
    0.0109
    0.0119
    0.0129
    0.0139
```

Obtaining Portfolio Standard Deviation and VaR

The PortfolioCVaR object has functions to compute standard deviations of portfolio returns and the value-at-risk of portfolios with the functions `estimatePortStd` and `estimatePortVaR`. These functions work with any portfolios, not necessarily efficient portfolios. For example, the following example obtains five portfolios (`pwgt`) on the efficient frontier and also has an initial portfolio in `pwgt0`. Various portfolio statistics are computed that include the return, risk, standard deviation, and value-at-risk. The listed estimates are for the initial portfolio in the first row followed by estimates for each of the five efficient portfolios in subsequent rows.

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR('initport', pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

pwgt = estimateFrontier(p, 5);

pret = estimatePortReturn(p, [pwgt0, pwgt]);
prsk = estimatePortRisk(p, [pwgt0, pwgt]);
pstd = estimatePortStd(p, [pwgt0, pwgt]);
pvar = estimatePortVaR(p, [pwgt0, pwgt]);

[pret, prsk, pstd, pvar]
```

```
ans =

    0.0207    0.0464    0.0381    0.0283
    0.1009    0.0214    0.0699   -0.0109
    0.1133    0.0217    0.0772   -0.0137
    0.1256    0.0226    0.0849   -0.0164
    0.1380    0.0240    0.0928   -0.0182
    0.1503    0.0262    0.1011   -0.0197
```

See Also

`PortfolioCVaR` | `estimatePortReturn` | `estimatePortStd` | `estimatePortVaR` | `plotFrontier`

Related Examples

- “Plotting the Efficient Frontier for a PortfolioCVaR Object” on page 5-123
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Postprocessing Results to Set Up Tradable Portfolios” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

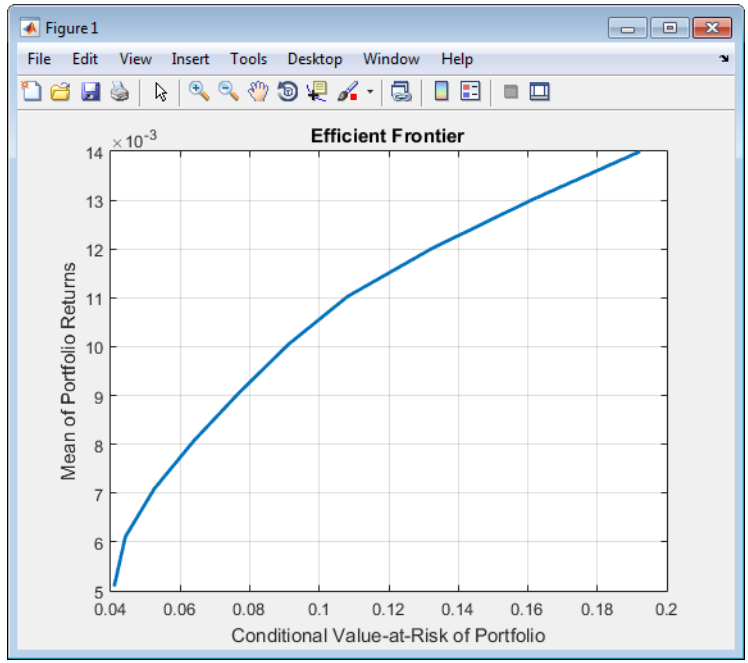
External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Plotting the Efficient Frontier for a PortfolioCVaR Object

The `plotFrontier` function creates a plot of the efficient frontier for a given portfolio optimization problem. This function accepts several types of inputs and generates a plot with an optional possibility to output the estimates for portfolio risks and returns along the efficient frontier. `plotFrontier` has four different ways that it can be used. In addition to a plot of the efficient frontier, if you have an initial portfolio in the `InitPort` property, `plotFrontier` also displays the return versus risk of the initial portfolio on the same plot. If you have a well-posed portfolio optimization problem set up in a PortfolioCVaR object and you use `plotFrontier`, you get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is currently `10` and is maintained in the hidden property `defaultNumPorts`). This example illustrates a typical use of `plotFrontier` to create a new plot:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```

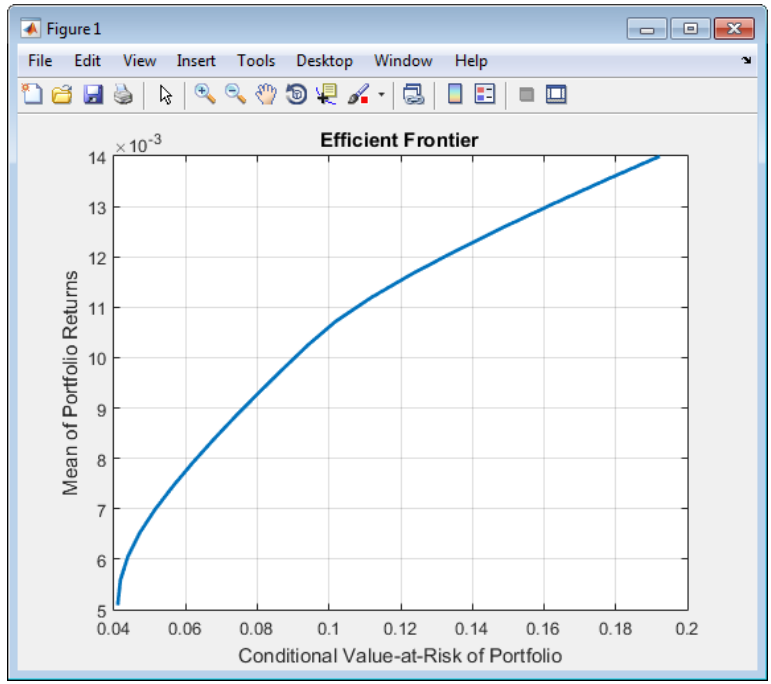


The `Name` property appears as the title of the efficient frontier plot if you set it in the PortfolioCVaR object. Without an explicit name, the title on the plot would be "Efficient Frontier." If you want to obtain a specific number of portfolios along the efficient frontier, use `plotFrontier` with the number of portfolios that you want. Suppose that you have the PortfolioCVaR object from the previous example and you want to plot 20 portfolios along the efficient frontier and to obtain 20 risk and return values for each portfolio:

```
[prsk, pret] = plotFrontier(p, 20);
display([pret, prsk]);
```

```
ans =

    0.0051    0.0406
    0.0056    0.0414
    0.0061    0.0437
    0.0066    0.0471
    0.0071    0.0515
    0.0076    0.0567
    0.0082    0.0624
    0.0087    0.0687
```

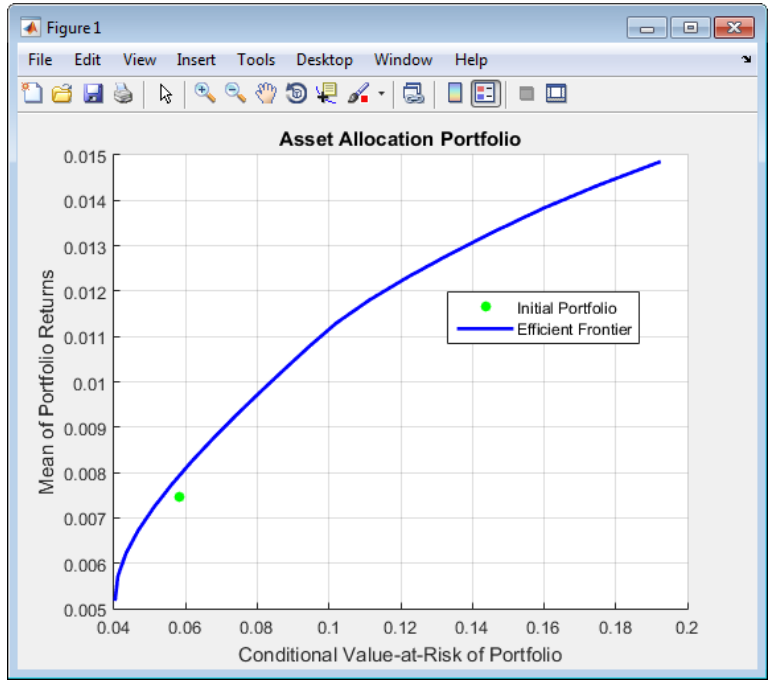
```
    0.0092    0.0753
    0.0097    0.0821
    0.0102    0.0891
    0.0107    0.0962
    0.0112    0.1044
    0.0117    0.1142
    0.0122    0.1251
    0.0127    0.1369
    0.0133    0.1496
    0.0138    0.1628
    0.0143    0.1766
    0.0148    0.1907
```



Plotting Existing Efficient Portfolios

If you already have efficient portfolios from any of the "estimateFrontier" functions (see “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101), pass them into `plotFrontier` directly to plot the efficient frontier:

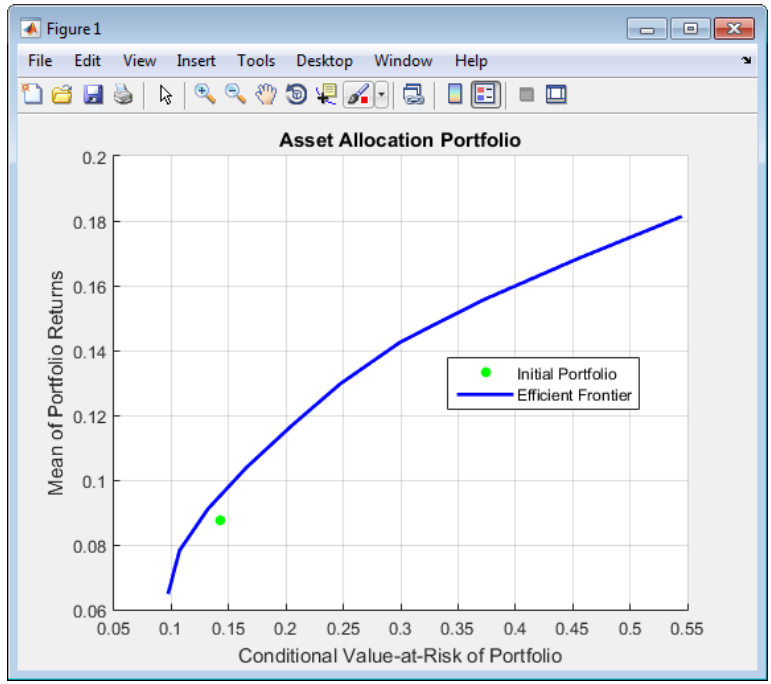
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p, 20);
plotFrontier(p, pwgt);
```



Plotting Existing Efficient Portfolio Risks and Returns

If you already have efficient portfolio risks and returns, you can use the interface to `plotFrontier` to pass them into `plotFrontier` to obtain a plot of the efficient frontier:

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

AssetScenarios = mvnrnd(m, C, 20000);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);

pret= estimatePortReturn(p, pwgt);
prsk = estimatePortRisk(p, pwgt);

plotFrontier(p, prsk, pret);
```



See Also

`PortfolioCVaR` | `estimatePortReturn` | `estimatePortStd` | `estimatePortVaR` | `plotFrontier`

Related Examples

- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Postprocessing Results to Set Up Tradable Portfolios” on page 5-130

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Postprocessing Results to Set Up Tradable Portfolios

After obtaining efficient portfolios or estimates for expected portfolio risks and returns, use your results to set up trades to move toward an efficient portfolio. For information on the workflow when using PortfolioCVaR objects, see "PortfolioCVaR Object Workflow" on page 5-20.

Setting Up Tradable Portfolios

Suppose that you set up a portfolio optimization problem and obtained portfolios on the efficient frontier. Use the `dataset` object from Statistics and Machine Learning Toolbox to form a blotter that lists your portfolios with the names for each asset. For example, suppose that you want to obtain five portfolios along the efficient frontier. You can set up a blotter with weights multiplied by 100 to view the allocations for each portfolio:

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

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{100*pwgt},pnames],'obsnames',p.AssetList);
display(Blotter);

Blotter =

                        Port1     Port2     Port3     Port4     Port5
    Bonds                78.84    43.688    8.3448         0    1.2501e-12
    Large-Cap Equities  9.3338    29.131    48.467    23.602    9.4219e-13
    Small-Cap Equities  4.8843    8.1284    12.419    16.357     8.281e-14
    Emerging Equities   6.9419    19.053    30.769    60.041           100
```

Note Your results may differ from this result due to the simulation of scenarios.

This result indicates that you would invest primarily in bonds at the minimum-risk/minimum-return end of the efficient frontier (`Port1`), and that you would invest completely in emerging equity at the maximum-risk/maximum-return end of the efficient frontier (`Port5`). You can also select a particular efficient portfolio, for example, suppose that you want a portfolio with 15% risk and you add purchase and sale weights outputs obtained from the "estimateFrontier" functions to set up a trade blotter:

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

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');

p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

[pwgt, pbuy, psell] = estimateFrontierByRisk(p, 0.15);

Blotter = dataset([{100*[pwgt0, pwgt, pbuy, psell]}, ...
{'Initial','Weight', 'Purchases','Sales'}],'obsnames',p.AssetList);
display(Blotter);

Blotter =

                        Initial    Weight    Purchases    Sales
    Bonds               30         15.036          0       14.964
    Large-Cap Equities  30         45.357    15.357            0
    Small-Cap Equities  20         12.102          0       7.8982
    Emerging Equities   10         27.505    17.505            0
```

If you have prices for each asset (in this example, they can be ETFs), add them to your blotter and then use the tools of the `dataset` object to obtain shares and shares to be traded.

See Also

`PortfolioCVaR` | `checkFeasibility` | `estimateScenarioMoments`

Related Examples

- "Troubleshooting CVaR Portfolio Optimization Results" on page 5-137
- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119

More About

- “PortfolioCVaR Object” on page 5-22
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Working with Other Portfolio Objects

The PortfolioCVaR object is for CVaR portfolio optimization. The Portfolio object is for mean-variance portfolio optimization. Sometimes, you might want to examine portfolio optimization problems according to different combinations of return and risk proxies. A common example is that you want to do a CVaR portfolio optimization and then want to work primarily with moments of portfolio returns. Suppose that you set up a CVaR portfolio optimization problem with:

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

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioCVaR;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);
```

To work with the same problem in a mean-variance framework, you can use the scenarios from the PortfolioCVaR object to set up a Portfolio object so that `p` contains a CVaR optimization problem and `q` contains a mean-variance optimization problem based on the same data.

```
q = Portfolio('AssetList', p.AssetList);
q = estimateAssetMoments(q, p.getScenarios);
q = setDefaultConstraints(q);

pwgt = estimateFrontier(p);
qwgt = estimateFrontier(q);
```

Since each object has a different risk proxy, it is not possible to compare results side by side. To obtain means and standard deviations of portfolio returns, you can use the functions associated with each object to obtain:

```
pret = estimatePortReturn(p, pwgt);
pstd = estimatePortStd(p, pwgt);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);

[pret, qret]
[pstd, qstd]
```

```
ans =

    0.0665    0.0585
    0.0787    0.0716
    0.0910    0.0848
    0.1033    0.0979
    0.1155    0.1111
    0.1278    0.1243
    0.1401    0.1374
    0.1523    0.1506
    0.1646    0.1637
    0.1769    0.1769


ans =

    0.0797    0.0774
    0.0912    0.0835
    0.1095    0.0995
    0.1317    0.1217
    0.1563    0.1472
    0.1823    0.1746
    0.2135    0.2059
    0.2534    0.2472
    0.2985    0.2951
    0.3499    0.3499
```

To produce comparable results, you can use the returns or risks from one portfolio optimization as target returns or risks for the other portfolio optimization.

```
qwgt = estimateFrontierByReturn(q, pret);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);

[pret, qret]
[pstd, qstd]

ans =

    0.0665    0.0665
    0.0787    0.0787
    0.0910    0.0910
    0.1033    0.1033
    0.1155    0.1155
    0.1278    0.1278
```

```
    0.1401    0.1401
    0.1523    0.1523
    0.1646    0.1646
    0.1769    0.1769


ans =

    0.0797    0.0797
    0.0912    0.0912
    0.1095    0.1095
    0.1317    0.1317
    0.1563    0.1563
    0.1823    0.1823
    0.2135    0.2135
    0.2534    0.2534
    0.2985    0.2985
    0.3499    0.3499
```

Now it is possible to compare standard deviations of portfolio returns from either type of portfolio optimization.

See Also

`Portfolio` | `PortfolioCVaR`

Related Examples

- “Creating the Portfolio Object” on page 4-28
- “Creating the PortfolioCVaR Object” on page 5-27
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119

More About

- “PortfolioCVaR Object” on page 5-22

- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-20
- “Portfolio Object Workflow” on page 4-21

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)
- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Troubleshooting CVaR Portfolio Optimization Results

PortfolioCVaR Object Destroyed When Modifying

If a PortfolioCVaR object is destroyed when modifying, remember to pass an existing object into the `PortfolioCVaR` function if you want to modify it, otherwise it creates a new object. See “Creating the PortfolioCVaR Object” on page 5-27 for details.

Matrix Incompatibility and "Non-Conformable" Errors

If you get matrix incompatibility or "non-conformable" errors, the representation of data in the tools follows a specific set of basic rules described in “Conventions for Representation of Data” on page 5-25.

CVaR Portfolio Optimization Warns About “Max Iterations”

If the `'cuttingplane'` solver displays the following warning:

```
Warning: Max iterations reached. Consider modifying the solver options, or using fmincon.
> In @PortfolioCVaR\private\cvar_cuttingplane_solver at 255
  In @PortfolioCVaR\private\cvar_optim_min_risk at 85
  In PortfolioCVaR.estimateFrontier at 69
```

this warning indicates that some of the reported efficient portfolios may not be accurate enough.

This warning is usually related to portfolios in the lower-left end of the efficient frontier. The cutting plane solver may have gotten very close to the solution, but there may be too many portfolios with very similar risks and returns in that neighborhood, and the solver runs out of iterations before reaching the desired accuracy.

To correct this problem, you can use `setSolver` to make any of these changes:

- Increase the maximum number of iterations (`'MaxIter'`).
- Relax the stopping tolerances (`'AbsTol'` and/or `'RelTol'`).
- Use a different master solver algorithm (`'MasterSolverOptions'`).
- Alternatively, you can try the `'fmincon'` solver.

When the default maximum number of iterations of the `'cuttingplane'` solver is reached, the solver usually needs many more iterations to reach the accuracy required by

the default stopping tolerances. You may want to combine increasing the number of iterations (e.g., multiply by 5) with relaxing the stopping tolerances (e.g., multiply by 10 or 100). Since the CVaR is a stochastic optimization problem, the accuracy of the solution is relative to the scenario sample, so a looser stopping tolerance may be acceptable. Keep in mind that the solution time may increase significantly when you increase the number of iterations. For example, doubling the number of iterations more than doubles the solution time. Sometimes using a different master solver (e.g., switching to `'interior-point'` if you are using the default `'simplex'`) can get the `'cuttingplane'` solver to converge without changing the maximum number of iterations.

Alternatively, the `'fmincon'` solver may be faster than the `'cuttingplane'` solver for problems where cutting plane reaches the maximum number of iterations.

CVaR Portfolio Optimization Errors with “Could Not Solve” Message

If the `'cuttingplane'` solver generates the following error:

```
Error using cvar_cuttingplane_solver (line 251)
Could not solve the problem. Consider modifying the solver options, or using fmincon.

Error in cvar_optim_by_return (line 100)
        [x,~,~,exitflag] = cvar_cuttingplane_solver(...

Error in PortfolioCVaR/estimateFrontier (line 80)
    pwgt = cvar_optim_by_return(obj, r(2:end-1), obj.NumAssets, ...
```

this error means that the master solver failed to solve one of the master problems. The error may be due to numerical instability or other problem-specific situation.

To correct this problem, you can use `setSolver` to make any of these changes:

- Modify the master solver options (`'MasterSolverOptions'`), for example, change the algorithm (`'Algorithm'`) or the termination tolerance (`'TolFun'`).
- Alternatively, you can try the `'fmincon'` solver.

Missing Data Estimation Fails

If asset return data has missing or `NaN` values, the `simulateNormalScenariosByData` function with the `'missingdata'` flag set to `true` may fail with either too many iterations or a singular covariance. To correct this problem, consider this:

- If you have asset return data with no missing or `NaN` values, you can compute a covariance matrix that may be singular without difficulties. If you have missing or

`NaN` values in your data, the supported missing data feature requires that your covariance matrix must be positive-definite, that is, nonsingular.

- `simulateNormalScenariosByData` uses default settings for the missing data estimation procedure that might not be appropriate for all problems.

In either case, you might want to estimate the moments of asset returns separately with either the ECM estimation functions such as `ecmnmle` or with your own functions.

cvar_optim_transform Errors

If you obtain optimization errors such as:

```
Error using cvar_optim_transform (line 276)
Portfolio set appears to be either empty or unbounded. Check constraints.

Error in PortfolioCVaR/estimateFrontier (line 64)
    [AI, bI, AE, bE, lB, uB, f0, f, x0] = cvar_optim_transform(obj);
```

or

```
Error using cvar_optim_transform (line 281)
Cannot obtain finite lower bounds for specified portfolio set.

Error in PortfolioCVaR/estimateFrontier (line 64)
    [AI, bI, AE, bE, lB, uB, f0, f, x0] = cvar_optim_transform(obj);
```

Since the portfolio optimization tools require a bounded portfolio set, these errors (and similar errors) can occur if your portfolio set is either empty and, if nonempty, unbounded. Specifically, the portfolio optimization algorithm requires that your portfolio set have at least a finite lower bound. The best way to deal with these problems is to use the validation functions in “Validate the CVaR Portfolio Problem” on page 5-96. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

If you obtain efficient portfolios that, do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between `0` and `1` and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the PortfolioCVaR object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

See Also

`PortfolioCVaR` | `checkFeasibility` | `estimateScenarioMoments`

Related Examples

- "Postprocessing Results to Set Up Tradable Portfolios" on page 5-130
- "Creating the PortfolioCVaR Object" on page 5-27
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119

More About

- "PortfolioCVaR Object" on page 5-22
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-20

External Websites

- CVaR Portfolio Optimization (5 min 33 sec)

- Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

MAD Portfolio Optimization Tools

Portfolio Optimization Theory

In this section...

Portfolio Optimization Problems

Portfolio optimization problems involve identifying portfolios that satisfy three criteria:

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

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

The set of feasible portfolios is necessarily a nonempty, closed, and bounded set. The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices. The proxy for return is a function that characterizes either the gross or net benefits associated with portfolio choices. The terms “risk” and “risk proxy” and “return” and “return proxy” are interchangeable. The fundamental insight of Markowitz (see “Portfolio Optimization” on page A-7) is that the goal of the portfolio choice problem is to seek minimum risk for a given level of return and to seek maximum return for a given level of risk. Portfolios satisfying these criteria are efficient portfolios and the graph of the risks and returns of these portfolios forms a curve called the efficient frontier.

Portfolio Problem Specification

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

- Proxy for portfolio return (μ)

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

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

- The Portfolio object (`Portfolio`) supports mean-variance portfolio optimization (see Markowitz [46], [47] at "Portfolio Optimization" on page A-7). This object has either gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set.
- The PortfolioCVaR object (`PortfolioCVaR`) implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at "Portfolio Optimization" on page A-7), which is generally referred to as CVaR portfolio optimization. CVaR portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses conditional value-at-risk of portfolio returns as the risk proxy.
- The PortfolioMAD object (`PortfolioMAD`) implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7), which is generally referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

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

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

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

and (sample) covariance of asset returns

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

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

Gross Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

Net Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

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

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

Risk Proxy

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

Variance

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

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

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

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

Although the risk proxy in mean-variance portfolio optimization is the variance of portfolio returns, the square root, which is the standard deviation of portfolio returns, is often reported and displayed. Moreover, this quantity is often called the "risk" of the portfolio. For details, see Markowitz [46], [47] at ("Portfolio Optimization" on page A-7).

Conditional Value-at-Risk

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

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

where:

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

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

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

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

The value-at-risk is defined as

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

An alternative formulation for CVaR has the form:

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

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

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

To describe the probability distribution of returns, the `PortfolioCVaR` object takes a finite sample of return scenarios y_s, with $s = 1,\ldots,S$. Each y_s is an n vector that contains the returns for each of the n assets under the scenario s. This sample of S scenarios is stored as a scenario matrix of size S-by-n. Then, the risk proxy for CVaR portfolio optimization, for a given portfolio $x \in X$ and $\alpha \in (0,1)$, is computed as

$$\sum(x) = VaR_\alpha(x) + \frac{1}{(1-\alpha)S}\sum_{s=1}^{S} \max\left\{0, -y_s^T x - VaR_\alpha(x)\right\}$$

The value-at-risk, $VaR_\alpha(x)$, is estimated whenever the CVaR is estimated. The loss function is $f(x,y_s) = -y_s^T x$, which is the portfolio loss under scenario s.

Under this definition, VaR and CVaR are sample estimators for VaR and CVaR based on the given scenarios. Better scenario samples yield more reliable estimates of VaR and CVaR.

For more information, see Rockafellar and Uryasev [48], [49], and Cornuejols and Tütüncü, [51], at "Portfolio Optimization" on page A-7.

Mean Absolute-Deviation

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

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

where:

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

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

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

such that

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

For more information, see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7.

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10
- "Default Portfolio Problem" on page 6-18
- "PortfolioMAD Object Workflow" on page 6-19

Portfolio Set for Optimization Using PortfolioMAD Object

The final element for a complete specification of a portfolio optimization problem is the

set of feasible portfolios, which is called a portfolio set. A portfolio set $X \subset R^n$ is specified by construction as the intersection of sets formed by a collection of constraints on portfolio weights. A portfolio set necessarily and sufficiently must be a nonempty, closed, and bounded set.

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

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

Linear Inequality Constraints

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

$$A_I x \le b_I$$

where:

x is the portfolio (n vector).

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

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

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

Portfolio object properties to specify linear inequality constraints are:

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

The default is to ignore these constraints.

Linear Equality Constraints

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

$$A_E x = b_E$$

where:

x is the portfolio (n vector).

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

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

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

Portfolio object properties to specify linear equality constraints are:

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

The default is to ignore these constraints.

Bound Constraints

Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Since every portfolio set must be bounded, it is

often a good practice, albeit not necessary, to set explicit bounds for the portfolio problem. To obtain explicit bounds for a given portfolio set, use the `estimateBounds` function. Bound constraints take the form

$$l_B \le x \le u_B$$

where:

x is the portfolio (n vector).

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

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

n is the number of assets in the universe.

Portfolio object properties to specify bound constraints are:

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

The default is to ignore these constraints.

The default portfolio optimization problem (see "Default Portfolio Problem" on page 6-18) has $l_B = 0$ with u_B set implicitly through a budget constraint.

Budget Constraints

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

$$l_S \le 1^T x \le u_S$$

where:

x is the portfolio (n vector).

`1` is the vector of ones (n vector).

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

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

n is the number of assets in the universe.

Portfolio object properties to specify budget constraints are:

- `LowerBudget` for l_S
- `UpperBudget` for u_S
- `NumAssets` for n

The default is to ignore this constraint.

The default portfolio optimization problem (see "Default Portfolio Problem" on page 6-18) has $l_S = u_S = 1$, which means that the portfolio weights sum to `1`. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0–100% invested in cash. If cash is to be a portfolio choice, set `RiskFreeRate` (r_0) to a suitable value (see "Return Proxy" on page 6-4 and "Working with a Riskless Asset" on page 6-54).

Group Constraints

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

$$l_G \le Gx \le u_G$$

where:

x is the portfolio (n vector).

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

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

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

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

Portfolio object properties to specify group constraints are:

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

The default is to ignore these constraints.

Group Ratio Constraints

Group ratio constraints are specialized linear constraints that enforce relationships among groups of assets. The constraints take the form
$l_{Ri}(G_B x)_i \le (G_A x)_i \le u_{Ri}(G_B x)_i$

for $i = 1,..., n_R$ where:

x is the portfolio (n vector).

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

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

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

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

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

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

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

Portfolio object properties to specify group ratio constraints are:

- `GroupA` for G_A
- `GroupB` for G_B
- `LowerRatio` for l_R

- `UpperRatio` for u_R
- `NumAssets` for n

The default is to ignore these constraints.

Average Turnover Constraints

Turnover constraint is a linear absolute value constraint that ensures estimated optimal portfolios differ from an initial portfolio by no more than a specified amount. Although portfolio turnover is defined in many ways, the turnover constraints implemented in Financial Toolbox computes portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

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

where:

x is the portfolio (n vector).

`1` is the vector of ones (n vector).

x_0 is the initial portfolio (n vector).

τ is the upper bound for turnover (scalar).

n is the number of assets in the universe.

Portfolio object properties to specify the average turnover constraint are:

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

The default is to ignore this constraint.

One-way Turnover Constraints

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

$$1^T \times \max\{0, x - x_0\} \le \tau_B$$

$$1^T \times \max\{0, x_0 - x\} \le \tau_S$$

where:

x is the portfolio (n vector)

`1` is the vector of ones (n vector).

x_0 is the Initial portfolio (n vector).

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

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

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

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

The default is to ignore this constraint.

Note The average turnover constraint (see "Working with Average Turnover Constraints Using PortfolioMAD Object" on page 6-85) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61

More About

- "PortfolioMAD Object" on page 6-21
- "Default Portfolio Problem" on page 6-18
- "PortfolioMAD Object Workflow" on page 6-19

Default Portfolio Problem

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

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

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10
- "PortfolioMAD Object Workflow" on page 6-19

PortfolioMAD Object Workflow

The PortfolioMAD object workflow for creating and modeling a MAD portfolio is:

1. Create a MAD Portfolio.

   Create a `PortfolioMAD` object for mean-absolute deviation (MAD) portfolio optimization. “Creating the PortfolioMAD Object” on page 6-26.

2. Define asset returns and scenarios.

   Evaluate scenarios for portfolio asset returns, including assets with missing data and financial time series data. For more information, see “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42.

3. Specify the MAD Portfolio Constraints.

   Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, and turnover constraints. For more information, see “Working with MAD Portfolio Constraints Using Defaults” on page 6-61.

4. Validate the MAD Portfolio.

   Identify errors for the portfolio specification. For more information, see “Validate the MAD Portfolio Problem” on page 6-91.

5. Estimate the efficient portfolios and frontiers.

   Analyze the efficient portfolios and efficient frontiers for a portfolio. For more information, see “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96 and “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111.

6. Postprocess the results.

   Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see “Postprocessing Results to Set Up Tradable Portfolios” on page 6-122.

See Also

More About

- “Portfolio Optimization Theory” on page 6-3

PortfolioMAD Object

In this section...

PortfolioMAD Object Properties and Functions

The PortfolioMAD object implements mean absolute-deviation (MAD) portfolio optimization and is derived from the abstract class `AbstractPortfolio`. Every property and function of the PortfolioMAD object is public, although some properties and functions are hidden. The PortfolioMAD object is a value object where every instance of the object is a distinct version of the object. Since the PortfolioMAD object is also a MATLAB object, it inherits the default functions associated with MATLAB objects.

Working with PortfolioMAD Objects

The PortfolioMAD object and its functions are an interface for mean absolute-deviation portfolio optimization. So, almost everything you do with the PortfolioMAD object can be done using the functions. The basic workflow is:

1. Design your portfolio problem.
2. Use the `PortfolioMAD` function to create the PortfolioMAD object or use the various set functions to set up your portfolio problem.
3. Use estimate functions to solve your portfolio problem.

In addition, functions are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a PortfolioMAD object, you can

save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of MAD portfolio optimization, means that you have either scenarios, data, or moments for asset returns, and a collection of constraints on your portfolios, use the `PortfolioMAD` function to set the properties for the PortfolioMAD object.

The `PortfolioMAD` function lets you create an object from scratch or update an existing object. Since the PortfolioMAD object is a value object, it is easy to create a basic object, then use functions to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see "Creating the PortfolioMAD Object" on page 6-26.

Setting and Getting Properties

You can set properties of a PortfolioMAD object using either the`PortfolioMAD` function or various set functions.

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

The `PortfolioMAD` function supports setting properties with name-value pair arguments such that each argument name is a property and each value is the value to assign to that property. For example, to set the `LowerBound` and `Budget` properties in an existing PortfolioMAD object `p`, use the syntax:

```
p = PortfolioMAD(p,'LowerBound', 0,'Budget',1);
```

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

- The `PortfolioMAD` function and set functions try to determine the dimensions of your problem with either explicit or implicit inputs.
- The `PortfolioMAD` function and set functions try to resolve ambiguities with default choices.

- The `PortfolioMAD` function and set functions perform scalar expansion on arrays when possible.
- The PortfolioMAD functions try to diagnose and warn about problems.

Displaying PortfolioMAD Objects

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

Saving and Loading PortfolioMAD Objects

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

Estimating Efficient Portfolios and Frontiers

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

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

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

Arrays of PortfolioMAD Objects

Although all functions associated with a PortfolioMAD object are designed to work on a scalar PortfolioMAD object, the array capabilities of MATLAB enables you to set up and

work with arrays of PortfolioMAD objects. The easiest way to do this is with the `repmat` function. For example, to create a 3-by-2 array of PortfolioMAD objects:

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

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

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

This example calls the `PortfolioMAD` function for the (`i`,`j`) element of a matrix of PortfolioMAD objects in the variable `p`.

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

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

and, once set, you can access these bounds with

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

PortfolioMAD object functions work on only one PortfolioMAD object at a time.

Subclassing PortfolioMAD Objects

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

Conventions for Representation of Data

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

- Asset returns or prices for scenarios are in matrix form with samples for a given asset going down the rows and assets going across the columns. In the case of prices, the earliest dates must be at the top of the matrix, with increasing dates going down.
- Portfolios are in vector or matrix form with weights for a given portfolio going down the rows and distinct portfolios going across the columns.
- Constraints on portfolios are formed in such a way that a portfolio is a column vector.
- Portfolio risks and returns are either scalars or column vectors (for multiple portfolio risks and returns).

See Also

`PortfolioMAD`

Related Examples

- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61

More About

- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Creating the PortfolioMAD Object

In this section...

To create a fully specified MAD portfolio optimization problem, instantiate the PortfolioMAD object using the `PortfolioMAD` function. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-19.

Syntax

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

```
p = PortfolioMAD;
```

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

The `PortfolioMAD` function also accepts collections of argument name-value pair arguments for properties and their values. The `PortfolioMAD` function accepts inputs for public properties with the general syntax:

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

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

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

Input argument names are not case-sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 6-31). The `PortfolioMAD` function tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process to

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

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

PortfolioMAD Problem Sufficiency

A MAD portfolio optimization problem is completely specified with the PortfolioMAD object if the following three conditions are met:

- You must specify a collection of asset returns or prices known as scenarios such that all scenarios are finite asset returns or prices. These scenarios are meant to be samples from the underlying probability distribution of asset returns. This condition can be satisfied by the `setScenarios` function or with several canned scenario simulation functions.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded. You can satisfy this condition using an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed and several tools, such as the `estimateBounds` function, provide ways to ensure that your problem is properly formulated.

  Although the general sufficient conditions for MAD portfolio optimization go beyond these conditions, the PortfolioMAD object handles all these additional conditions.

PortfolioMAD Function Examples

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

```
p = PortfolioMAD;
disp(p);
```

```
 PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
```

```
    NumScenarios: []
            Name: []
       NumAssets: []
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: []
      UpperBound: []
     LowerBudget: []
     UpperBudget: []
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the `PortfolioMAD` function. The custom set functions offer additional ways to set and modify collections of properties in the PortfolioMAD object.

Using the PortfolioMAD Function for a Single-Step Setup

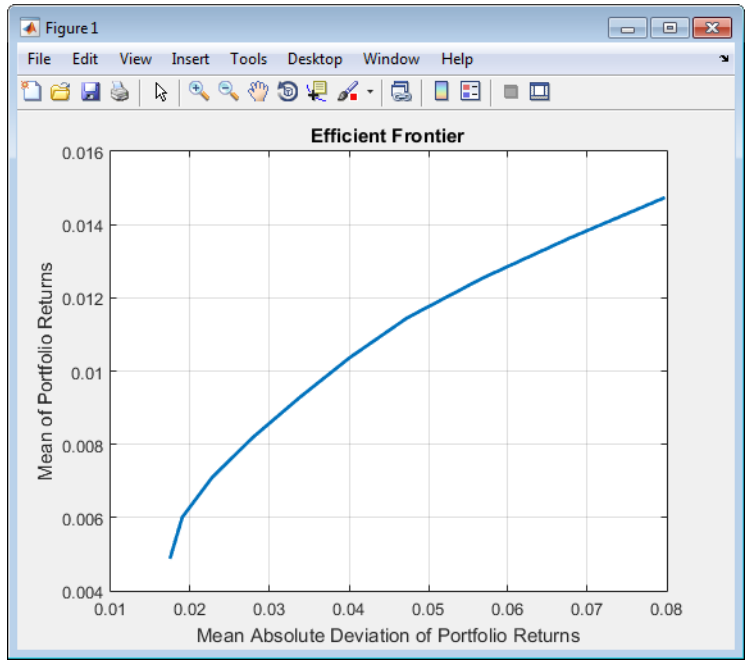
You can use the `PortfolioMAD` function to directly set up a "standard" portfolio optimization problem. Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('Scenarios', AssetScenarios, ...
'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1);
```

The `LowerBound` property value undergoes scalar expansion since `AssetScenarios` provides the dimensions of the problem.

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

```
p.plotFrontier;
```

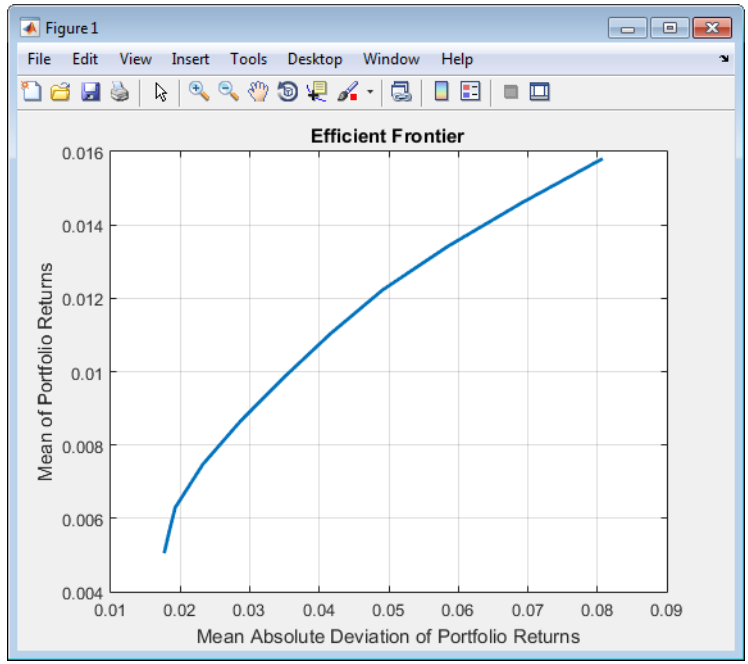


Using the PortfolioMAD Function with a Sequence of Steps

An alternative way to accomplish the same task of setting up a "standard" MAD portfolio optimization problem, given `AssetScenarios` variable is:

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = PortfolioMAD(p, 'LowerBound', 0);
```

```
p = PortfolioMAD(p, 'LowerBudget', 1, 'UpperBudget', 1);

plotFrontier(p);
```



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

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
```

```
p = PortfolioMAD(p, 'LowerBound', zeros(size(m)));
p = PortfolioMAD(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setScenarios(p, AssetScenarios);
```

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

Shortcuts for Property Names

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

Shortcuts for Property Names

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

For example, this call to the `PortfolioMAD` function uses these shortcuts for properties:

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('scenario', AssetScenarios, 'lb', 0, 'budget', 1);
plotFrontier(p);
```

Direct Setting of Portfolio Object Properties

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

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

```
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;

p = setScenarios(p, AssetScenarios);

p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));

plotFrontier(p);
```

Note Scenarios cannot be assigned directly to a PortfolioMAD object. Scenarios must always be set through either the `PortfolioMAD` function, the `setScenarios` function, or any of the scenario simulation functions.

See Also

`PortfolioMAD` | `estimateBounds`

Related Examples

- “Common Operations on the PortfolioMAD Object” on page 6-34
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Common Operations on the PortfolioMAD Object

In this section...

Naming a PortfolioMAD Object

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

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

Configuring the Assets in the Asset Universe

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

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

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

Setting Up a List of Asset Identifiers

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

Setting Up Asset Lists Using the PortfolioMAD Function

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

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

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

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

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

```
5
```

Setting Up Asset Lists Using the setAssetList Function

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

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

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

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

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

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

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

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

```
p = PortfolioMAD('numassets',5);
p = setAssetList(p);
disp(p.AssetList);
```

```
'Asset1'    'Asset2'    'Asset3'    'Asset4'    'Asset5'
```

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

```
p = PortfolioMAD('numassets',5);
p.defaultforAssetList = 'ETF';
p = setAssetList(p);
disp(p.AssetList);
```

```
'ETF1'    'ETF2'    'ETF3'    'ETF4'    'ETF5'
```

Truncating and Padding Asset Lists

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

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

```
Warning: Input list of assets has 2 too few identifiers. Padding with numbered
assets.
> In PortfolioMAD.setAssetList at 121
  Columns 1 through 4

    '921937835'    '922908769'    '922042775'    'UnknownCUSIP4'

  Column 5

    'UnknownCUSIP5'
```

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

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

```
Warning: AssetList has 1 too many identifiers. Using first 4 assets.
> In PortfolioMAD.checkarguments at 410
  In PortfolioMAD.PortfolioMAD>PortfolioMAD.PortfolioMAD at 187
    'AGG'    'EEM'    'MDY'    'SPY'
```

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

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

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

See Also

`PortfolioMAD` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort`

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 6-39
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Setting Up an Initial or Current Portfolio

In many applications, creating a new optimal portfolio requires comparing the new portfolio with an initial or current portfolio to form lists of purchases and sales. The PortfolioMAD object property `InitPort` lets you identify an initial or current portfolio. The initial portfolio also plays an essential role if you have either transaction costs or turnover constraints. The initial portfolio need not be feasible within the constraints of the problem. This can happen if the weights in a portfolio have shifted such that some constraints become violated. To check if your initial portfolio is feasible, use the`checkFeasibility` function described in "Validating MAD Portfolios" on page 6-93. Suppose that you have an initial portfolio in `x0`, then use the `PortfolioMAD` function to set up an initial portfolio:

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

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

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

To clear an initial portfolio from your PortfolioMAD object, use either the `PortfolioMAD` function or the `setInitPort` function with an empty input for the `InitPort` property. If transaction costs or turnover constraints are set, it is not possible to clear the

InitPort property in this way. In this case, to clear InitPort, first clear the dependent properties and then clear theInitPort property.

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

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

```
 0.3000
 0.2000
 0.2000
      0
```

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

```
p = PortfolioMAD;
p = setInitPort(p, 1/4, 4);
disp(p.InitPort);
```

```
 0.2500
 0.2500
 0.2500
 0.2500
```

PortfolioMAD object functions that work with either transaction costs or turnover constraints also depend on the InitPort property. So, the set functions for transaction costs or turnover constraints permit the assignment of a value for the InitPort property as part of their implementation. For details, see “Working with Average Turnover Constraints Using PortfolioMAD Object” on page 6-85, “Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-88, and “Working with Transaction Costs” on page 6-56. If either transaction costs or turnover constraints are used, then the InitPort property must have a nonempty value. Absent a specific value assigned through the PortfolioMAD function or various set functions, the PortfolioMAD object sets InitPort to 0 and warns if BuyCost, SellCost, or Turnover properties are set. This example shows what happens if you specify an average turnover constraint with an initial portfolio:

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

```
    0.3000
    0.2000
    0.2000
         0
```

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

```
p = PortfolioMAD('Turnover', 0.3);
disp(p.InitPort);
```

```
Warning: InitPort and NumAssets are empty and either transaction costs or
turnover constraints specified. Will set NumAssets = 1 and InitPort = 0.
> In PortfolioMAD.checkarguments at 446
  In PortfolioMAD.PortfolioMAD>PortfolioMAD.PortfolioMAD at 190
     0
```

See Also

`PortfolioMAD` | `checkFeasibility` | `estimateBounds` | `setAssetList` | `setInitPort`

Related Examples

- “Common Operations on the PortfolioMAD Object” on page 6-34
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Asset Returns and Scenarios Using PortfolioMAD Object

In this section...

How Stochastic Optimization Works

The MAD of a portfolio is mean-absolute deviation. For the definition of the MAD function, see "Risk Proxy" on page 6-6. Although analytic solutions for MAD exist for a few probability distributions, an alternative is to compute the expectation for MAD with samples from the probability distribution of asset returns. These samples are called scenarios and, given a collection of scenarios, the portfolio optimization problem becomes a stochastic optimization problem.

As a function of the portfolio weights, the MAD of the portfolio is a convex non-smooth function (see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7). The PortfolioMAD object computes MAD as this nonlinear function which can be handled by the solver `fmincon` Optimization Toolbox. The nonlinear programming solver `fmincon` has several algorithms that can be selected with the `setSolver` function, the two algorithms that work best in practice are `'sqp'` and `'active-set'`.

There are reformulations of the MAD portfolio optimization problem (see Konno and Yamazaki [50] at "Portfolio Optimization" on page A-7) that result in a linear programming problem, which can be solved either with standard linear programming techniques or with stochastic programming solvers. The PortfolioMAD object, however, does not reformulate the problem in such a manner. The PortfolioMAD object computes the MAD as a nonlinear function. The convexity of the MAD, as a function of the portfolio

weights and the dull edges when the number of scenarios is large, make the MAD portfolio optimization problem tractable, in practice, for certain nonlinear programming solvers, such as `fmincon` from Optimization Toolbox. To learn more about the workflow when using PortfolioMAD objects, see "PortfolioMAD Object Workflow" on page 6-19.

What Are Scenarios?

Since mean absolute deviation portfolio optimization works with scenarios of asset returns to perform the optimization, several ways exist to specify and simulate scenarios. In many applications with MAD portfolio optimization, asset returns may have distinctly nonnormal probability distributions with either multiple modes, binning of returns, truncation of distributions, and so forth. In other applications, asset returns are modeled as the result of various simulation methods that might include Monte-Carlo simulation, quasi-random simulation, and so forth. Often, the underlying probability distribution for risk factors may be multivariate normal but the resultant transformations are sufficiently nonlinear to result in distinctively nonnormal asset returns.

For example, this occurs with bonds and derivatives. In the case of bonds with a nonzero probability of default, such scenarios would likely include asset returns that are −100% to indicate default and some values slightly greater than −100% to indicate recovery rates.

Although the PortfolioMAD object has functions to simulate multivariate normal scenarios from either data or moments (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`), the usual approach is to specify scenarios directly from your own simulation functions. These scenarios are entered directly as a matrix with a sample for all assets across each row of the matrix and with samples for an asset down each column of the matrix. The architecture of the MAD portfolio optimization tools references the scenarios through a function handle so scenarios that have been set cannot be accessed directly as a property of the PortfolioMAD object.

Setting Scenarios Using the PortfolioMAD Function

Suppose that you have a matrix of scenarios in the `AssetScenarios` variable. The scenarios are set through the `PortfolioMAD` function with:

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

```
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('Scenarios', AssetScenarios);

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

```
4

20000
```

Notice that the PortfolioMAD object determines and fixes the number of assets in `NumAssets` and the number of scenarios in `NumScenarios` based on the scenario's matrix. You can change the number of scenarios by calling the `PortfolioMAD` function with a different scenario matrix. However, once the `NumAssets` property has been set in the object, you cannot enter a scenario matrix with a different number of assets. The `getScenarios` function lets you recover scenarios from a PortfolioMAD object. You can also obtain the mean and covariance of your scenarios using `estimateScenarioMoments`.

Although not recommended for the casual user, an alternative way exists to recover scenarios by working with the function handle that points to scenarios in the PortfolioMAD object. To access some or all the scenarios from a PortfolioMAD object, the hidden property `localScenarioHandle` is a function handle that points to a function to obtain scenarios that have already been set. To get scenarios directly from a PortfolioMAD object `p`, use

```
scenarios = p.localScenarioHandle([], []);
```

and to obtain a subset of scenarios from rows `startrow` to `endrow`, use

```
scenarios = p.localScenarioHandle(startrow, endrow);
```

where $1 \le$ `startrow` $\le$ `endrow` $\le$ `numScenarios`.

Setting Scenarios Using the setScenarios Function

You can also set scenarios using `setScenarios`. For example, given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set:

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);

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

```
4

20000
```

Estimating the Mean and Covariance of Scenarios

The `estimateScenarioMoments` function obtains estimates for the mean and covariance of scenarios in a PortfolioMAD object.

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

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```
p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
[mean, covar] = estimateScenarioMoments(p)

mean =

    0.0044
    0.0084
    0.0108
    0.0155


covar =

    0.0005    0.0003    0.0002   -0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0047    0.0028
   -0.0000    0.0010    0.0028    0.0103
```

Simulating Normal Scenarios

As a convenience, the two functions (`simulateNormalScenariosByData` and `simulateNormalScenariosByMoments`) exist to simulate scenarios from data or moments under an assumption that they are distributed as multivariate normal random asset returns.

Simulating Normal Scenarios from Returns or Prices

Given either return or price data, use the `simulateNormalScenariosByData` function to simulate multivariate normal scenarios. Either returns or prices are stored as matrices with samples going down the rows and assets going across the columns. In addition, returns or prices can be stored in a financial time series `fints` object (see "Simulating Normal Scenarios from Time Series Data" on page 6-50). To illustrate using `simulateNormalScenariosByData`, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with `portsim`. The default behavior of `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by `portsim` in the variable *X*, a price series is created in the variable *Y*:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
```

```
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);
Y = ret2tick(X);
```

Note Portfolio optimization requires that you use total returns and not just price returns. So, "returns" should be total returns and "prices" should be total return prices.

Given asset returns and prices in variables *X* and *Y* from above, this sequence of examples demonstrates equivalent ways to simulate multivariate normal scenarios for the PortfolioMAD object. Assume a PortfolioMAD object created in `p` that uses the asset returns in *X* uses `simulateNormalScenariosByData`:

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, X, 20000);

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0033
    0.0085
    0.0095
    0.1503


passetcovar =

    0.0055    0.0004    0.0002    0.0001
    0.0004    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0049    0.0028
    0.0001    0.0010    0.0028    0.0102
```

The moments that you obtain from this simulation will likely differ from the moments listed here because the scenarios are random samples from the estimated multivariate normal probability distribution of the input returns *X*.

The default behavior of `simulateNormalScenariosByData` is to work with asset returns. If, instead, you have asset prices as in the variable *Y*, `simulateNormalScenariosByData` accepts a name-value pair argument name

'DataFormat' with a corresponding value set to 'prices' to indicate that the input to the function is in the form of asset prices and not returns (the default value for the 'DataFormat' argument is 'returns'). This example simulates scenarios with the asset price data in *Y* for the PortfolioMAD object p:

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =

    0.0043
    0.0083
    0.0099
    0.1500


passetcovar =

    0.0053    0.0003    0.0001    0.0002
    0.0003    0.0024    0.0017    0.0010
    0.0001    0.0017    0.0047    0.0027
    0.0002    0.0010    0.0027    0.0100
```

Simulating Normal Scenarios with Missing Data

Often when working with multiple assets, you have missing data indicated by NaN values in your return or price data. Although "Multivariate Normal Regression" on page 9-2 goes into detail about regression with missing data, the simulateNormalScenariosByData function has a name-value pair argument name 'MissingData' that indicates with a Boolean value whether to use the missing data capabilities of Financial Toolbox. The default value for 'MissingData' is false which removes all samples with NaN values. If, however, 'MissingData' is set to true, simulateNormalScenariosByData uses the ECM algorithm to estimate asset moments. This example shows how this works on price data with missing values:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];
```

```
X = portsim(m', C, 120);
Y = ret2tick(X);
Y(1:20,1) = NaN;
Y(1:12,4) = NaN;
```

Notice that the prices above in `Y` have missing values in the first and fourth series.

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

q = PortfolioMAD;
q = simulateNormalScenariosByData(q, Y, 20000, 'dataformat', 'prices', 'missingdata', true);

[passetmean, passetcovar] = estimateScenarioMoments(p)
[qassetmean, qassetcovar] = estimateScenarioMoments(q)
```

```
passetmean =

    0.0095
    0.0103
    0.0124
    0.1505


passetcovar =

    0.0054    0.0000   -0.0005   -0.0006
    0.0000    0.0021    0.0015    0.0010
   -0.0005    0.0015    0.0046    0.0026
   -0.0006    0.0010    0.0026    0.0100


qassetmean =

    0.0092
    0.0082
    0.0094
    0.1463


qassetcovar =

    0.0071   -0.0000   -0.0006   -0.0006
   -0.0000    0.0032    0.0023    0.0015
   -0.0006    0.0023    0.0064    0.0036
   -0.0006    0.0015    0.0036    0.0133
```

The first PortfolioMAD object, `p`, contains scenarios obtained from price data in `Y` where `NaN` values are discarded and the second PortfolioMAD object, `q`, contains scenarios obtained from price data in `Y` that accommodate missing values. Each time you run this example, you get different estimates for the moments in `p` and `q`.

Simulating Normal Scenarios from Time Series Data

The `simulateNormalScenariosByData` function also accepts asset returns or prices stored in financial time series (`fints`) objects. The function implicitly works with matrices of data or data in a `fints` object using the same rules for whether the data are returns or prices. To illustrate, use `fints` to create the `fints` object `Xfts` that contains asset returns generated with `fints` (see "Simulating Normal Scenarios from Returns or Prices" on page 6-46) and add series labels:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

X = portsim(m', C, 120);

d = (datenum('31-jan-2001'):datenum('31-dec-2010'))';
Xfts = fints(d, zeros(numel(d),4), {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'});
Xfts = tomonthly(Xfts);

Xfts.Bonds = X(:,1);
Xfts.LargeCap = X(:,2);
Xfts.SmallCap = X(:,3);
Xfts.Emerging = X(:,4);

p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Xfts, 20000);

[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =

    0.0038
    0.0078
    0.0102
    0.1492


passetcovar =

    0.0053    0.0004    0.0001   -0.0000
    0.0004    0.0024    0.0017    0.0010
```

```
    0.0001    0.0017    0.0048    0.0028
   -0.0000    0.0010    0.0028    0.0103
```

The name-value inputs `'DataFormat'` to handle return or price data and`'MissingData'` to ignore or use samples with missing values also work for `fints` data. In addition, `simulateNormalScenariosByData` extracts asset names or identifiers from a `fints` object if the argument name `'GetAssetList'` is set to `true` (the default value is `false`). If the `'GetAssetList'` value is `true`, the identifiers are used to set the `AssetList` property of the PortfolioMAD object. Thus, repeating the formation of the PortfolioMAD object `q` from the previous example with the `'GetAssetList'` flag set to `true` extracts the series labels from the `fints` object:

```
p = simulateNormalScenariosByData(p, Xfts, 20000, 'getassetlist', true);
disp(p.AssetList)
```

```
 'Bonds'    'LargeCap'    'SmallCap'    'Emerging'
```

If you set the`'GetAssetList'` flag set to `true` and your input data is in a matrix, `simulateNormalScenariosByData` uses the default labeling scheme from `setAssetList` as described in “Setting Up a List of Asset Identifiers” on page 6-35.

Simulating Normal Scenarios for Mean and Covariance

Given the mean and covariance of asset returns, use the `simulateNormalScenariosByMoments` function to simulate multivariate normal scenarios. The mean can be either a row or column vector and the covariance matrix must be a symmetric positive-semidefinite matrix. Various rules for scalar expansion apply. To illustrate using `simulateNormalScenariosByMoments`, start with moments in `m` and `C` and generate 20,000 scenarios:

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

p = PortfolioMAD;
p = simulateNormalScenariosByMoments(p, m, C, 20000);
[passetmean, passetcovar] = estimateScenarioMoments(p)

passetmean =
```

```
    0.0040
    0.0084
    0.0105
    0.1513


passetcovar =

    0.0053    0.0003    0.0002    0.0001
    0.0003    0.0024    0.0017    0.0009
    0.0002    0.0017    0.0048    0.0028
    0.0001    0.0009    0.0028    0.0102
```

`simulateNormalScenariosByMoments` performs scalar expansion on arguments for the moments of asset returns. If `NumAssets` has not already been set, a scalar argument is interpreted as a scalar with `NumAssets` set to `1`.

`simulateNormalScenariosByMoments` provides an additional optional argument to specify the number of assets so that scalar expansion works with the correct number of assets. In addition, if either a scalar or vector is input for the covariance of asset returns, a diagonal matrix is formed such that a scalar expands along the diagonal and a vector becomes the diagonal.

See Also

`PortfolioMAD` | `setCosts` | `setScenarios` | `simulateNormalScenariosByData` | `simulateNormalScenariosByMoments`

Related Examples

- "Working with a Riskless Asset" on page 6-54
- "Working with Transaction Costs" on page 6-56
- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with a Riskless Asset

The PortfolioMAD object has a separate `RiskFreeRate` property that stores the rate of return of a riskless asset. Thus, you can separate your universe into a riskless asset and a collection of risky assets. For example, assume that your riskless asset has a return in the scalar variable `r0`, then the property for the `RiskFreeRate` is set using the `PortfolioMAD` function:

```
r0 = 0.01/12;

p = PortfolioMAD;
p = PortfolioMAD('RiskFreeRate', r0);
disp(p.RiskFreeRate);

 8.3333e-04
```

Note If your portfolio problem has a budget constraint such that your portfolio weights must sum to `1`, then the riskless asset is irrelevant.

See Also

`PortfolioMAD` | `setCosts` | `setScenarios` | `simulateNormalScenariosByData` | `simulateNormalScenariosByMoments`

Related Examples

- “Working with Transaction Costs” on page 6-56
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Validate the MAD Portfolio Problem” on page 6-91
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with Transaction Costs

The difference between net and gross portfolio returns is transaction costs. The net portfolio return proxy has distinct proportional costs to purchase and to sell assets which are maintained in the PortfolioMAD object properties `BuyCost` and `SellCost`. Transaction costs are in units of total return and, as such, are proportional to the price of an asset so that they enter the model for net portfolio returns in return form. For example, suppose that you have a stock currently priced $40 and your usual transaction costs are 5 cents per share. Then the transaction cost for the stock is 0.05/40 = 0.00125 (as defined in "Net Portfolio Returns" on page 6-5). Costs are entered as positive values and credits are entered as negative values.

Setting Transaction Costs Using the PortfolioMAD Function

To set up transaction costs, you must specify an initial or current portfolio in the `InitPort` property. If the initial portfolio is not set when you set up the transaction cost properties, `InitPort` is `0`. The properties for transaction costs can be set using the `PortfolioMAD` function. For example, assume that purchase and sale transaction costs are in the variables `bc` and `sc` and an initial portfolio is in the variable `x0`, then transaction costs are set:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioMAD('BuyCost', bc, 'SellCost', sc, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
     5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
```

```
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Setting Transaction Costs Using the setCosts Function

You can also set the properties for transaction costs using `setCosts`. Assume that you have the same costs and initial portfolio as in the previous example. Given a PortfolioMAD object `p` with an initial portfolio already set, use `setCosts` to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioMAD('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
    5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

    0.4000
    0.2000
```

```
    0.2000
    0.1000
    0.1000
```

You can also set up the initial portfolio's `InitPort` value as an optional argument to `setCosts` so that the following is an equivalent way to set up transaction costs:

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioMAD;
p = setCosts(p, bc, sc, x0);

disp(p.NumAssets);
disp(p.BuyCost);
disp(p.SellCost);
disp(p.InitPort);
```

```
    5

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Setting Transaction Costs with Scalar Expansion

Both the `PortfolioMAD` function and `setCosts` function implement scalar expansion on the arguments for transaction costs and the initial portfolio. If the `NumAssets`

property is already set in the PortfolioMAD object, scalar arguments for these properties are expanded to have the same value across all dimensions. In addition, `setCosts` lets you specify `NumAssets` as an optional final argument. For example, assume that you have an initial portfolio `x0` and you want to set common transaction costs on all assets in your universe. You can set these costs in any of these equivalent ways:

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioMAD('InitPort', x0, 'BuyCost', 0.002, 'SellCost', 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setCosts(p, 0.002, 0.002);
```

or

```
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];
p = PortfolioMAD;
p = setCosts(p, 0.002, 0.002, x0);
```

To clear costs from your PortfolioMAD object, use either the`PortfolioMAD` function or `setCosts` with empty inputs for the properties to be cleared. For example, you can clear sales costs from the PortfolioMAD object `p` in the previous example:

```
p = PortfolioMAD(p, 'SellCost', []);
```

See Also

`PortfolioMAD` | `setCosts` | `setScenarios` | `simulateNormalScenariosByData` | `simulateNormalScenariosByMoments`

Related Examples

- “Working with a Riskless Asset” on page 6-54
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Validate the MAD Portfolio Problem” on page 6-91
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96

- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Working with MAD Portfolio Constraints Using Defaults

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

When setting up your portfolio set, ensure that the portfolio set satisfies these conditions. The most basic or "default" portfolio set requires portfolio weights to be nonnegative (using the lower-bound constraint) and to sum to `1` (using the budget constraint). For information on the workflow when using PortfolioMAD objects, see "PortfolioMAD Object Workflow" on page 6-19.

Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object

The "default" MAD portfolio problem has two constraints on portfolio weights:

- Portfolio weights must be nonnegative.
- Portfolio weights must sum to `1`.

Implicitly, these constraints imply that portfolio weights are no greater than `1`, although this is a superfluous constraint to impose on the problem.

Setting Default Constraints Using the PortfolioMAD Function

Given a portfolio optimization problem with `NumAssets` = `20` assets, use the `PortfolioMAD` function to set up a default problem and explicitly set bounds and budget constraints:

```
p = PortfolioMAD('NumAssets', 20, 'LowerBound', 0, 'Budget', 1);
disp(p);
```

```
 PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
```

```
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
            Name: []
       NumAssets: 20
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [20x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Setting Default Constraints Using the setDefaultConstraints Function

An alternative approach is to use the `setDefaultConstraints` function. If the number of assets is already known in a PortfolioMAD object, use `setDefaultConstraints` with no arguments to set up the necessary bound and budget constraints. Suppose that you have 20 assets to set up the portfolio set for a default problem:

```
p = PortfolioMAD('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);

PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
            Name: []
```

```
       NumAssets: 20
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [20x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

If the number of assets is unknown, `setDefaultConstraints` accepts `NumAssets` as an optional argument to form a portfolio set for a default problem. Suppose that you have 20 assets:

```
p = PortfolioMAD;
p = setDefaultConstraints(p, 20);
disp(p);
```

```
PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
            Name: []
       NumAssets: 20
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [20x1 double]
```

```
 UpperBound: []
LowerBudget: 1
UpperBudget: 1
GroupMatrix: []
 LowerGroup: []
 UpperGroup: []
     GroupA: []
     GroupB: []
 LowerRatio: []
 UpperRatio: []
```

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Working with Bound Constraints Using PortfolioMAD Object" on page 6-66
- "Working with Budget Constraints Using PortfolioMAD Object" on page 6-69
- "Working with Group Constraints Using PortfolioMAD Object" on page 6-71
- "Working with Group Ratio Constraints Using PortfolioMAD Object" on page 6-75
- "Working with Linear Equality Constraints Using PortfolioMAD Object" on page 6-79
- "Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-82
- "Working with Average Turnover Constraints Using PortfolioMAD Object" on page 6-85
- "Working with One-Way Turnover Constraints Using PortfolioMAD Object" on page 6-88
- "Creating the PortfolioMAD Object" on page 6-26
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111

- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Working with Bound Constraints Using PortfolioMAD Object

Bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “Bound Constraints” on page 6-11). Although every MAD portfolio set must be bounded, it is not necessary to specify a MAD portfolio set with explicit bound constraints. For example, you can create a MAD portfolio set with an implicit upper bound constraint or a MAD portfolio set with average turnover constraints. The bound constraints have properties `LowerBound` for the lower-bound constraint and `UpperBound` for the upper-bound constraint. Set default values for these constraints using the `setDefaultConstraints` function (see “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-61).

Setting Bounds Using the PortfolioMAD Function

The properties for bound constraints are set through the `PortfolioMAD` function. Suppose that you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. The bound constraints for a balanced fund are set with:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioMAD('LowerBound', lb, 'UpperBound', ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
    2

    0.5000
    0.2500

    0.7500
    0.5000
```

To continue with this example, you must set up a budget constraint. For details, see “Budget Constraints” on page 6-12.

Setting Bounds Using the setBounds Function

You can also set the properties for bound constraints using `setBounds`. Suppose that you have a balanced fund with stocks that can range from 50% to 75% of your portfolio

and bonds that can range from 25% to 50% of your portfolio. Given a PortfolioMAD object `p`, use `setBounds` to set the bound constraints:

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];
p = PortfolioMAD;
p = setBounds(p, lb, ub);
disp(p.NumAssets);
disp(p.LowerBound);
disp(p.UpperBound);
```

```
    2

    0.5000
    0.2500

    0.7500
    0.5000
```

Setting Bounds Using the PortfolioMAD Function or setBounds Function

Both the `PortfolioMAD` function and `setBounds` function implement scalar expansion on either the `LowerBound` or `UpperBound` properties. If the `NumAssets` property is already set in the PortfolioMAD object, scalar arguments for either property expand to have the same value across all dimensions. In addition, `setBounds` lets you specify `NumAssets` as an optional argument. Suppose that you have a universe of 500 assets and you want to set common bound constraints on all assets in your universe. Specifically, you are a long-only investor and want to hold no more than 5% of your portfolio in any single asset. You can set these bound constraints in any of these equivalent ways:

```
p = PortfolioMAD('NumAssets', 500, 'LowerBound', 0, 'UpperBound', 0.05);
```

or

```
p = PortfolioMAD('NumAssets', 500);
p = setBounds(p, 0, 0.05);
```

or

```
p = PortfolioMAD;
p = setBounds(p, 0, 0.05, 500);
```

To clear bound constraints from your PortfolioMAD object, use either the `PortfolioMAD` function or `setBounds` with empty inputs for the properties to be cleared. For example, to clear the upper-bound constraint from the PortfolioMAD object `p` in the previous example:

```
p = PortfolioMAD(p, 'UpperBound', []);
```

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-61
- “Creating the PortfolioMAD Object” on page 6-26
- “Validate the MAD Portfolio Problem” on page 6-91
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Working with Budget Constraints Using PortfolioMAD Object

The budget constraint is an optional linear constraint that maintains upper and lower bounds on the sum of portfolio weights (see "Budget Constraints" on page 5-12). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a MAD portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to `1` using `setDefaultConstraints` (see "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61).

Setting Budget Constraints Using the PortfolioMAD Function

The properties for the budget constraint can also be set using the `PortfolioMAD` function. Suppose that you have an asset universe with many risky assets and a riskless asset and you want to ensure that your portfolio never holds more than 1% cash, that is, you want to ensure that you are 99–100% invested in risky assets. The budget constraint for this portfolio can be set with:

```
p = PortfolioMAD('LowerBudget', 0.99, 'UpperBudget', 1);
disp(p.LowerBudget);
disp(p.UpperBudget);
```

```
 0.9900

1
```

Setting Budget Constraints Using the setBudget Function

You can also set the properties for a budget constraint using `setBudget`. Suppose that you have a fund that permits up to 10% leverage which means that your portfolio can be from 100% to 110% invested in risky assets. Given a PortfolioMAD object `p`, use `setBudget` to set the budget constraints:

```
p = PortfolioMAD;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);
disp(p.UpperBudget);
```

```
 1

 1.1000
```

If you were to continue with this example, then set the `RiskFreeRate` property to the borrowing rate to finance possible leveraged positions. For details on the `RiskFreeRate` property, see “Working with a Riskless Asset” on page 6-54. To clear either bound for the budget constraint from your PortfolioMAD object, use either the `PortfolioMAD` function or `setBudget` with empty inputs for the properties to be cleared. For example, clear the upper-budget constraint from the PortfolioMAD object `p` in the previous example with:

```
p = PortfolioMAD(p, 'UpperBudget', []);
```

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-61
- “Creating the PortfolioMAD Object” on page 6-26
- “Validate the MAD Portfolio Problem” on page 6-91
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Working with Group Constraints Using PortfolioMAD Object

Group constraints are optional linear constraints that group assets together and enforce bounds on the group weights (see “Group Constraints” on page 6-13). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the PortfolioMAD Function

The properties for group constraints are set through the `PortfolioMAD` function. Suppose that you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio, then you can set group constraints:

```
G = [ 1 1 1 0 0 ];
p = PortfolioMAD('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

The group matrix `G` can also be a logical matrix so that the following code achieves the same result.

```
G = [ true true true false false ];
p = PortfolioMAD('GroupMatrix', G, 'UpperGroup', 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

Setting Group Constraints Using the setGroups and addGroups Functions

You can also set the properties for group constraints using `setGroups`. Suppose that you have a portfolio of five assets and want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a PortfolioMAD object `p`, use `setGroups` to set the group constraints:

```
G = [ true true true false false ];
p = PortfolioMAD;
p = setGroups(p, G, [], 0.3);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.UpperGroup);
```

```
5

1     1     1     0     0

0.3000
```

In this example, you would set the `LowerGroup` property to be empty (`[]`).

Suppose that you want to add another group constraint to make odd-numbered assets constitute at least 20% of your portfolio. Set up an augmented group matrix and introduce infinite bounds for unconstrained group bounds or use the `addGroups` function to build up group constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioMAD;
G = [ true true true false false ];    % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ];    % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
 5

1     1     1     0     0
1     0     1     0     1

  -Inf
0.2000
```

```
0.3000
  Inf
```

`addGroups` determines which bounds are unbounded so you only need to focus on the constraints that you want to set.

The `PortfolioMAD` function, `setGroups`, and `addGroups` implement scalar expansion on either the `LowerGroup` or `UpperGroup` properties based on the dimension of the group matrix in the property `GroupMatrix`. Suppose that you have a universe of 30 assets with 6 asset classes such that assets 1–5, assets 6–12, assets 13–18, assets 19–22, assets 23–27, and assets 28–30 constitute each of your asset classes and you want each asset class to fall from 0% to 25% of your portfolio. Let the following group matrix define your groups and scalar expansion define the common bounds on each group:

```
p = PortfolioMAD;
G = blkdiag(true(1,5), true(1,7), true(1,6), true(1,4), true(1,5), true(1,3));
p = setGroups(p, G, 0, 0.25);
disp(p.NumAssets);
disp(p.GroupMatrix);
disp(p.LowerGroup);
disp(p.UpperGroup);
```

```
30

  Columns 1 through 13

     1     1     1     1     1     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     1     1     1     1     0
     0     0     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0

  Columns 14 through 26

     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0
     1     1     1     1     1     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     1     0     0     0     0
     0     0     0     0     0     0     0     0     0     1     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     0

  Columns 27 through 30

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     1     0     0     0
     0     1     1     1

     0
     0
     0
     0
```

```
     0
     0

0.2500
0.2500
0.2500
0.2500
0.2500
0.2500
```

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
- "Creating the PortfolioMAD Object" on page 6-26
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with Group Ratio Constraints Using PortfolioMAD Object

Group ratio constraints are optional linear constraints that maintain bounds on proportional relationships among groups of assets (see “Group Ratio Constraints” on page 6-14). Although the constraints are implemented as general constraints, the usual convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group compared to a comparison group fall within specified bounds. Group ratio constraints have properties:

- `GroupA` for the base membership matrix
- `GroupB` for the comparison membership matrix
- `LowerRatio` for the lower-bound constraint on the ratio of groups
- `UpperRatio` for the upper-bound constraint on the ratio of groups

Setting Group Ratio Constraints Using the PortfolioMAD Function

The properties for group ratio constraints are set using `PortfolioMAD` function. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose that you have six assets with three financial companies (assets 1–3) and three nonfinanical companies (assets 4–6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ];    % financial companies
GB = [ 0 0 0 1 1 1 ];    % nonfinancial companies
p = PortfolioMAD('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioMAD('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Setting Group Ratio Constraints Using the setGroupRatio and addGroupRatio Functions

You can also set the properties for group ratio constraints using `setGroupRatio`. For example, assume that you want the ratio of financial to nonfinancial companies in your portfolios to never go above 50%. Suppose that you have six assets with three financial companies (assets 1–3) and three nonfinanical companies (assets 4–6). Given a PortfolioMAD object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose that you want to add another group ratio constraint to ensure that the weights in odd-numbered assets constitute at least 20% of the weights in nonfinancial assets your portfolio. You can set up augmented group ratio matrices and introduce infinite bounds for unconstrained group ratio bounds, or you can use the `addGroupRatio` function to build up group ratio constraints. For this example, create another group matrix for the second group constraint:

```
p = PortfolioMAD;
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ];   % odd-numbered companies
GB = [ false false false true true true ];   % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);
disp(p.GroupA);
disp(p.GroupB);
disp(p.LowerRatio);
disp(p.UpperRatio);
```

```
6

1     1     1     0     0     0
1     0     1     0     1     0

0     0     0     1     1     1
0     0     0     1     1     1

  -Inf
0.2000

0.5000
  Inf
```

Notice that `addGroupRatio` determines which bounds are unbounded so you only need to focus on the constraints you want to set.

The`PortfolioMAD` function, `setGroupRatio`, and `addGroupRatio` implement scalar expansion on either the `LowerRatio` or `UpperRatio` properties based on the dimension of the group matrices in `GroupA` and `GroupB` properties.

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
- "Creating the PortfolioMAD Object" on page 6-26
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with Linear Equality Constraints Using PortfolioMAD Object

Linear equality constraints are optional linear constraints that impose systems of equalities on portfolio weights (see “Linear Equality Constraints” on page 6-11). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `bEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the PortfolioMAD Function

The properties for linear equality constraints are set using the `PortfolioMAD` function. Suppose that you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. To set this constraint:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD('AEquality', A, 'bEquality', b);
disp(p.NumAssets);
disp(p.AEquality);
disp(p.bEquality);
```

```
5

1     1     1     0     0

0.5000
```

Setting Linear Equality Constraints Using the setEquality and addEquality Functions

You can also set the properties for linear equality constraints using `setEquality`. Suppose that you have a portfolio of five assets and want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioMAD object `p`, use `setEquality` to set the linear equality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setEquality(p, A, b);
disp(p.NumAssets);
```

```
disp(p.AEquality);
disp(p.bEquality);
```

```
5

1     1     1     0     0

0.5000
```

Suppose that you want to add another linear equality constraint to ensure that the last three assets also constitute 50% of your portfolio. You can set up an augmented system of linear equalities or use `addEquality` to build up linear equality constraints. For this example, create another system of equalities:

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ];    % first equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % second equality constraint
b = 0.5;
p = addEquality(p, A, b);

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

```
5

1     1     1     0     0
0     0     1     1     1

0.5000
0.5000
```

The `PortfolioMAD` function, `setEquality`, and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
- "Creating the PortfolioMAD Object" on page 6-26
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with Linear Inequality Constraints Using PortfolioMAD Object

Linear inequality constraints are optional linear constraints that impose systems of inequalities on portfolio weights (see “Linear Inequality Constraints” on page 6-10). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the PortfolioMAD Function

The properties for linear inequality constraints are set using the `PortfolioMAD` function. Suppose that you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. To set up these constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD('AInequality', A, 'bInequality', b);
disp(p.NumAssets);
disp(p.AInequality);
disp(p.bInequality);
```

```
5

1     1     1     0     0

0.5000
```

Setting Linear Inequality Constraints Using the setInequality and addInequality Functions

You can also set the properties for linear inequality constraints using `setInequality`. Suppose that you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 50% of your portfolio. Given a PortfolioMAD object `p`, use `setInequality` to set the linear inequality constraints:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setInequality(p, A, b);
disp(p.NumAssets);
```

```
disp(p.AInequality);
disp(p.bInequality);
```

```
5

1     1     1     0     0

0.5000
```

Suppose that you want to add another linear inequality constraint to ensure that the last three assets constitute at least 50% of your portfolio. You can set up an augmented system of linear inequalities or use the `addInequality` function to build up linear inequality constraints. For this example, create another system of inequalities:

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

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

```
5

1     1     1     0     0
0     0    -1    -1    -1

0.5000
-0.5000
```

The `PortfolioMAD` function, `setInequality`, and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
- "Creating the PortfolioMAD Object" on page 6-26
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with Average Turnover Constraints Using PortfolioMAD Object

The turnover constraint is an optional linear absolute value constraint (see "Average Turnover Constraints" on page 6-15) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `PortfolioMAD` function or the `setTurnover` function. The turnover constraint depends on an initial or current portfolio, which is assumed to be zero if not set when the turnover constraint is set. The turnover constraint has properties `Turnover`, for the upper bound on average turnover, and `InitPort`, for the portfolio against which turnover is computed.

Setting Average Turnover Constraints Using the PortfolioMAD Function

The properties for the turnover constraints are set using the `PortfolioMAD` function. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and you want to ensure that average turnover is no more than 30%. To set this turnover constraint:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('Turnover', 0.3, 'InitPort', x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

Note if the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see "Setting Up an Initial or Current Portfolio" on page 6-39).

Setting Average Turnover Constraints Using the setTurnover Function

You can also set properties for portfolio turnover using `setTurnover` to specify both the upper bound for average turnover and an initial portfolio. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that average turnover is no more than 30%. Given a PortfolioMAD object `p`, use `setTurnover` to set the turnover constraint with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setTurnover(p, 0.3);

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

```
10

0.3000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD;
p = setTurnover(p, 0.3, x0);
disp(p.NumAssets);
disp(p.Turnover);
disp(p.InitPort);
```

```
10

0.3000

0.1200
0.0900
0.0800
```

```
0.0700
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

`setTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the PortfolioMAD object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setTurnover` lets you specify `NumAssets` as an optional argument. To clear turnover from your PortfolioMAD object, use the `PortfolioMAD` function or `setTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
- "Creating the PortfolioMAD Object" on page 6-26
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Working with One-Way Turnover Constraints Using PortfolioMAD Object

One-way turnover constraints are optional constraints (see “One-way Turnover Constraints” on page 6-15) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the `PortfolioMAD` function or the `setOneWayTurnover` function. One-way turnover constraints depend upon an initial or current portfolio, which is assumed to be zero if not set when the turnover constraints are set. One-way turnover constraints have properties `BuyTurnover`, for the upper bound on net purchases, `SellTurnover`, for the upper bound on net sales, and `InitPort`, for the portfolio against which turnover is computed.

Setting One-Way Turnover Constraints Using the PortfolioMAD Function

The Properties for the one-way turnover constraints are set using the `PortfolioMAD` function. Suppose that you have an initial portfolio with 10 assets in a variable `x0` and you want to ensure that turnover on purchases is no more than 30% and turnover on sales is no more than 20% of the initial portfolio. To set these turnover constraints:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('BuyTurnover', 0.3, 'SellTurnover', 0.2, 'InitPort', x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
10

0.3000

0.2000

0.1200
0.0900
0.0800
0.0700
0.1000
0.1000
0.1500
0.1100
```

```
0.0800
0.1000
```

If the `NumAssets` or `InitPort` properties are not set before or when the turnover constraint is set, various rules are applied to assign default values to these properties (see "Setting Up an Initial or Current Portfolio" on page 6-39).

Setting Turnover Constraints Using the setOneWayTurnover Function

You can also set properties for portfolio turnover using `setOneWayTurnover` to specify to the upper bounds for turnover on purchases (`BuyTurnover`) and sales (`SellTurnover`) and an initial portfolio. Suppose that you have an initial portfolio of 10 assets in a variable `x0` and want to ensure that turnover on purchases is no more than 30% and that turnover on sales is no more than 20% of the initial portfolio. Given a PortfolioMAD object `p`, use `setOneWayTurnover` to set the turnover constraints with and without the initial portfolio being set previously:

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);

disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

or

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD;
p = setOneWayTurnover(p, 0.3, 0.2, x0);
disp(p.NumAssets);
disp(p.BuyTurnover);
disp(p.SellTurnover);
disp(p.InitPort);
```

```
10

0.3000

0.2000

0.1200
0.0900
0.0800
0.0700
```

```
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

`setOneWayTurnover` implements scalar expansion on the argument for the initial portfolio. If the `NumAssets` property is already set in the PortfolioMAD object, a scalar argument for `InitPort` expands to have the same value across all dimensions. In addition, `setOneWayTurnover` lets you specify `NumAssets` as an optional argument. To remove one-way turnover from your PortfolioMAD object, use the `PortfolioMAD` function or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioMAD` | `setBounds` | `setBudget` | `setDefaultConstraints` | `setEquality` | `setGroupRatio` | `setGroups` | `setInequality` | `setOneWayTurnover` | `setTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-61
- “Creating the PortfolioMAD Object” on page 6-26
- “Validate the MAD Portfolio Problem” on page 6-91
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Validate the MAD Portfolio Problem

In this section...

Sometimes, you may want to validate either your inputs to, or outputs from, a portfolio optimization problem. Although most error checking that occurs during the problem setup phase catches most difficulties with a portfolio optimization problem, the processes to validate MAD portfolio sets and portfolios are time consuming and are best done offline. So, the portfolio optimization tools have specialized functions to validate MAD portfolio sets and portfolios. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-19.

Validating a MAD Portfolio Set

Since it is necessary and sufficient that your MAD portfolio set must be a nonempty, closed, and bounded set to have a valid portfolio optimization problem, the `estimateBounds` function lets you examine your portfolio set to determine if it is nonempty and, if nonempty, whether it is bounded. Suppose that you have the following MAD portfolio set which is an empty set because the initial portfolio at `0` is too far from a portfolio that satisfies the budget and turnover constraint:

```
p = PortfolioMAD('NumAssets', 3, 'Budget', 1);
p = setTurnover(p, 0.3, 0);
```

If a MAD portfolio set is empty, `estimateBounds` returns `NaN` bounds and sets the `isbounded` flag to `[]`:

```
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =

   NaN
   NaN
   NaN

ub =

   NaN
```

```
   NaN
   NaN

isbounded =

     []
```

Suppose that you create an unbounded MAD portfolio set as follows:

```
p = PortfolioMAD('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

  -Inf
  -Inf

ub =

  1.0e-008 *

   -0.3712
       Inf

isbounded =

     0
```

In this case, `estimateBounds` returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Finally, suppose that you created a MAD object that is both nonempty and bounded. `estimateBounds` not only validates the set, but also obtains tighter bounds which are useful if you are concerned with the actual range of portfolio choices for individual assets in your portfolio:

```
p = PortfolioMAD;
p = setBudget(p, 1,1);
p = setBounds(p, [ -0.1; 0.2; 0.3; 0.2 ], [ 0.5; 0.3; 0.9; 0.8 ]);

[lb, ub, isbounded] = estimateBounds(p)

lb =

   -0.1000
```

```
    0.2000
    0.3000
    0.2000

ub =

    0.3000
    0.3000
    0.7000
    0.6000

isbounded =

     1
```

In this example, all but the second asset has tighter upper bounds than the input upper bound implies.

Validating MAD Portfolios

Given a MAD portfolio set specified in a PortfolioMAD object, you often want to check if specific portfolios are feasible with respect to the portfolio set. This can occur with, for example, initial portfolios and with portfolios obtained from other procedures. The `checkFeasibility` function determines whether a collection of portfolios is feasible. Suppose that you perform the following portfolio optimization and want to determine if the resultant efficient portfolios are feasible relative to a modified problem.

First, set up a problem in the PortfolioMAD object `p`, estimate efficient portfolios in `pwgt`, and then confirm that these portfolios are feasible relative to the initial problem:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)

ans =

     1     1     1     1     1     1     1     1     1     1
```

Next, set up a different portfolio problem that starts with the initial problem with an additional a turnover constraint and an equally weighted initial portfolio:

```
q = setTurnover(p, 0.3, 0.25);
checkFeasibility(q, pwgt)

ans =

     0     0     1     1     1     0     0     0     0     0
```

In this case, only two of the 10 efficient portfolios from the initial problem are feasible relative to the new problem in PortfolioMAD object `q`. Solving the second problem using `checkFeasibility` demonstrates that the efficient portfolio for PortfolioMAD object `q` is feasible relative to the initial problem:

```
qwgt = estimateFrontier(q);
checkFeasibility(p, qwgt)

ans =

     1     1     1     1     1     1     1     1     1     1
```

See Also

`PortfolioMAD` | `checkFeasibility` | `estimateBounds`

Related Examples

- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96

- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object

There are two ways to look at a portfolio optimization problem that depends on what you are trying to do. One goal is to estimate efficient portfolios and the other is to estimate efficient frontiers. This section focuses on the former goal and “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111 focuses on the latter goal. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-19.

Obtaining Portfolios Along the Entire Efficient Frontier

The most basic way to obtain optimal portfolios is to obtain points over the entire range of the efficient frontier. Given a portfolio optimization problem in a PortfolioMAD object, the `estimateFrontier` function computes efficient portfolios spaced evenly according to the return proxy from the minimum to maximum return efficient portfolios. The number of portfolios estimated is controlled by the hidden property `defaultNumPorts` which is set to `10`. A different value for the number of portfolios estimated is specified as input to `estimateFrontier`. This example shows the default number of efficient portfolios over the entire range of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
disp(pwgt);
```

```
 Columns 1 through 8

    0.8907    0.7289    0.5614    0.3946    0.2257    0.0612         0    0.0000
    0.0330    0.1163    0.2119    0.3042    0.3998    0.4876    0.4400    0.3125
    0.0420    0.0469    0.0472    0.0505    0.0534    0.0580    0.0374    0.0018
    0.0343    0.1079    0.1794    0.2507    0.3211    0.3933    0.5226    0.6857

  Columns 9 through 10

    0.0000    0.0000
    0.1570    0.0000
    0.0000    0.0000
    0.8430    1.0000
```

If you want only four portfolios in the previous example:

```
pwgt = estimateFrontier(p, 4);

disp(pwgt);
```

```
  0.8907    0.3946         0    0.0000
  0.0330    0.3042    0.4401    0.0000
  0.0420    0.0505    0.0373    0.0000
  0.0343    0.2507    0.5227    1.0000
```

Starting from the initial portfolio, `estimateFrontier` also returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

  Columns 1 through 8

    0.8907    0.7289    0.5614    0.3946    0.2257    0.0612         0    0.0000
    0.0330    0.1163    0.2119    0.3042    0.3998    0.4876    0.4400    0.3125
    0.0420    0.0469    0.0472    0.0505    0.0534    0.0580    0.0374    0.0018
    0.0343    0.1079    0.1794    0.2507    0.3211    0.3933    0.5226    0.6857

  Columns 9 through 10

    0.0000    0.0000
    0.1570    0.0000
    0.0000    0.0000
    0.8430    1.0000


pbuy =

  Columns 1 through 8

    0.5907    0.4289    0.2614    0.0946         0         0         0         0
         0         0         0    0.0042    0.0998    0.1876    0.1400    0.0125
         0         0         0         0         0         0         0         0
         0    0.0079    0.0794    0.1507    0.2211    0.2933    0.4226    0.5857

  Columns 9 through 10

         0         0
         0         0
         0         0
    0.7430    0.9000
```

```
psell =

  Columns 1 through 8

         0         0         0         0    0.0743    0.2388    0.3000    0.3000
    0.2670    0.1837    0.0881         0         0         0         0         0
    0.1580    0.1531    0.1528    0.1495    0.1466    0.1420    0.1626    0.1982
    0.0657         0         0         0         0         0         0         0

  Columns 9 through 10

    0.3000    0.3000
    0.1430    0.3000
    0.2000    0.2000
         0         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Obtaining Endpoints of the Efficient Frontier” on page 6-100
- “Obtaining Efficient Portfolios for Target Returns” on page 6-103
- “Obtaining Efficient Portfolios for Target Risks” on page 6-106
- “Obtaining MAD Portfolio Risks and Returns” on page 6-111
- “Obtaining the PortfolioMAD Standard Deviation” on page 6-113
- “Plotting the Efficient Frontier for a PortfolioMAD Object” on page 6-115
- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21

- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Obtaining Endpoints of the Efficient Frontier

Often, you might be interested in the endpoint portfolios for the efficient frontier. Suppose that you want to determine the range of returns from minimum to maximum to refine a search for a portfolio with a specific target return. Use the `estimateFrontierLimits` function to obtain the endpoint portfolios:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);

disp(pwgt);

  0.8875    0.0000
  0.0373    0.0000
  0.0386    0.0000
  0.0366    1.0000
```

Note The endpoints of the efficient frontier depend upon the Scenarios in the PortfolioMAD object. If you change the Scenarios, you are likely to obtain different endpoints.

Starting from an initial portfolio, `estimateFrontierLimits` also returns purchases and sales to get from the initial portfolio to the endpoint portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales:

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

```
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierLimits(p);

display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.8927    0.0000
    0.0334    0.0000
    0.0422    0.0000
    0.0317    1.0000


pbuy =

    0.5927         0
         0         0
         0         0
         0    0.9000


psell =

         0    0.3000
    0.2666    0.3000
    0.1578    0.2000
    0.0683         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Obtaining Efficient Portfolios for Target Returns

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` function accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. For example, assume that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 12%:

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

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByReturn(p, [0.07,  0.10, .12]);
display(pwgt);

pwgt =

    0.7537    0.3899    0.1478
    0.1113    0.2934    0.4136
    0.0545    0.1006    0.1319
    0.0805    0.2161    0.3066
```

Sometimes, you can request a return for which no efficient portfolio exists. Based on the previous example, suppose that you want a portfolio with a 4% return (which is the return of the first asset). A portfolio that is fully invested in the first asset, however, is inefficient. `estimateFrontierByReturn` warns if your target returns are outside the range of efficient portfolio returns and replaces it with the endpoint portfolio of the efficient frontier closest to your target return:

```
 pwgt = estimateFrontierByReturn(p, [0.04]);

Warning: One or more target return values are outside the feasible range [
0.0591121, 0.182542 ].
    Will return portfolios associated with endpoints of the range for these values.
> In PortfolioMAD.estimateFrontierByReturn at 90
```

The best way to avoid this situation is to bracket your target portfolio returns with `estimateFrontierLimits` and `estimatePortReturn` (see “Obtaining Endpoints of

the Efficient Frontier" on page 6-100 and "Obtaining MAD Portfolio Risks and Returns" on page 6-111).

```
pret = estimatePortReturn(p, p.estimateFrontierLimits);

display(pret);
```

```
pret =

    0.0591
    0.1825
```

This result indicates that efficient portfolios have returns that range from 6.5% to 17.8%. Note, your results for these examples may be different due to the random generation of scenarios.

If you have an initial portfolio, `estimateFrontierByReturn` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, to obtain purchases and sales with target returns of 7%, 10%, and 12%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByReturn(p, [0.07,  0.10, .12]);


display(pwgt);
display(pbuy);
display(psell);
```

```
pwgt =

    0.7537    0.3899    0.1478
    0.1113    0.2934    0.4136
    0.0545    0.1006    0.1319
    0.0805    0.2161    0.3066


pbuy =

    0.4537    0.0899         0
         0         0    0.1136
         0         0         0
         0    0.1161    0.2066
```

```
psell =

         0         0    0.1522
    0.1887    0.0066         0
    0.1455    0.0994    0.0681
    0.0195         0         0
```

If you do not have an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Obtaining Efficient Portfolios for Target Risks

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` function accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Suppose that you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);

pwgt =

    0.2102    0.0621         0
    0.3957    0.4723    0.4305
    0.1051    0.1204    0.1291
    0.2889    0.3452    0.4404
```

Sometimes, you can request a risk for which no efficient portfolio exists. Based on the previous example, suppose that you want a portfolio with 6% risk (individual assets in this universe have risks ranging from 7% to 42.5%). It turns out that a portfolio with 6% risk cannot be formed with these four assets. `estimateFrontierByRisk` warns if your target risks are outside the range of efficient portfolio risks and replaces it with the endpoint of the efficient frontier closest to your target risk:

```
pwgt = estimateFrontierByRisk(p, 0.06)

Warning: One or more target risk values are outside the feasible range [
0.0610574, 0.278711 ].
    Will return portfolios associated with endpoints of the range for these values.
> In PortfolioMAD.estimateFrontierByRisk at 82

pwgt =
```

```
    0.8867
    0.0396
    0.0404
    0.0332
```

The best way to avoid this situation is to bracket your target portfolio risks with `estimateFrontierLimits` and `estimatePortRisk` (see "Obtaining Endpoints of the Efficient Frontier" on page 6-100 and "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111).

```
prsk = estimatePortRisk(p, p.estimateFrontierLimits);

display(prsk);

prsk =

    0.0611
    0.2787
```

This result indicates that efficient portfolios have risks that range from 7% to 42.5%. Note, your results for these examples may be different due to the random generation of scenarios.

Starting with an initial portfolio, `estimateFrontierByRisk` also returns purchases and sales to get from your initial portfolio to the target portfolios on the efficient frontier. For example, given an initial portfolio in `pwgt0`, you can obtain purchases and sales from the example with target risks of 12%, 14%, and 16%:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

display(pwgt);
display(pbuy);
display(psell);

pwgt =

    0.2102    0.0621         0
    0.3957    0.4723    0.4305
    0.1051    0.1204    0.1291
    0.2889    0.3452    0.4404


pbuy =
```

```
         0         0         0
    0.0957    0.1723    0.1305
         0         0         0
    0.1889    0.2452    0.3404


psell =

    0.0898    0.2379    0.3000
         0         0         0
    0.0949    0.0796    0.0709
         0         0         0
```

If you do not specify an initial portfolio, the purchase and sale weights assume that your initial portfolio is `0`.

See Also

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Choosing and Controlling the Solver

When solving portfolio optimizations for a PortfolioMAD object, while all variations of `fmincon` from Optimization Toolbox are supported, using `'sqp'` and `'active-set'` algorithms for `fmincon` is recommended and the use of `'interior-point'` algorithm is not recommended for MAD portfolio optimization.

Unlike Optimization Toolbox which uses the `'trust-region-reflective'` algorithm as the default algorithm for `fmincon`, the portfolio optimization for a PortfolioMAD object uses the `'active-set'` algorithm. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox).

To modify `fmincon` options for MAD portfolio optimizations, use `setSolver` to set the hidden properties `solverType` and `solverOptions` to specify and control the solver. Since these solver properties are hidden, you cannot set them using the `PortfolioMAD` function. The default solver is `fmincon` with the `'sqb'` algorithm and no displayed output, so you do not need to use `setSolver` to specify this.

If you want to specify additional options associated with the `fmincon` solver, `setSolver` accepts these options as name-value pair arguments. For example, if you want to use `fmincon` with the `sqp` algorithm and with displayed output, use `setSolver` with:

```
p = PortfolioMAD;
p = setSolver(p, 'fmincon', 'Algorithm', 'sqp', 'Display', 'final');
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
sqp
final
```

Alternatively, the `setSolver` function accepts an `optimoptions` object as the second argument. For example, you can change the algorithm to `trust-region-reflective` with no displayed output as follows:

```
p = PortfolioMAD;
options = optimoptions('fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
display(p.solverOptions.Display);
```

```
trust-region-reflective
off
```

See Also

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimatePortReturn` | `estimatePortRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21
- "Portfolio Optimization Theory" on page 6-3
- "PortfolioMAD Object Workflow" on page 6-19

Estimate Efficient Frontiers for PortfolioMAD Object

In this section...

Whereas "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96 focused on estimation of efficient portfolios, this section focuses on the estimation of efficient frontiers. For information on the workflow when using PortfolioMAD objects, see "PortfolioMAD Object Workflow" on page 6-19.

Obtaining MAD Portfolio Risks and Returns

Given any portfolio and, in particular, efficient portfolios, the functions `estimatePortReturn` and `estimatePortRisk` provide estimates for the return (or return proxy), risk (or the risk proxy). Each function has the same input syntax but with different combinations of outputs. Suppose that you have this following portfolio optimization problem that gave you a collection of portfolios along the efficient frontier in `pwgt`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = setInitPort(p, pwgt0);
pwgt = estimateFrontier(p)
```

```
pwgt =
```

```
Columns 1 through 8

  0.8954    0.7264    0.5573    0.3877    0.2176    0.0495    0.0000         0
  0.0310    0.1239    0.2154    0.3081    0.4028    0.4924    0.4069    0.2386
  0.0409    0.0524    0.0660    0.0792    0.0907    0.1047    0.1054    0.1132
  0.0328    0.0973    0.1613    0.2250    0.2890    0.3534    0.4877    0.6482

Columns 9 through 10

       0    0.0000
  0.0694    0.0000
  0.1221    0.0000
  0.8084    1.0000
```

Note Remember that the risk proxy for MAD portfolio optimization is mean-absolute deviation.

Given `pwgt0` and `pwgt`, use the portfolio risk and return estimation functions to obtain risks and returns for your initial portfolio and the portfolios on the efficient frontier:

```
prsk0 = estimatePortRisk(p, pwgt0);
pret0 = estimatePortReturn(p, pwgt0);
prsk = estimatePortRisk(p, pwgt);
pret = estimatePortReturn(p, pwgt);
display(prsk0);
display(pret0);
display(prsk);
display(pret);
```

You obtain these risks and returns:

```
prsk0 =

    0.0256


pret0 =

    0.0072


prsk =

    0.0178
    0.0193
    0.0233
```

```
    0.0286
    0.0348
    0.0414
    0.0489
    0.0584
    0.0692
    0.0809


pret =

    0.0047
    0.0059
    0.0072
    0.0084
    0.0096
    0.0108
    0.0120
    0.0133
    0.0145
    0.0157
```

Obtaining the PortfolioMAD Standard Deviation

The PortfolioMAD object has a function to compute standard deviations of portfolio returns, `estimatePortStd`. This function works with any portfolios, not necessarily efficient portfolios. For example, the following example obtains five portfolios (`pwgt`) on the efficient frontier and also has an initial portfolio in `pwgt0`. Various portfolio statistics are computed that include the return, risk, and standard deviation. The listed estimates are for the initial portfolio in the first row followed by estimates for each of the five efficient portfolios in subsequent rows.

```
m = [ 0.0042; 0.0083; 0.01; 0.15 ];
C = [ 0.005333 0.00034 0.00016 0;
0.00034 0.002408 0.0017 0.000992;
0.00016 0.0017 0.0048 0.0028;
0 0.000992 0.0028 0.010208 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD('initport', pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
```

```
pwgt = estimateFrontier(p, 5);

pret = estimatePortReturn(p, [pwgt0, pwgt]);
prsk = estimatePortRisk(p, [pwgt0, pwgt]);
pstd = estimatePortStd(p, [pwgt0, pwgt]);

[pret, prsk, pstd]

ans =

    0.0212    0.0305    0.0381
    0.0187    0.0326    0.0407
    0.0514    0.0369    0.0462
    0.0841    0.0484    0.0607
    0.1168    0.0637    0.0796
    0.1495    0.0807    0.1009
```

See Also

`PortfolioMAD` | `estimatePortReturn` | `estimatePortStd` | `plotFrontier`

Related Examples

- “Plotting the Efficient Frontier for a PortfolioMAD Object” on page 6-115
- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
- “Postprocessing Results to Set Up Tradable Portfolios” on page 6-122

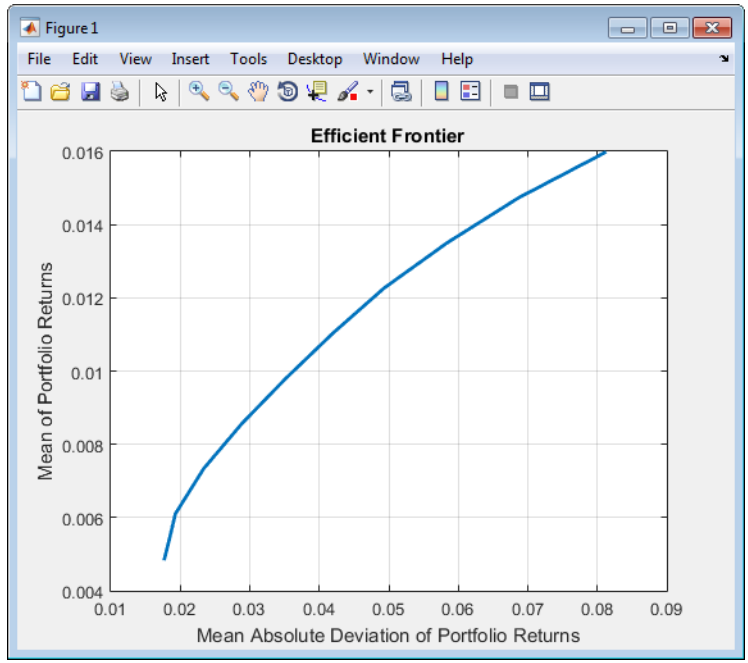
More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Plotting the Efficient Frontier for a PortfolioMAD Object

The `plotFrontier` function creates a plot of the efficient frontier for a given portfolio optimization problem. This function accepts several types of inputs and generates a plot with an optional possibility to output the estimates for portfolio risks and returns along the efficient frontier. `plotFrontier` has four different ways that it can be used. In addition to a plot of the efficient frontier, if you have an initial portfolio in the `InitPort` property, `plotFrontier` also displays the return versus risk of the initial portfolio on the same plot. If you have a well-posed portfolio optimization problem set up in a PortfolioMAD object and you use `plotFrontier`, you get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is currently 10 and is maintained in the hidden property `defaultNumPorts`). This example illustrates a typical use of `plotFrontier` to create a new plot:

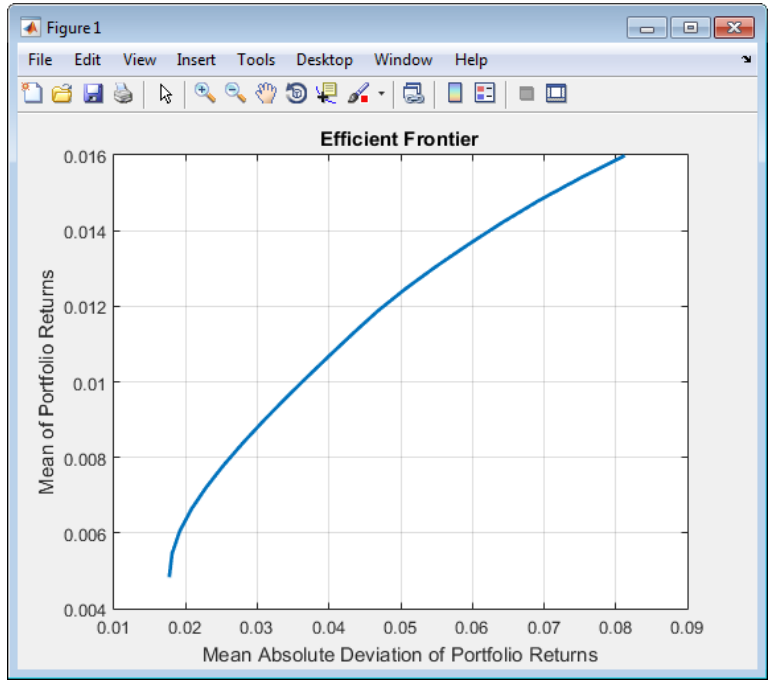
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

plotFrontier(p);
```



The `Name` property appears as the title of the efficient frontier plot if you set it in the PortfolioMAD object. Without an explicit name, the title on the plot would be "Efficient Frontier." If you want to obtain a specific number of portfolios along the efficient frontier, use `plotFrontier` with the number of portfolios that you want. Suppose that you have the PortfolioMAD object from the previous example and you want to plot 20 portfolios along the efficient frontier and to obtain 20 risk and return values for each portfolio:

```
[prsk, pret] = plotFrontier(p, 20);
display([pret, prsk]);

ans =

    0.0049    0.0176
    0.0054    0.0179
    0.0058    0.0189
    0.0063    0.0205
    0.0068    0.0225
    0.0073    0.0248
    0.0078    0.0274
    0.0083    0.0302
```

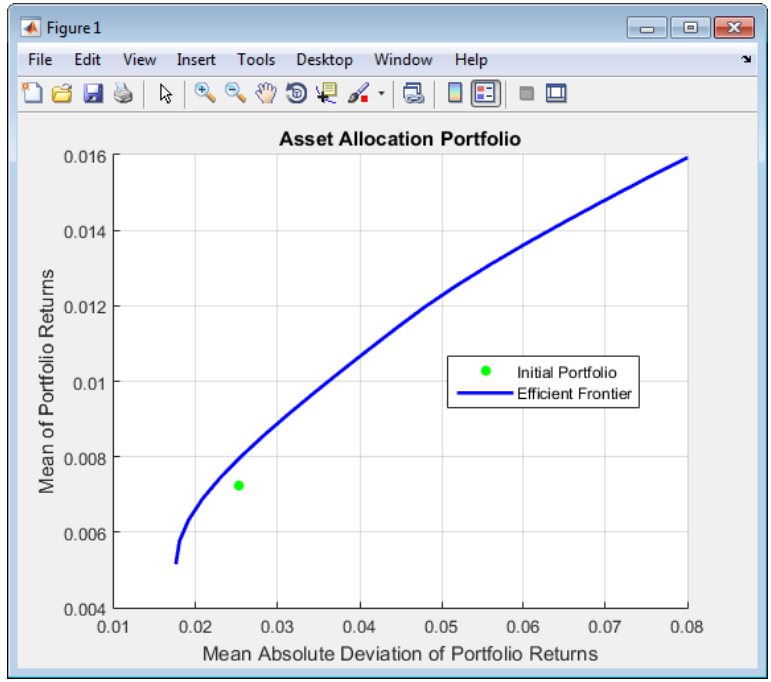
```
    0.0088    0.0331
    0.0093    0.0361
    0.0098    0.0392
    0.0103    0.0423
    0.0108    0.0457
    0.0112    0.0496
    0.0117    0.0539
    0.0122    0.0586
    0.0127    0.0635
    0.0132    0.0687
    0.0137    0.0744
    0.0142    0.0806
```



Plotting Existing Efficient Portfolios

If you already have efficient portfolios from any of the "estimateFrontier" functions (see “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111), pass them into `plotFrontier` directly to plot the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p, 20);
plotFrontier(p, pwgt)
```



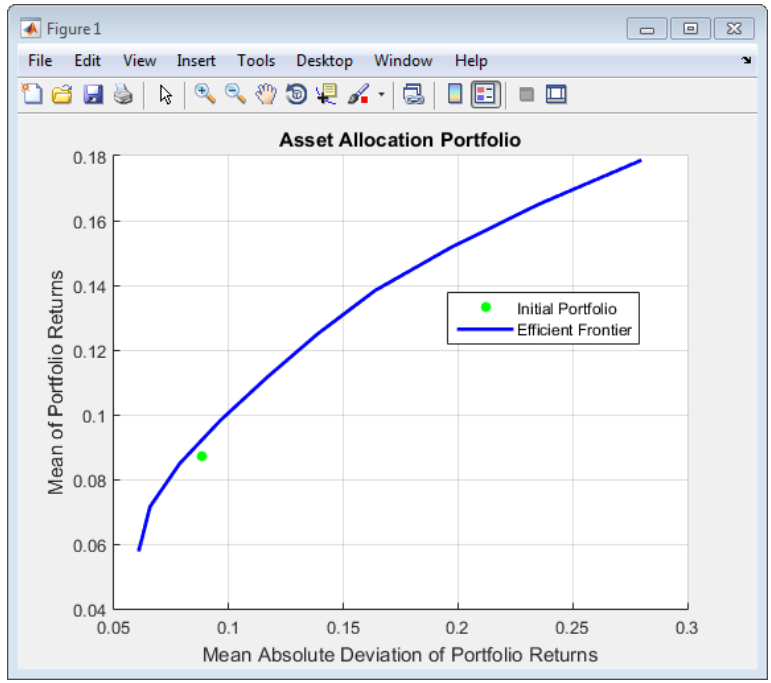
Plotting Existing Efficient Portfolio Risks and Returns

If you already have efficient portfolio risks and returns, you can use the interface to `plotFrontier` to pass them into `plotFrontier` to obtain a plot of the efficient frontier:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

AssetScenarios = mvnrnd(m, C, 20000);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD('Name', 'Asset Allocation Portfolio', 'InitPort', pwgt0);

p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p);

pret= estimatePortReturn(p, pwgt)
prsk = estimatePortRisk(p, pwgt)

plotFrontier(p, prsk, pret)
```

```
pret =

    0.0590
    0.0723
    0.0857
    0.0991
    0.1124
    0.1258
    0.1391
    0.1525
    0.1658
    0.1792


prsk =

    0.0615
    0.0664
    0.0795
    0.0976
    0.1184
    0.1408
```

```
0.1663
0.1992
0.2368
0.2787
```



See Also

`PortfolioMAD` | `estimatePortReturn` | `estimatePortStd` | `plotFrontier`

Related Examples

- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96

- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
- “Postprocessing Results to Set Up Tradable Portfolios” on page 6-122

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Postprocessing Results to Set Up Tradable Portfolios

After obtaining efficient portfolios or estimates for expected portfolio risks and returns, use your results to set up trades to move toward an efficient portfolio. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-19.

Setting Up Tradable Portfolios

Suppose that you set up a portfolio optimization problem and obtained portfolios on the efficient frontier. Use the `dataset` object from Statistics and Machine Learning Toolbox to form a blotter that lists your portfolios with the names for each asset. For example, suppose that you want to obtain five portfolios along the efficient frontier. You can set up a blotter with weights multiplied by 100 to view the allocations for each portfolio:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{100*pwgt},pnames],'obsnames',p.AssetList);
display(Blotter);
```

```
Blotter =

                        Port1     Port2     Port3     Port4     Port5
    Bonds               88.154    50.867    13.611         0    1.0609e-12
    Large-Cap Equities  4.0454    22.571    41.276     23.38    7.9362e-13
    Small-Cap Equities  4.2804    9.3108    14.028    17.878    6.4823e-14
    Emerging Equities   3.5202    17.252    31.084    58.743           100
```

Note Your results may differ from this result due to the simulation of scenarios.

This result indicates that you would invest primarily in bonds at the minimum-risk/minimum-return end of the efficient frontier (`Port1`), and that you would invest completely in emerging equity at the maximum-risk/maximum-return end of the efficient frontier (`Port5`). You can also select a particular efficient portfolio, for example, suppose that you want a portfolio with 15% risk and you add purchase and sale weights outputs obtained from the "estimateFrontier" functions to set up a trade blotter:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');

p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = p.setDefaultConstraints;

[pwgt, pbuy, psell] = estimateFrontierByRisk(p, 0.15);

Blotter = dataset([{100*[pwgt0, pwgt, pbuy, psell]}, ...
{'Initial','Weight', 'Purchases','Sales'}],'obsnames',p.AssetList);
display(Blotter);
```

```
Blotter =

                        Initial    Weight        Purchases    Sales
    Bonds               30         6.0364e-18          0          30
    Large-Cap Equities  30             50.179     20.179          0
    Small-Cap Equities  20              13.43          0     6.5696
    Emerging Equities   10             36.391     26.391          0
```

If you have prices for each asset (in this example, they can be ETFs), add them to your blotter and then use the tools of the `dataset` object to obtain shares and shares to be traded.

See Also

`PortfolioMAD` | `checkFeasibility` | `estimateScenarioMoments`

Related Examples

- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96

- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Working with Other Portfolio Objects

The PortfolioMAD object is for MAD portfolio optimization. The PortfolioCVaR object is for CVaR portfolio optimization. The Portfolio object is for mean-variance portfolio optimization. Sometimes, you might want to examine portfolio optimization problems according to different combinations of return and risk proxies. A common example is that you want to do a MAD portfolio optimization and then want to work primarily with moments of portfolio returns. Suppose that you set up a MAD portfolio optimization problem with:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

p = PortfolioMAD;
p = setAssetList(p, 'Bonds','Large-Cap Equities','Small-Cap Equities','Emerging Equities');
p = setInitPort(p, pwgt0);
p = simulateNormalScenariosByMoments(p, m, C, 20000);
p = setDefaultConstraints(p);
```

To work with the same problem in a mean-variance framework, you can use the scenarios from the PortfolioMAD object to set up a Portfolio object so that `p` contains a MAD optimization problem and `q` contains a mean-variance optimization problem based on the same data.

```
q = Portfolio('AssetList', p.AssetList);
q = estimateAssetMoments(q, p.getScenarios);
q = setDefaultConstraints(q);

pwgt = estimateFrontier(p);
qwgt = estimateFrontier(q);
```

Since each object has a different risk proxy, it is not possible to compare results side by side. To obtain means and standard deviations of portfolio returns, you can use the functions associated with each object to obtain:

```
pret = estimatePortReturn(p, pwgt);
pstd = estimatePortStd(p, pwgt);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);

[pret, qret]
[pstd, qstd]
```

```
ans =

    0.0592    0.0590
    0.0730    0.0728
    0.0868    0.0867
    0.1006    0.1005
    0.1145    0.1143
    0.1283    0.1282
    0.1421    0.1420
    0.1559    0.1558
    0.1697    0.1697
    0.1835    0.1835


ans =

    0.0767    0.0767
    0.0829    0.0828
    0.0989    0.0987
    0.1208    0.1206
    0.1461    0.1459
    0.1732    0.1730
    0.2042    0.2040
    0.2453    0.2452
    0.2929    0.2928
    0.3458    0.3458
```

To produce comparable results, you can use the returns or risks from one portfolio optimization as target returns or risks for the other portfolio optimization.

```
qwgt = estimateFrontierByReturn(q, pret);
qret = estimatePortReturn(q, qwgt);
qstd = estimatePortStd(q, qwgt);

[pret, qret]
[pstd, qstd]

ans =

    0.0592    0.0592
    0.0730    0.0730
    0.0868    0.0868
    0.1006    0.1006
    0.1145    0.1145
    0.1283    0.1283
```

```
    0.1421    0.1421
    0.1559    0.1559
    0.1697    0.1697
    0.1835    0.1835


ans =

    0.0767    0.0767
    0.0829    0.0829
    0.0989    0.0989
    0.1208    0.1208
    0.1461    0.1461
    0.1732    0.1732
    0.2042    0.2042
    0.2453    0.2453
    0.2929    0.2929
    0.3458    0.3458
```

Now it is possible to compare standard deviations of portfolio returns from either type of portfolio optimization.

See Also

`Portfolio` | `PortfolioMAD`

Related Examples

- "Creating the Portfolio Object" on page 4-28
- "Creating the PortfolioMAD Object" on page 6-26
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

More About

- "PortfolioMAD Object" on page 6-21

- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Troubleshooting MAD Portfolio Optimization Results

PortfolioMAD Object Destroyed When Modifying

If a PortfolioMAD object is destroyed when modifying, remember to pass an existing object into the `PortfolioMAD` function if you want to modify it, otherwise it creates a new object. See “Creating the PortfolioMAD Object” on page 6-26 for details.

Matrix Incompatibility and "Non-Conformable" Errors

If you get matrix incompatibility or "non-conformable" errors, the representation of data in the tools follows a specific set of basic rules described in “Conventions for Representation of Data” on page 6-24.

Missing Data Estimation Fails

If asset return data has missing or `NaN` values, the `simulateNormalScenariosByData` function with the `'missingdata'` flag set to `true` may fail with either too many iterations or a singular covariance. To correct this problem, consider this:

- If you have asset return data with no missing or `NaN` values, you can compute a covariance matrix that may be singular without difficulties. If you have missing or `NaN` values in your data, the supported missing data feature requires that your covariance matrix must be positive-definite, that is, nonsingular.
- `simulateNormalScenariosByData` uses default settings for the missing data estimation procedure that might not be appropriate for all problems.

In either case, you might want to estimate the moments of asset returns separately with either the ECM estimation functions such as `ecmnmle` or with your own functions.

mad_optim_transform Errors

If you obtain optimization errors such as:

```
Error using mad_optim_transform (line 276)
Portfolio set appears to be either empty or unbounded. Check constraints.

Error in PortfolioMAD/estimateFrontier (line 64)
    [AI, bI, AE, bE, lB, uB, f0, f, x0] = mad_optim_transform(obj);
```

or

```
Error using mad_optim_transform (line 281)
Cannot obtain finite lower bounds for specified portfolio set.

Error in PortfolioMAD/estimateFrontier (line 64)
    [AI, bI, AE, bE, lB, uB, f0, f, x0] = mad_optim_transform(obj);
```

Since the portfolio optimization tools require a bounded portfolio set, these errors (and similar errors) can occur if your portfolio set is either empty and, if nonempty, unbounded. Specifically, the portfolio optimization algorithm requires that your portfolio set have at least a finite lower bound. The best way to deal with these problems is to use the validation methods in "Validate the MAD Portfolio Problem" on page 6-91. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

If you obtain efficient portfolios that, do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between `0` and `1` and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the PortfolioMAD object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

See Also

`PortfolioMAD` | `checkFeasibility` | `estimateScenarioMoments`

Related Examples

- "Postprocessing Results to Set Up Tradable Portfolios" on page 6-122

- “Creating the PortfolioMAD Object” on page 6-26
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

More About

- “PortfolioMAD Object” on page 6-21
- “Portfolio Optimization Theory” on page 6-3
- “PortfolioMAD Object Workflow” on page 6-19

Investment Performance Metrics

Performance Metrics Overview

Performance Metrics Types

Sharpe first proposed a ratio of excess return to total risk as an investment performance metric. Subsequent work by Sharpe, Lintner, and Mossin extended these ideas to entire asset markets in what is called the Capital Asset Pricing Model (CAPM). Since the development of the CAPM, various investment performance metrics has evolved.

This section presents four types of investment performance metrics:

- The first type of metrics is absolute investment performance metrics that are called "classic" metrics since they are based on the CAPM. They include the Sharpe ratio, the information ratio, and tracking error. To compute the Sharpe ratio from data, use `sharpe` to calculate the ratio for one or more asset return series. To compute the information ratio and associated tracking error, use `inforatio` to calculate these quantities for one or more asset return series.
- The second type of metrics is relative investment performance metrics to compute risk-adjusted returns. These metrics are also based on the CAPM and include Beta, Jensen's Alpha, the Security Market Line (SML), Modigliani and Modigliani Risk-Adjusted Return, and the Graham-Harvey measures. To calculate risk-adjusted alpha and return, use `portalpha`.
- The third type of metrics is alternative investment performance metrics based on lower partial moments. To calculate lower partial moments, use `lpm` for sample lower partial moments and `elpm` for expected lower partial moments.
- The fourth type of metrics is performance metrics based on maximum drawdown and expected maximum drawdown. To calculate maximum or expected maximum drawdowns, use `maxdrawdown` and `emaxdrawdown`.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- "Using the Sharpe Ratio" on page 7-6

- "Using the Information Ratio" on page 7-8
- "Using Tracking Error" on page 7-10
- "Using Risk-Adjusted Return" on page 7-12
- "Using Sample and Expected Lower Partial Moments" on page 7-15
- "Using Maximum and Expected Maximum Drawdown" on page 7-18

Performance Metrics Illustration

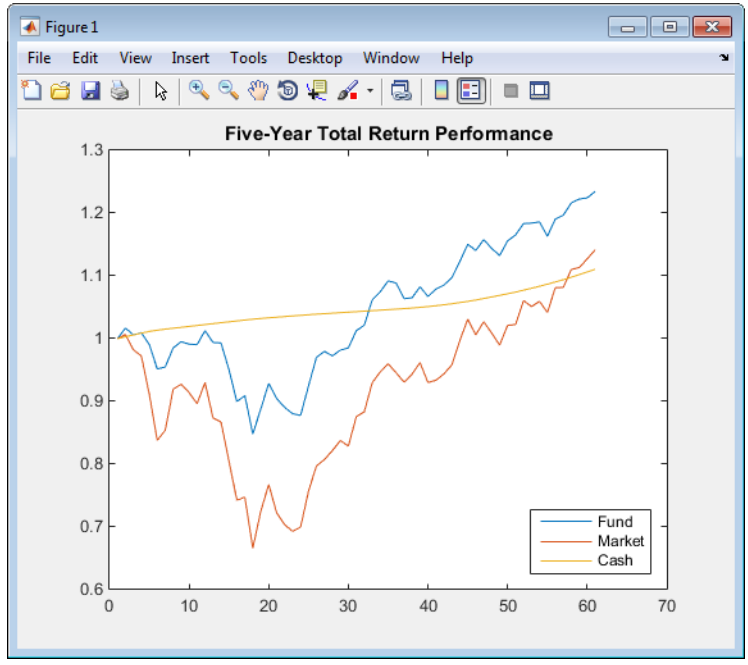
To illustrate the functions for investment performance metrics, you work with three financial time series objects using performance data for:

- An actively managed, large-cap value mutual fund
- A large-cap market index
- 90-day Treasury bills

The data is monthly total return prices that cover a span of five years.

The following plot illustrates the performance of each series in terms of total returns to an initial $1 invested at the start of this 5-year period:

```
load FundMarketCash
plot(TestData)
hold on
title('\bfFive-Year Total Return Performance');
legend('Fund','Market','Cash','Location','SouthEast');
hold off
```



The mean (`Mean`) and standard deviation (`Sigma`) of returns for each series are

```
Returns = tick2ret(TestData);
Assets
Mean = mean(Returns)
Sigma = std(Returns, 1)
```

which gives the following result:

```
Assets =
    'Fund'    'Market'    'Cash'
Mean =
    0.0038    0.0030    0.0017
Sigma =
    0.0229    0.0389    0.0009
```

Note Functions for investment performance metrics use total return price and total returns. To convert between total return price and total returns, use `ret2tick` and `tick2ret`.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- “Using the Sharpe Ratio” on page 7-6
- “Using the Information Ratio” on page 7-8
- “Using Tracking Error” on page 7-10
- “Using Risk-Adjusted Return” on page 7-12
- “Using Sample and Expected Lower Partial Moments” on page 7-15
- “Using Maximum and Expected Maximum Drawdown” on page 7-18

Using the Sharpe Ratio

In this section...

Introduction

The Sharpe ratio is the ratio of the excess return of an asset divided by the asset's standard deviation of returns. The Sharpe ratio has the form:

```
(Mean - Riskless) / Sigma
```

Here `Mean` is the mean of asset returns, `Riskless` is the return of a riskless asset, and `Sigma` is the standard deviation of asset returns. A higher Sharpe ratio is better than a lower Sharpe ratio. A negative Sharpe ratio indicates “anti-skill” since the performance of the riskless asset is superior. For more information, see `sharpe`.

Sharpe Ratio

To compute the Sharpe ratio, the mean return of the cash asset is used as the return for the riskless asset. Thus, given asset return data and the riskless asset return, the Sharpe ratio is calculated with

```
load FundMarketCash
Returns = tick2ret(TestData);
Riskless = mean(Returns(:,3))
Sharpe = sharpe(Returns, Riskless)
```

which gives the following result:

```
Riskless =
    0.0017
Sharpe =
    0.0886    0.0315         0
```

The Sharpe ratio of the example fund is significantly higher than the Sharpe ratio of the market. As is demonstrated with `portalpha`, this translates into a strong risk-adjusted return. Since the `Cash` asset is the same as `Riskless`, it makes sense that its Sharpe

ratio is 0. The Sharpe ratio was calculated with the mean of cash returns. It can also be calculated with the cash return series as input for the riskless asset

```
Sharpe = sharpe(Returns, Returns(:,3))
```

which gives the following result:

```
Sharpe =
    0.0886    0.0315         0
```

When using the `Portfolio` object, you can use the `estimateMaxSharpeRatio` function to estimate an efficient portfolio that maximizes the Sharpe ratio. For more information, see “Efficient Portfolio That Maximizes Sharpe Ratio” on page 4-123.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Information Ratio” on page 7-8
- “Using Tracking Error” on page 7-10
- “Using Risk-Adjusted Return” on page 7-12
- “Using Sample and Expected Lower Partial Moments” on page 7-15
- “Using Maximum and Expected Maximum Drawdown” on page 7-18

Using the Information Ratio

In this section...

Introduction

Although originally called the “appraisal ratio” by Treynor and Black, the information ratio is the ratio of relative return to relative risk (known as “tracking error”). Whereas the Sharpe ratio looks at returns relative to a riskless asset, the information ratio is based on returns relative to a risky benchmark which is known colloquially as a “bogey.” Given an asset or portfolio of assets with random returns designated by `Asset` and a benchmark with random returns designated by `Benchmark`, the information ratio has the form:

```
Mean(Asset - Benchmark) / Sigma (Asset - Benchmark)
```

Here `Mean(Asset - Benchmark)` is the mean of `Asset` minus `Benchmark` returns, and `Sigma(Asset - Benchmark)` is the standard deviation of `Asset` minus `Benchmark` returns. A higher information ratio is considered better than a lower information ratio. For more information, see `inforatio`.

Information Ratio

To calculate the information ratio using the example data, the mean return of the market series is used as the return of the benchmark. Thus, given asset return data and the riskless asset return, compute the information ratio with

```
load FundMarketCash
Returns = tick2ret(TestData);
Benchmark = Returns(:,2);
InfoRatio = inforatio(Returns, Benchmark)
```

which gives the following result:

```
InfoRatio =
    0.0432       NaN   -0.0315
```

Since the market series has no risk relative to itself, the information ratio for the second series is undefined (which is represented as `NaN` in MATLAB software). Its standard deviation of relative returns in the denominator is 0.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- "Performance Metrics Overview" on page 7-2
- "Using the Sharpe Ratio" on page 7-6
- "Using Tracking Error" on page 7-10
- "Using Risk-Adjusted Return" on page 7-12
- "Using Sample and Expected Lower Partial Moments" on page 7-15
- "Using Maximum and Expected Maximum Drawdown" on page 7-18

Using Tracking Error

In this section...

Introduction

Given an asset or portfolio of assets and a benchmark, the relative standard deviation of returns between the asset or portfolio of assets and the benchmark is called tracking error.

Tracking Error

The function `inforatio` computes tracking error and returns it as a second argument

```
load FundMarketCash
Returns = tick2ret(TestData);
Benchmark = Returns(:,2);
[InfoRatio, TrackingError] = inforatio(Returns, Benchmark)
```

which gives the following results:

```
InfoRatio =
    0.0432       NaN   -0.0315
TrackingError =
    0.0187         0    0.0390
```

Tracking error is a useful measure of performance relative to a benchmark since it is in units of asset returns. For example, the tracking error of 1.87% for the fund relative to the market in this example is reasonable for an actively managed, large-cap value fund.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-6
- “Using the Information Ratio” on page 7-8
- “Using Risk-Adjusted Return” on page 7-12
- “Using Sample and Expected Lower Partial Moments” on page 7-15
- “Using Maximum and Expected Maximum Drawdown” on page 7-18

Using Risk-Adjusted Return

In this section...

Introduction

Risk-adjusted return either shifts the risk (which is the standard deviation of returns) of a portfolio to match the risk of a market portfolio or shifts the risk of a market portfolio to match the risk of a fund. According to the Capital Asset Pricing Model (CAPM), the market portfolio and a riskless asset are points on a Security Market Line (SML). The return of the resultant shifted portfolio, levered or unlevered, to match the risk of the market portfolio, is the risk-adjusted return. The SML provides another measure of risk-adjusted return, since the difference in return between the fund and the SML, return at the same level of risk.

Risk-Adjusted Return

Given our example data with a fund, a market, and a cash series, you can calculate the risk-adjusted return and compare it with the fund and market's mean returns

```
load FundMarketCash
Returns = tick2ret(TestData);
Fund = Returns(:,1);
Market = Returns(:,2);
Cash = Returns(:,3);
MeanFund = mean(Fund)
MeanMarket = mean(Market)

[MM, aMM] = portalpha(Fund, Market, Cash, 'MM')
[GH1, aGH1] = portalpha(Fund, Market, Cash, 'gh1')
[GH2, aGH2] = portalpha(Fund, Market, Cash, 'gh2')
[SML, aSML] = portalpha(Fund, Market, Cash, 'sml')
```

which gives the following results:

```
MeanFund =

    0.0038
```

```
MeanMarket =

    0.0030

MM =

    0.0022

aMM =

    0.0052

GH1 =

    0.0013

aGH1 =

    0.0025

GH2 =

    0.0022

aGH2 =

    0.0052

SML =

    0.0013

aSML =

    0.0025
```

Since the fund's risk is much less than the market's risk, the risk-adjusted return of the fund is much higher than both the nominal fund and market returns.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- "Performance Metrics Overview" on page 7-2
- "Using the Sharpe Ratio" on page 7-6
- "Using the Information Ratio" on page 7-8
- "Using Tracking Error" on page 7-10
- "Using Sample and Expected Lower Partial Moments" on page 7-15
- "Using Maximum and Expected Maximum Drawdown" on page 7-18

Using Sample and Expected Lower Partial Moments

In this section...

Introduction

Use lower partial moments to examine what is colloquially known as "downside risk." The main idea of the lower partial moment framework is to model moments of asset returns that fall below a minimum acceptable level of return. To compute lower partial moments from data, use `lpm` to calculate lower partial moments for multiple asset return series and for multiple moment orders. To compute expected values for lower partial moments under several assumptions about the distribution of asset returns, use `elpm` to calculate lower partial moments for multiple assets and for multiple orders.

Sample Lower Partial Moments

The following example demonstrates `lpm` to compute the zero-order, first-order, and second-order lower partial moments for the three time series, where the mean of the third time series is used to compute `MAR` (minimum acceptable return) with the so-called risk-free rate.

```
load FundMarketCash
Returns = tick2ret(TestData);
Assets
MAR = mean(Returns(:,3))
LPM = lpm(Returns, MAR, [0 1 2])
```

which gives the following results:

```
Assets =
    'Fund'    'Market'    'Cash'
MAR =
    0.0017
LPM =
    0.4333    0.4167    0.6167
    0.0075    0.0140    0.0004
    0.0003    0.0008    0.0000
```

The first row of LPM contains zero-order lower partial moments of the three series. The fund and market index fall below MAR about 40% of the time and cash returns fall below its own mean about 60% of the time.

The second row contains first-order lower partial moments of the three series. The fund and market have large average shortfall returns relative to MAR by 75 and 140 basis points per month. On the other hand, cash underperforms MAR by about only four basis points per month on the downside.

The third row contains second-order lower partial moments of the three series. The square root of these quantities provides an idea of the dispersion of returns that fall below the MAR. The market index has a much larger variation on the downside when compared to the fund.

Expected Lower Partial Moments

To compare realized values with expected values, use elpm to compute expected lower partial moments based on the mean and standard deviations of normally distributed asset returns. The elpm function works with the mean and standard deviations for multiple assets and multiple orders.

```
load FundMarketCash
Returns = tick2ret(TestData);
MAR = mean(Returns(:,3))
Mean = mean(Returns)
Sigma = std(Returns, 1)
Assets
ELPM = elpm(Mean, Sigma, MAR, [0 1 2])
```

which gives the following results:

```
Assets =
    'Fund'    'Market'    'Cash'
ELPM =
    0.4647    0.4874    0.5000
    0.0082    0.0149    0.0004
    0.0002    0.0007    0.0000
```

Based on the moments of each asset, the expected values for lower partial moments imply better than expected performance for the fund and market and worse than expected performance for cash. This function works with either degenerate or

nondegenerate normal random variables. For example, if cash were truly riskless, its standard deviation would be 0. You can examine the difference in average shortfall.

```
RisklessCash = elpm(Mean(3), 0, MAR, 1)
```

which gives the following result:

```
RisklessCash =
     0
```

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-6
- “Using the Information Ratio” on page 7-8
- “Using Tracking Error” on page 7-10
- “Using Risk-Adjusted Return” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-18

Using Maximum and Expected Maximum Drawdown

In this section...

Introduction

Maximum drawdown is the maximum decline of a series, measured as return, from a peak to a nadir over a period of time. Although additional metrics exist that are used in the hedge fund and commodity trading communities (see Pederson and Rudholm-Alfvin [20] in “Bibliography” on page A-2), the original definition and subsequent implementation of these metrics is not yet standardized.

It is possible to compute analytically the expected maximum drawdown for a Brownian motion with drift (see Magdon-Ismail, Atiya, Pratap, and Abu-Mostafa [16] “Bibliography” on page A-2). These results are used to estimate the expected maximum drawdown for a series that approximately follows a geometric Brownian motion.

Use `maxdrawdown` and `emaxdrawdown` to calculate the maximum and expected maximum drawdowns.

Maximum Drawdown

This example demonstrates how to compute the maximum drawdown (`MaxDD`) using our example data with a fund, a market, and a cash series:

```
load FundMarketCash
MaxDD = maxdrawdown(TestData)
```

which gives the following results:

```
MaxDD =
    0.1658    0.3381         0
```

The maximum drop in the given time period was of 16.58% for the fund series, and 33.81% for the market. There was no decline in the cash series, as expected, because the cash account never loses value.

`maxdrawdown` can also return the indices (`MaxDDIndex`) of the maximum drawdown intervals for each series in an optional output argument:

```
[MaxDD, MaxDDIndex] = maxdrawdown(TestData)
```

which gives the following results:

```
MaxDD =

    0.1658    0.3381         0


MaxDDIndex =

     2     2   NaN
    18    18   NaN
```

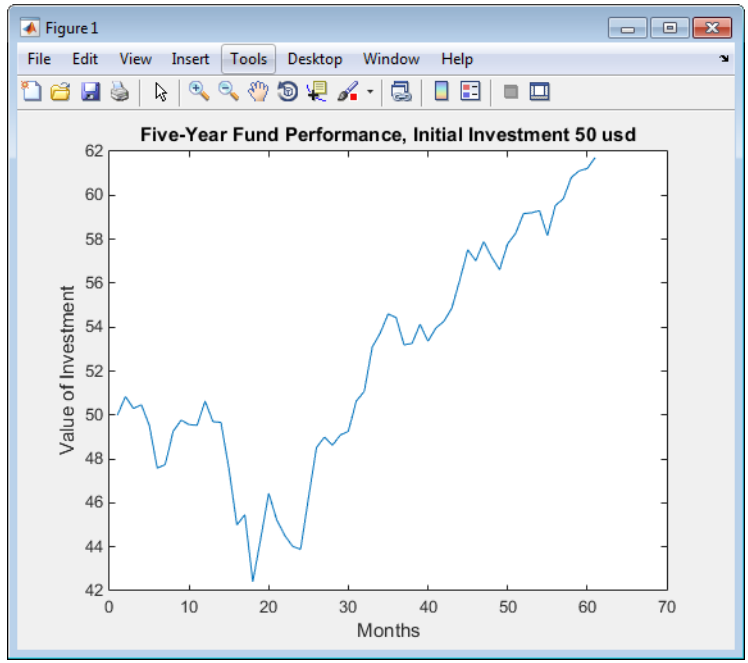
The first two series experience their maximum drawdowns from the second to the 18th month in the data. The indices for the third series are `NaN`s because it never has a drawdown.

The 16.58% value loss from month 2 to month 18 for the fund series is verified using the reported indices:

```
Start = MaxDDIndex(1,:);
End = MaxDDIndex(2,:);
(TestData(Start(1),1) - TestData(End(1),1))/TestData(Start(1),1)

ans =

    0.1658
```

Although the maximum drawdown is measured in terms of returns, `maxdrawdown` can measure the drawdown in terms of absolute drop in value, or in terms of log-returns. To contrast these alternatives more clearly, we work with the fund series assuming, an initial investment of 50 dollars:

```
Fund50 = 50*TestData(:,1);
plot(Fund50);
title('\bfFive-Year Fund Performance, Initial Investment 50 usd');
xlabel('Months');
ylabel('Value of Investment');
```



First, we compute the standard maximum drawdown, which coincides with the results above because returns are independent of the initial amounts invested:

```
MaxDD50Ret = maxdrawdown(Fund50)
```

```
MaxDD50Ret =

    0.1658
```

Next, we compute the maximum drop in value, using the `arithmetic` argument:

```
[MaxDD50Arith, Ind50Arith] = maxdrawdown(Fund50,'arithmetic')
```

```
MaxDD50Arith =

    8.4285


Ind50Arith =
```

```
     2
    18
```

The value of this investment was $50.84 in month 2, but by month 18 the value was down to $42.41, a drop of $8.43. This is the largest loss in dollar value from a previous high in the given time period. In this case, the maximum drawdown period, 2nd to 18th month, is the same independently of whether drawdown is measured as return or as dollar value loss.

Last, we compute the maximum decline based on log-returns using the `geometric` argument. In this example, the log-returns result in a maximum drop of 18.13%, again from the second to the 18th month, not far from the 16.58% obtained using standard returns.

```
[MaxDD50LogRet, Ind50LogRet] = maxdrawdown(Fund50,'geometric')

MaxDD50LogRet =

    0.1813


Ind50LogRet =

     2
    18
```

Note, the last measure is equivalent to finding the arithmetic maximum drawdown for the log of the series:

```
MaxDD50LogRet2 = maxdrawdown(log(Fund50),'arithmetic')

MaxDD50LogRet2 =

    0.1813
```

Expected Maximum Drawdown

This example demonstrates using the log-return moments of the fund to compute the expected maximum drawdown (`EMaxDD`) and then compare it with the realized maximum drawdown (`MaxDD`).

```
load FundMarketCash
logReturns = log(TestData(2:end,:) ./ TestData(1:end - 1,:));
```

```
Mu = mean(logReturns(:,1));
Sigma = std(logReturns(:,1),1);
T = size(logReturns,1);

MaxDD = maxdrawdown(TestData(:,1),'geometric')
EMaxDD = emaxdrawdown(Mu-0.5*Sigma^2, Sigma, T)
```

which gives the following results:

```
MaxDD =

    0.1813

EMaxDD =

    0.1588
```

The drawdown observed in this time period is above the expected maximum drawdown. There is no contradiction here. The expected maximum drawdown is not an upper bound on the maximum losses from a peak, but an estimate of their average, based on a geometric Brownian motion assumption.

See Also

`elpm` | `emaxdrawdown` | `inforatio` | `lpm` | `maxdrawdown` | `portalpha` | `ret2tick` | `sharpe` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-6
- “Using the Information Ratio” on page 7-8
- “Using Tracking Error” on page 7-10
- “Using Risk-Adjusted Return” on page 7-12
- “Using Sample and Expected Lower Partial Moments” on page 7-15

Credit Risk Analysis

Estimation of Transition Probabilities

In this section...

Introduction

Credit ratings rank borrowers according to their credit worthiness. Though this ranking is, in itself, useful, institutions are also interested in knowing how likely it is that borrowers in a particular rating category will be upgraded or downgraded to a different rating, and especially, how likely it is that they will default.

Transition probabilities offer one-way to characterize the past changes in credit quality of obligors (typically firms), and are cardinal inputs to many risk management applications. Financial Toolbox software supports the estimation of transition probabilities using both cohort and duration (also known as hazard rate or intensity) approaches using `transprob` and related functions.

Note The sample dataset used throughout this section is simulated using a single transition matrix. No attempt is made to match historical trends in transition rates.

Estimate Transition Probabilities

The `Data_TransProb.mat` file contains sample credit ratings data.

```
load Data_TransProb
data(1:10,:)

ans =

        ID            Date          Rating
    ___________    ______________    _______

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '12-May-1986'    'B'
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'
    '00013326'    '09-Feb-1985'    'A'
    '00013326'    '24-Feb-1994'    'AA'
    '00013326'    '10-Nov-2000'    'BBB'
    '00014413'    '23-Dec-1982'    'B'
    '00014413'    '20-Apr-1988'    'BB'
    '00014413'    '16-Jan-1998'    'B'
```

The sample data is formatted as a cell array with three columns. Each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). The assigned credit rating corresponds to the associated ID on the associated date. All information corresponding to the same ID must be stored in contiguous rows. In this example, IDs, dates, and ratings are stored in character vector format, but you also can enter them in numeric format.

In this example, the simplest calling syntax for `transprob` passes the `nRecords`-by-3 cell array as the only input argument. The default `startDate` and `endDate` are the earliest and latest dates in the data. The default estimation algorithm is the duration method and 1-year transition probabilities are estimated:

```
transMat0 = transprob(data)

transMat0 =

93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001    0.0017
 1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004    0.0396
 0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028    0.0753
 0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642    0.2193
 0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919    0.7050
 0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169    2.4399
 0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927   12.7167
      0         0         0         0         0         0         0  100.0000
```

It is recommended to provide explicit start and end dates. Otherwise the estimation window for two different datasets can differ, and the estimates might not be comparable. From this point, assume that the time window of interest is the 5-year period from the

end of 1995 to the end of 2000. For comparisons, compute the estimates for this time window. First use the `duration` algorithm (default option), and then the `cohort` algorithm explicitly set.

```
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
transMat1 = transprob(data,'startDate',startDate,'endDate',endDate)
transMat2 = transprob(data,'startDate',startDate,'endDate',endDate,...
'algorithm','cohort')
```

```
transMat1 =

90.6236    7.9051    1.0314    0.4123    0.0210    0.0020    0.0003    0.0043
 4.4780   89.5558    4.5298    1.1225    0.2284    0.0094    0.0009    0.0754
 0.3983    6.1164   87.0641    5.4801    0.7637    0.0892    0.0050    0.0832
 0.1029    0.8572   10.7918   83.0204    3.9971    0.7001    0.1313    0.3992
 0.1043    0.3745    2.2962   14.0954   78.9840    3.0013    0.0463    1.0980
 0.0113    0.0544    0.7055    3.2925   15.4350   75.5988    1.8166    3.0860
 0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
      0         0         0         0         0         0         0  100.0000

transMat2 =

90.1554    8.5492    0.9067    0.3886         0         0         0         0
 4.9512   88.5221    5.1763    1.0503    0.2251         0         0    0.0750
 0.2770    6.6482   86.2188    6.0942    0.6233    0.0693         0    0.0693
 0.0794    0.8737   11.6759   81.6521    4.3685    0.7943    0.1589    0.3971
 0.1002    0.4008    1.9038   15.4309   77.8557    3.4068         0    0.9018
      0         0    0.2262    2.4887   17.4208   74.2081    2.2624    3.3937
      0         0    0.7576    1.5152    6.0606   10.6061   75.0000    6.0606
      0         0         0         0         0         0         0  100.0000
```

By default, the `cohort` algorithm internally gets yearly snapshots of the credit ratings, but the number of snapshots per year is definable using the parameter/value pair `snapsPerYear`. To get the estimates using quarterly snapshots:

```
transMat3 = transprob(data,'startDate',startDate,'endDate',endDate,...
'algorithm','cohort','snapsPerYear',4)
```

```
transMat3 =

90.4765    8.0881    1.0072    0.4069    0.0164    0.0015    0.0002    0.0032
 4.5949   89.3216    4.6489    1.1239    0.2276    0.0074    0.0007    0.0751
 0.3747    6.3158   86.7380    5.6344    0.7675    0.0856    0.0040    0.0800
 0.0958    0.7967   11.0441   82.6138    4.1906    0.7230    0.1372    0.3987
 0.1028    0.3571    2.3312   14.4954   78.4276    3.1489    0.0383    1.0987
 0.0084    0.0399    0.6465    3.0962   16.0789   75.1300    1.9044    3.0956
 0.0031    0.0125    0.1445    1.8759    6.2613   10.7022   75.6300    5.3705
      0         0         0         0         0         0         0  100.0000
```

Both `duration` and `cohort` compute 1-year transition probabilities by default, but the time interval for the transitions is definable using the parameter/value pair `transInterval`. For example, to get the 2-year transition probabilities using the `cohort` algorithm with the same snapshot periodicity and estimation window:

```
transMat4 = transprob(data,'startDate',startDate,'endDate',endDate,...
'algorithm','cohort','snapsPerYear',4,'transInterval',2)
```

```
transMat4 =

82.2358   14.6092    2.2062    0.8543    0.0711    0.0074    0.0011    0.0149
 8.2803   80.4584    8.3606    2.2462    0.4665    0.0316    0.0030    0.1533
 0.9604   11.1975   76.1729    9.7284    1.5322    0.2044    0.0162    0.1879
 0.2483    2.0903   18.8440   69.5145    6.9601    1.2966    0.2329    0.8133
 0.2129    0.8713    5.4893   23.5776   62.6438    4.9464    0.1390    2.1198
 0.0378    0.1895    1.7679    7.2875   24.9444   57.1783    2.8816    5.7132
 0.0154    0.0716    0.6576    4.2157   11.4465   16.3455   57.4078    9.8399
      0         0         0         0         0         0         0  100.0000
```

Estimate Transition Probabilities for Different Rating Scales

The dataset `data` from `Data_TransProb.mat` contains sample credit ratings using the default rating scale `{'AAA', 'AA','A', 'BBB', 'BB', 'B', 'CCC', 'D'}`. It also contains the dataset `dataIGSG` with ratings investment grade (`'IG'`), speculative grade (`'SG'`), and default (`'D'`). To estimate the transition matrix for this dataset, use the `labels` argument.

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
dataIGSG(1:10,:)
transMatIGSG = transprob(dataIGSG,'labels',{'IG','SG','D'},...
'startDate',startDate,'endDate',endDate)

ans =

    '00011253'    '04-Apr-1983'    'IG'
    '00012751'    '17-Feb-1985'    'SG'
    '00012751'    '19-May-1986'    'D'
    '00014690'    '17-Jan-1983'    'IG'
    '00012144'    '21-Nov-1984'    'IG'
    '00012144'    '25-Mar-1992'    'SG'
    '00012144'    '07-May-1994'    'IG'
    '00012144'    '23-Jan-2000'    'SG'
    '00012144'    '20-Aug-2001'    'IG'
    '00012937'    '07-Feb-1984'    'IG'

transMatIGSG =

   98.1986    1.5179    0.2835
    8.5396   89.4891    1.9713
         0         0  100.0000
```

There is another dataset, `dataIGSGnum`, with the same information as `dataIGSG`, except the ratings are mapped to a numeric scale where `'IG'=1`, `'SG'=2`, and `'D'=3`. To estimate the transition matrix, use the `labels` optional argument specifying the numeric scale as a cell array.

```
dataIGSGnum(1:10,:)
% Note {1,2,3} and num2cell(1:3) are equivalent; num2cell is convenient
% when the number of ratings is larger
transMatIGSGnum = transprob(dataIGSGnum,'labels',{1,2,3},...
'startDate',startDate,'endDate',endDate)
```

```
ans =

    '00011253'    '04-Apr-1983'    [1]
    '00012751'    '17-Feb-1985'    [2]
    '00012751'    '19-May-1986'    [3]
    '00014690'    '17-Jan-1983'    [1]
    '00012144'    '21-Nov-1984'    [1]
    '00012144'    '25-Mar-1992'    [2]
    '00012144'    '07-May-1994'    [1]
    '00012144'    '23-Jan-2000'    [2]
    '00012144'    '20-Aug-2001'    [1]
    '00012937'    '07-Feb-1984'    [1]

transMatIGSGnum =

   98.1986    1.5179    0.2835
    8.5396   89.4891    1.9713
         0         0  100.0000
```

Any time the input dataset contains ratings not included in the default rating scale `{'AAA', 'AA', 'A', 'BBB', 'BB', 'B', 'CCC', 'D'}`, the full rating scale must be specified using the `labels` optional argument. For example, if the dataset contains ratings `'AAA', ..., 'CCC', 'D'`, and `'NR'` (not rated), use `labels` with this cell array `{'AAA', 'AA', 'A','BBB','BB','B','CCC','D','NR'}`.

Estimate Point-in-Time and Through-the-Cycle Probabilities

Transition probability estimates are sensitive to the length of the estimation window. When the estimation window is small, the estimates only capture recent credit events, and these can change significantly from one year to the next. These are called point-in-time (PIT) estimates. In contrast, a large time window yields fairly stable estimates that average transition rates over a longer period of time. These are called through-the-cycle (TTC) estimates.

The estimation of PIT probabilities requires repeated calls to `transprob` with a rolling estimation window. Use `transprobprep` every time repeated calls to `transprob` are required. `transprobprep` performs a preprocessing step on the raw dataset that is independent of the estimation window. The benefits of `transprobprep` are greater as the number of repeated calls to `transprob` increases. Also, the performance gains from `transprobprep` are more significant for the `cohort` algorithm.

```
load Data_TransProb
prepData = transprobprep(data);

Years = 1991:2000;
nYears = length(Years);
nRatings = length(prepData.ratingsLabels);
transMatPIT = zeros(nRatings,nRatings,nYears);
algorithm = 'duration';
sampleTotals(nYears,1) = struct('totalsVec',[],'totalsMat',[],...
'algorithm',algorithm);
for t = 1:nYears
   startDate = ['31-Dec-' num2str(Years(t)-1)];
   endDate = ['31-Dec-' num2str(Years(t))];
   [transMatPIT(:,:,t),sampleTotals(t)] = transprob(prepData,...
    'startDate',startDate,'endDate',endDate,'algorithm',algorithm);
end
```

Here is the PIT transition matrix for 1993. Recall that the sample dataset contains simulated credit migrations so the PIT estimates in this example do not match actual historical transition rates.

```
transMatPIT(:,:,Years==1993)

ans =

   95.3193    4.5999    0.0802    0.0004    0.0002    0.0000    0.0000    0.0000
    2.0631   94.5931    3.3057    0.0254    0.0126    0.0002    0.0000    0.0000
    0.0237    2.1748   95.5901    1.4700    0.7284    0.0131    0.0000    0.0000
    0.0003    0.0372    3.2585   95.2914    1.3876    0.0250    0.0001    0.0000
    0.0000    0.0005    0.0657    3.8292   92.7474    3.3459    0.0111    0.0001
    0.0000    0.0001    0.0128    0.7977    8.0926   90.4897    0.5958    0.0113
    0.0000    0.0000    0.0005    0.0459    0.5026   11.1621   84.9315    3.3574
         0         0         0         0         0         0         0  100.0000
```

A structure array stores the `sampleTotals` optional output from `transprob`. The `sampleTotals` structure contains summary information on the total time spent on each rating, and the number of transitions out of each rating, for each year under consideration. For more information on the `sampleTotals` structure, see `transprob`.

As an example, the `sampleTotals` structure for 1993 is used here. The total time spent on each rating is stored in the `totalsVec` field of the structure. The total transitions out of each rating are stored in the `totalsMat` field. A third field, `algorithm`, indicates the algorithm used to generate the structure.

```
sampleTotals(Years==1993).totalsVec
sampleTotals(Years==1993).totalsMat
sampleTotals(Years==1993).algorithm

ans =

  144.4411  230.0356  262.2438  204.9671  246.1315  147.0767   54.9562  215.1479


ans =

     0     7     0     0     0     0     0     0
     5     0     8     0     0     0     0     0
     0     6     0     4     2     0     0     0
     0     0     7     0     3     0     0     0
     0     0     0    10     0     9     0     0
     0     0     0     1    13     0     1     0
     0     0     0     0     0     7     0     2
     0     0     0     0     0     0     0     0


ans =

duration
```

To get the TTC transition matrix, pass the `sampleTotals` structure array to `transprobbytotals`. Internally, `transprobbytotals` aggregates the information in the `sampleTotals` structures to get the total time spent on each rating over the 10 years considered in this example, and the total number of transitions out of each rating during the same period. `transprobbytotals` uses the aggregated information to get the TTC matrix, or average 1-year transition matrix.

```
transMatTTC = transprobbytotals(sampleTotals)

transMatTTC =

   92.8544    6.1068    0.7463    0.2761    0.0123    0.0009    0.0001    0.0032
    2.9399   92.2329    3.8394    0.7349    0.1676    0.0050    0.0004    0.0799
    0.2410    4.5963   90.3468    3.9572    0.6909    0.0521    0.0025    0.1133
    0.0530    0.4729    7.9221   87.2751    3.5075    0.4650    0.0791    0.2254
    0.0460    0.1636    1.1873    9.3442   85.4305    2.9520    0.1150    0.7615
    0.0031    0.0152    0.2608    1.5563   10.4468   83.8525    1.9771    1.8882
    0.0009    0.0041    0.0542    0.8378    2.9996    7.3614   82.4758    6.2662
         0         0         0         0         0         0         0  100.0000
```

The same TTC matrix could be obtained with a direct call to `transprob`, setting the estimation window to the 10 years under consideration. But it is much more efficient to use the `sampleTotals` structures, whenever they are available. (Note, for the `duration` algorithm, these alternative workflows can result in small numerical differences in the estimates whenever leap years are part of the sample.)

In “Estimate Transition Probabilities” on page 8-2, a 1-year transition matrix is estimated using the 5-year time window from 1996 through 2000. This is another

example of a TTC matrix and this can also be computed using the `sampleTotals` structure array.

```
transprobbytotals(sampleTotals(Years>=1996&Years<=2000))

ans =

   90.6239    7.9048    1.0313    0.4123    0.0210    0.0020    0.0003    0.0043
    4.4776   89.5565    4.5294    1.1224    0.2283    0.0094    0.0009    0.0754
    0.3982    6.1159   87.0651    5.4797    0.7636    0.0892    0.0050    0.0832
    0.1029    0.8571   10.7909   83.0218    3.9968    0.7001    0.1313    0.3991
    0.1043    0.3744    2.2960   14.0947   78.9851    3.0012    0.0463    1.0980
    0.0113    0.0544    0.7054    3.2922   15.4341   75.6004    1.8165    3.0858
    0.0044    0.0189    0.1903    1.9742    6.2318   10.2332   75.9990    5.3482
         0         0         0         0         0         0         0  100.0000
```

Estimate t-Year Default Probabilities

By varying the start and end dates, the amount of data considered for the estimation is changed, but the output still contains, by default, 1-year transition probabilities. You can change the default behavior by specifying the `transInterval` argument, as illustrated in “Estimate Transition Probabilities” on page 8-2.

However, when *t*-year transition probabilities are required for a whole range of values of *t*, for example, 1-year, 2-year, 3-year, 4-year, and 5-year transition probabilities, it is more efficient to call `transprob` once to get the optional output `sampleTotals`. You can use the same `sampleTotals` structure can be used to get the *t*-year transition matrix for any transition interval *t*. Given a `sampleTotals` structure and a transition interval, you can get the corresponding transition matrix by using `transprobbytotals`.

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';

[~,sampleTotals] = transprob(data,'startDate', ...
startDate, 'endDate',endDate);

DefProb = zeros(7,5);
for t = 1:5
   transMatTemp = transprobbytotals(sampleTotals,'transInterval',t);
   DefProb(:,t) = transMatTemp(1:7,8);
end
DefProb

DefProb =

    0.0043    0.0169    0.0377    0.0666    0.1033
    0.0754    0.1542    0.2377    0.3265    0.4213
    0.0832    0.1936    0.3276    0.4819    0.6536
```

```
    0.3992    0.8127    1.2336    1.6566    2.0779
    1.0980    2.1189    3.0668    3.9468    4.7644
    3.0860    5.6994    7.9281    9.8418   11.4963
    5.3484    9.8053   13.5320   16.6599   19.2964
```

Estimate Bootstrap Confidence Intervals

`transprob` also returns the `idTotals` structure array which contains, for each ID, or company, the total time spent on each rating, and the total transitions out of each rating. For more information on the `idTotals` structure, see `transprob`. The `idTotals` structure is similar to the `sampleTotals` structures (see "Estimate Point-in-Time and Through-the-Cycle Probabilities" on page 8-6), but `idTotals` has the information at an ID level. Because most companies only migrate between very few ratings, the numeric arrays in `idTotals` are stored as sparse arrays to reduce memory requirements.

You can use the `idTotals` structure array to estimate confidence intervals for the transition probabilities using a bootstrapping procedure, as the following example demonstrates. To do this, call `transprob` and keep the third output argument, `idTotals`. The `idTotals` fields are displayed for the last company in the sample. Within the estimation window, this company spends almost a year as `'AA'` and it is then upgraded to `'AAA'`.

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';

[transMat,~,idTotals] = transprob(data,...
   'startDate',startDate,'endDate',endDate);

% Total time spent on each rating
full(idTotals(end).totalsVec)
% Total transitions out of each rating
full(idTotals(end).totalsMat)
% Algorithm
idTotals(end).algorithm

ans =

    4.0820    0.9180         0         0         0         0         0         0


ans =

     0     0     0     0     0     0     0     0
     1     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

```
ans =

duration
```

Next, use `bootstrp` from Statistics and Machine Learning Toolbox with `transprobbytotals` as the bootstrap function and `idTotals` as the data to sample from. Each bootstrap sample corresponds to a dataset made of companies sampled with replacement from the original data. However, you do not have to draw companies from the original data, because a bootstrap `idTotals` sample contains all the information required to compute the transition probabilities. `transprobbytotals` aggregates all structures in each bootstrap `idTotals` sample and finds the corresponding transition matrix.

To estimate 95% confidence intervals for the transition matrix and display the probabilities of default together with its upper and lower confidence bounds:

```
PD = transMat(1:7,8);

bootstat = bootstrp(100,@(totals)transprobbytotals(totals),idTotals);
ci = prctile(bootstat,[2.5 97.5]); % 95% confidence
CIlower = reshape(ci(1,:),8,8);
CIupper = reshape(ci(2,:),8,8);
PD_LB = CIlower(1:7,8);
PD_UB = CIupper(1:7,8);

[PD_LB PD PD_UB]

ans =

    0.0004    0.0043    0.0106
    0.0028    0.0754    0.2192
    0.0126    0.0832    0.2180
    0.1659    0.3992    0.6617
    0.5703    1.0980    1.7260
    1.7264    3.0860    4.7602
    1.7678    5.3484    9.5055
```

Group Credit Ratings

Credit rating scales can be more or less granular. For example, there are ratings with qualifiers (such as, `'AA+'`, `'BB-'`, etc.), whole ratings (`'AA'`, `'BB'`, etc.), and investment or speculative grade (`'IG'`, `'SG'`) categories. Given a dataset with credit ratings at a more granular level, transition probabilities for less granular categories can be of interest. For example, you might be interested in a transition matrix for investment and

speculative grades given a dataset with whole ratings. Use `transprobgrouptotals` for this evaluation, as illustrated in the following examples. The sample dataset data has whole credit ratings:

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
data(1:5,:)

ans =

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '12-May-1986'    'B'
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'
    '00013326'    '09-Feb-1985'    'A'
```

A call to `transprob` returns the transition matrix and totals structures for the eight (`'AAA'` to `'D'`) whole credit ratings. The array with number of transitions out of each credit rating is displayed after the call to `transprob`:

```
[transMat,sampleTotals,idTotals] = transprob(data,'startDate',startDate,...
'endDate',endDate);
sampleTotals.totalsMat

ans =

     0    67     7     3     0     0     0     0
    67     0    68    15     3     0     0     1
     4   101     0    93    11     1     0     1
     1     7   163     0    62    10     2     5
     1     3    16   168     0    37     0    11
     0     0     2    10    83     0    10    14
     0     0     0     2     8    16     0     7
     0     0     0     0     0     0     0     0
```

Next, use `transprobgrouptotals` to group whole ratings into investment and speculative grades. This function takes a totals structure as the first argument. The second argument indicates the edges between rating categories. In this case, ratings 1 through 4 (`'AAA'` through `'BBB'`) correspond to the first category (`'IG'`), ratings 5 through 7 (`'BB'` through `'CCC'`) to the second category (`'SG'`), and rating 8 (`'D'`) is a category of its own. `transprobgrouptotals` adds up the total time spent on ratings that belong to the same category. For example, total times spent on `'AAA'` through `'BBB'` are added up as the total time spent on `'IG'`. `transprobgrouptotals` also adds

up the total number of transitions between any 'IG' rating and any 'SG' rating, for example, a credit migration from 'BBB' to 'BB'.

The grouped totals can then be passed to transprobbytotals to obtain the transition matrix for investment and speculative grades. Both totalsMat and the new transition matrix are both 3-by-3, corresponding to the grouped categories 'IG', 'SG', and 'D'.

```
sampleTotalsIGSG = transprobgrouptotals(sampleTotals,[4 7 8])
transMatIGSG = transprobbytotals(sampleTotalsIGSG)

sampleTotalsIGSG =

    totalsVec: [4.8591e+003 1.5034e+003 1.1621e+003]
    totalsMat: [3x3 double]
    algorithm: 'duration'

transMatIGSG =

   98.1591    1.6798    0.1611
   12.3228   85.6961    1.9811
         0         0  100.0000
```

When a totals structure array is passed to transprobgrouptotals, this function groups each structure in the array individually and preserves sparsity, if the fields in the input structures are sparse. One way to exploit this feature is to compute confidence intervals for the investment grade default rate and the speculative grade default rate (see also “Estimate Bootstrap Confidence Intervals” on page 8-10).

```
PDIGSG = transMatIGSG(1:2,3);

idTotalsIGSG = transprobgrouptotals(idTotals,[4 7 8]);
bootstat = bootstrp(100,@(totals)transprobbytotals(totals),idTotalsIGSG);
ci = prctile(bootstat,[2.5 97.5]); % 95% confidence
CIlower = reshape(ci(1,:),3,3);
CIupper = reshape(ci(2,:),3,3);
PDIGSG_LB = CIlower(1:2,3);
PDIGSG_UB = CIupper(1:2,3);

[PDIGSG_LB PDIGSG PDIGSG_UB]

ans =

    0.0603    0.1611    0.2538
    1.3470    1.9811    2.6195
```

Work with Nonsquare Matrices

Transition probabilities and the number of transitions between ratings are usually reported without the `'D'` (`'Default'`) row. For example, a credit report can contain the following table, indicating the number of issuers starting in each rating (first column), and the number of transitions between ratings (remaining columns):

	Initial	AAA	AA	A	BBB	BB	B	CCC	D
AAA	98	88	9	1	0	0	0	0	0
AA	389	0	368	19	2	0	0	0	0
A	1165	1	21	1087	56	0	0	0	0
BBB	1435	0	2	89	1289	45	8	0	2
BB	915	0	0	1	60	776	73	2	3
B	867	0	0	1	7	88	715	39	17
CCC	112	0	0	0	1	3	34	61	13

You can store the information in this table in a totals structure compatible with the `cohort` algorithm. For more information on the `cohort` algorithm and the totals structure, see `transprob`. The `totalsMat` field is a nonsquare array in this case.

```
% Define totals structure
totals.totalsVec = [98 389 1165 1435 915 867 112];
totals.totalsMat = [
   88    9    1    0    0    0    0    0;
    0  368   19    2    0    0    0    0;
    1   21 1087   56    0    0    0    0;
    0    2   89 1289   45    8    0    2;
    0    0    1   60  776   73    2    3;
    0    0    1    7   88  715   39   17;
    0    0    0    1    3   34   61   13];
totals.algorithm = 'cohort';
```

`transprobbytotals` and `transprobgrouptotals` accept totals inputs with nonsquare `totalsMat` fields. To get the transition matrix corresponding to the previous table, and to group ratings into investment and speculative grade with the corresponding matrix:

```
transMat = transprobbytotals(totals)

% Group into IG/SG and get IG/SG transition matrix
totalsIGSG = transprobgrouptotals(totals,[4 7]);
transMatIGSG = transprobbytotals(totalsIGSG)

transMat =

   89.7959    9.1837    1.0204         0         0         0         0         0
         0   94.6015    4.8843    0.5141         0         0         0         0
    0.0858    1.8026   93.3047    4.8069         0         0         0         0
```

```
         0    0.1394    6.2021   89.8258    3.1359    0.5575         0    0.1394
         0         0    0.1093    6.5574   84.8087    7.9781    0.2186    0.3279
         0         0    0.1153    0.8074   10.1499   82.4683    4.4983    1.9608
         0         0         0    0.8929    2.6786   30.3571   54.4643   11.6071


transMatIGSG =

   98.2183    1.7169    0.0648
    3.6959   94.5618    1.7423
```

Remove Outliers

The `idTotals` output from `transprob` can also be exploited to update the transition probability estimates after removing some outlier information. For more information on `idTotals`, see `transprob`. For example, if you know that the credit rating migration information for the 4th and 27th companies in the data have problems, you can remove those companies and efficiently update the transition probabilities as follows:

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
[transMat,~,idTotals] = transprob(data,'startDate', ...
startDate, 'endDate',endDate);
transMat
```

```
transMat =

90.6236    7.9051    1.0314    0.4123    0.0210    0.0020    0.0003    0.0043
 4.4780   89.5558    4.5298    1.1225    0.2284    0.0094    0.0009    0.0754
 0.3983    6.1164   87.0641    5.4801    0.7637    0.0892    0.0050    0.0832
 0.1029    0.8572   10.7918   83.0204    3.9971    0.7001    0.1313    0.3992
 0.1043    0.3745    2.2962   14.0954   78.9840    3.0013    0.0463    1.0980
 0.0113    0.0544    0.7055    3.2925   15.4350   75.5988    1.8166    3.0860
 0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
      0         0         0         0         0         0         0  100.0000

nIDs = length(idTotals);
keepInd = setdiff(1:nIDs,[4 27]);
transMatNoOutlier = transprobbytotals(idTotals(keepInd))

transMatNoOutlier =

90.6241    7.9067    1.0290    0.4124    0.0211    0.0020    0.0003    0.0043
 4.4917   89.5918    4.4779    1.1240    0.2288    0.0094    0.0009    0.0756
 0.3990    6.1220   87.0530    5.4841    0.7643    0.0893    0.0050    0.0833
 0.1030    0.8576   10.7909   83.0207    3.9971    0.7001    0.1313    0.3992
 0.1043    0.3746    2.2960   14.0955   78.9840    3.0013    0.0463    1.0980
 0.0113    0.0544    0.7054    3.2925   15.4350   75.5988    1.8166    3.0860
 0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
      0         0         0         0         0         0         0  100.0000
```

Deciding which companies to remove is a case-by-case situation. Reasons to remove a company can include a typo in one of the ratings histories, or an unusual migration between ratings whose impact on the transition probability estimates must be measured.

`transprob` does not reorder the companies in any way. The ordering of companies in the input data is the same as the ordering in the `idTotals` array.

Estimate Probabilities for Different Segments

You can use `idTotals` efficiently to get estimates over different segments of the sample. For more information on `idTotals`, see `transprob`. For example, assume that the companies in the example are grouped into three geographic regions and that the companies were grouped by geographic regions previously, so that the first 340 companies correspond to the first region, the next 572 companies to the second region, and the rest to the third region. You can efficiently get transition probabilities for each region as follows:

```
load Data_TransProb
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
[~,~,idTotals] = transprob(data,'startDate', ...
startDate, 'endDate',endDate);

n1 = 340;
n2 = 572;
transMatG1 = transprobbytotals(idTotals(1:n1))
transMatG2 = transprobbytotals(idTotals(n1+1:n1+n2))
transMatG3 = transprobbytotals(idTotals(n1+n2+1:end))
```

```
transMatG1 =

90.8299    7.6501    0.3178    1.1700    0.0255    0.0044    0.0021    0.0002
 4.3572   89.0262    5.7838    0.8039    0.0245    0.0029    0.0013    0.0001
 0.7066    6.7567   86.6320    5.4950    0.3721    0.0252    0.0101    0.0023
 0.0626    1.3688   10.3895   83.5022    3.6823    0.6466    0.3084    0.0396
 0.0256    0.7884    2.6970   13.7857   78.8321    2.8310    0.0561    0.9842
 0.0026    0.1095    0.4280    3.5204   21.1437   72.9230    1.6456    0.2273
 0.0005    0.0216    0.0730    0.4574    4.9586    4.2821   80.3062    9.9006
      0         0         0         0         0         0         0  100.0000

transMatG2 =

90.5798    8.4877    0.8202    0.0884    0.0132    0.0011    0.0000    0.0096
 4.1999   90.0371    3.8657    1.4744    0.2144    0.0128    0.0001    0.1956
 0.3022    5.9869   86.7128    5.5526    1.0411    0.1902    0.0015    0.2127
 0.0204    0.5606   10.9342   82.9195    4.0123    0.7398    0.0059    0.8073
 0.0089    0.3338    2.1185   16.6496   76.2395    3.1241    0.0261    1.4995
 0.0013    0.0465    0.6710    2.4731   14.7281   76.7378    1.2993    4.0428
 0.0002    0.0080    0.0681    0.4598    4.1324    8.4380   80.9092    5.9843
      0         0         0         0         0         0         0  100.0000

transMatG3 =

90.5655    7.5408    1.5288    0.3369    0.0258    0.0015    0.0003    0.0004
 4.8073   89.3842    4.4865    0.9582    0.3509    0.0095    0.0009    0.0025
 0.3153    5.8771   87.6353    5.4101    0.7160    0.0322    0.0052    0.0088
 0.1995    0.8625   10.8682   82.8717    4.1423    0.6903    0.1565    0.2090
 0.2465    0.1091    2.1558   12.0289   81.5803    3.0057    0.0616    0.8122
```

```
    0.0227    0.0400    0.9380    4.3175   12.3632   75.9429    2.5766    3.7991
    0.0149    0.0180    0.3414    3.6918    8.1414   13.6010   70.7254    3.4661
         0         0         0         0         0         0         0  100.0000
```

Work with Large Datasets

This example shows how to aggregate estimates from two (or more) datasets. It is possible that two datasets, coming from two different databases, must be considered for the estimation of the transition probabilities. Also, if a dataset is too large and cannot be loaded into memory, the dataset can be split into two (or more) datasets. In these cases, it is simple to apply `transprob` to each individual dataset, and then get the final estimates corresponding to the aggregated data with a call to `transprobbytotals` at the end.

For example, the dataset data is artificially split into two sections in this example. In practice the two datasets would come from different files or databases. When aggregating multiple datasets, the history of a company cannot be split across datasets. You can analyze that this condition is satisfied for the arbitrarily chosen cut-off point.

```
load Data_TransProb

cutoff = 2099;
data(cutoff-5:cutoff,:)
data(cutoff+1:cutoff+6,:)
```

```
ans =

    '00011166'    '24-Aug-1995'    'BBB'
    '00011166'    '25-Jan-1997'    'A'
    '00011166'    '01-Feb-1998'    'AA'
    '00014878'    '15-Mar-1983'    'B'
    '00014878'    '21-Sep-1986'    'BB'
    '00014878'    '17-Jan-1998'    'BBB'


ans =

    '00012043'    '09-Feb-1985'    'BBB'
    '00012043'    '03-Jan-1988'    'A'
    '00012043'    '15-Jan-1994'    'AAA'
    '00011157'    '24-Jun-1984'    'A'
    '00011157'    '09-Dec-1999'    'BBB'
    '00011157'    '28-Mar-2001'    'A'
```

When working with multiple datasets, it is important to set the start and end dates explicitly. Otherwise, the estimation window differs for each dataset because the default start and end dates used by `transprob` are the earliest and latest dates found in the input data.

```
startDate = '31-Dec-1995';
endDate = '31-Dec-2000';
```

In practice, this is the point where you can read in the first dataset. Now, the dataset is already obtained. Call `transprob` with the first dataset and the explicit start and end dates. Keep only the `sampleTotals` output. For details on `sampleTotals`, see `transprob`.

```
[~,sampleTotals(1)] = transprob(data(1:cutoff,:),...
   'startDate',startDate,'endDate',endDate);
```

Repeat for the remaining datasets. Note the different `sampleTotals` structures are stored in a structured array.

```
[~,sampleTotals(2)] = transprob(data(cutoff+1:end,:),...
   'startDate',startDate,'endDate',endDate);
```

To get the transition matrix corresponding to the aggregated dataset, use `transprobbytotals`. When the totals input is a structure array, `transprobbytotals` aggregates the information over all structures, and returns a single transition matrix.

```
transMatAggr = transprobbytotals(sampleTotals)
```

```
transMatAggr =

   90.6236    7.9051    1.0314    0.4123    0.0210    0.0020    0.0003    0.0043
    4.4780   89.5558    4.5298    1.1225    0.2284    0.0094    0.0009    0.0754
    0.3983    6.1164   87.0641    5.4801    0.7637    0.0892    0.0050    0.0832
    0.1029    0.8572   10.7918   83.0204    3.9971    0.7001    0.1313    0.3992
    0.1043    0.3745    2.2962   14.0954   78.9840    3.0013    0.0463    1.0980
    0.0113    0.0544    0.7055    3.2925   15.4350   75.5988    1.8166    3.0860
    0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
         0         0         0         0         0         0         0  100.0000
```

As a sanity check, for this example you can analyze that the aggregation procedure yields the same estimates (up to numerical differences) as estimating the probabilities directly over the entire sample:

```
transMatWhole = transprob(data,'startDate',startDate,'endDate',endDate)
aggError = max(max(abs(transMatAggr - transMatWhole)))
```

```
transMatWhole =

   90.6236    7.9051    1.0314    0.4123    0.0210    0.0020    0.0003    0.0043
    4.4780   89.5558    4.5298    1.1225    0.2284    0.0094    0.0009    0.0754
```

```
    0.3983    6.1164   87.0641    5.4801    0.7637    0.0892    0.0050    0.0832
    0.1029    0.8572   10.7918   83.0204    3.9971    0.7001    0.1313    0.3992
    0.1043    0.3745    2.2962   14.0954   78.9840    3.0013    0.0463    1.0980
    0.0113    0.0544    0.7055    3.2925   15.4350   75.5988    1.8166    3.0860
    0.0044    0.0189    0.1903    1.9743    6.2320   10.2334   75.9983    5.3484
         0         0         0         0         0         0         0  100.0000

aggError =

  2.8422e-014
```

See Also

`bootstrp` | `transprob` | `transprobbytotals` | `transprobfromthresholds` | `transprobgrouptotals` | `transprobprep` | `transprobtothresholds`

Related Examples

- “Credit Quality Thresholds” on page 8-52
- “Credit Rating by Bagging Decision Trees” (Statistics and Machine Learning Toolbox)
- “Forecasting Corporate Default Rates” on page 8-20

External Websites

- Credit Risk Modeling with MATLAB (53 min 09 sec)
- Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Forecasting Corporate Default Rates

This example shows how to build a forecasting model for corporate default rates.

Risk parameters are dynamic in nature, and understanding how these parameters change in time is a fundamental task for risk management.

In the first part of this example, we work with historical credit migrations data to construct some time series of interest, and to visualize default rates dynamics. In the second part of this example, we use some of the series constructed in the first part, and some additional data, to fit a forecasting model for corporate default rates, and to show some backtesting and stress testing concepts. A linear regression model for corporate default rates is presented, but the tools and concepts described can be used with other forecasting methodologies. The appendix at the end references the handling of models for full transition matrices.

People interested in forecasting, backtesting, and stress testing can go directly to the second part of this example. The first part of this example is more relevant for people who work with credit migration data.

Part I: Working with Credit Migrations Data

We work with historical transition probabilities for corporate issuers (variable `TransMat`). This is yearly data for the period 1981-2005, from [10]. The data includes, for each year, the number of issuers per rating at the beginning of the year (variable `nIssuers`), and the number of new issuers per rating per year (variable `nNewIssuers`). There is also a corporate profits forecast, from [9], and a corporate spread, from [4] (variables `CPF` and `SPR`). A variable indicating recession years (`Recession`), consistent with recession dates from [7], is used mainly for visualizations.

```
Example_LoadData
```

Getting Default Rates for Different Ratings Categories

We start by performing some aggregations to get corporate default rates for Investment Grade (IG) and Speculative Grade (SG) issuers, and the overall corporate default rate.

Aggregation and segmentation are relative terms. IG is an aggregate with respect to credit ratings, but a segment from the perspective of the overall corporate portfolio. Other segments are of interest in practice, for example, economic sectors, industries, or geographic regions. The data we use, however, is aggregated by credit ratings, so further

segmentation is not possible. Nonetheless, the tools and workflow discussed here can be useful to work with other segment-specific models.

We use existing functionality in Financial Toolbox™, specifically, functions `transprobgrouptotals` and `transprobbytotals`, to perform the aggregation. These functions take as inputs structures with credit migration information in a particular format. We set up the inputs here, and visualize them below to understand their information and format.

```
% Pre-allocate the struct array
totalsByRtg(nYears,1) = struct('totalsVec',[],'totalsMat',[],...
   'algorithm','cohort');
for t = 1:nYears
   % Number of issuers per rating at the beginning of the year
   totalsByRtg(t).totalsVec = nIssuers(t,:);
   % Number of transitions between ratings during the year
   totalsByRtg(t).totalsMat = round(diag(nIssuers(t,:))*...
      (0.01*TransMat(:,:,t)));
   % Algorithm
   totalsByRtg(t).algorithm = 'cohort';
end
```

It is useful to see both the original data and the data stored in these totals structures side to side. The original data contains number of issuers and transition probabilities for each year. For example, for 2005:

```
fprintf('\nTransition matrix for 2005:\n\n')
```

```
Transition matrix for 2005:
```

```
Example_DisplayTransitions(squeeze(TransMat(:,:,end)),nIssuers(end,:),...
   {'AAA','AA','A','BBB','BB','B','CCC'},...
   {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

	Init	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	98	88.78	9.18	1.02	0	0	0	0	0	1.02
AA	407	0	90.66	4.91	0.49	0	0	0	0	3.93
A	1224	0.08	1.63	88.89	4.41	0	0	0	0	4.98
BBB	1535	0	0.2	5.93	84.04	3.06	0.46	0	0.07	6.25
BB	1015	0	0	0	5.71	76.75	6.9	0.2	0.2	10.25
B	1010	0	0	0.1	0.59	8.51	70.59	3.76	1.58	14.85
CCC	126	0	0	0	0.79	0.79	25.4	46.83	8.73	17.46

The totals structure stores the total number of issuers per rating at the beginning of the year in the `totalsVec` field, and the total *number of migrations* between ratings

(instead of transition probabilities) in the `totalsMat` field. Here is the information for 2005:

```
fprintf('\nTransition counts (totals struct) for 2005:\n\n')
```

```
Transition counts (totals struct) for 2005:
```

```
Example_DisplayTransitions(totalsByRtg(end).totalsMat,...
   totalsByRtg(end).totalsVec,...
   {'AAA','AA','A','BBB','BB','B','CCC'},...
   {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

	Init	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	98	87	9	1	0	0	0	0	0	1
AA	407	0	369	20	2	0	0	0	0	16
A	1224	1	20	1088	54	0	0	0	0	61
BBB	1535	0	3	91	1290	47	7	0	1	96
BB	1015	0	0	0	58	779	70	2	2	104
B	1010	0	0	1	6	86	713	38	16	150
CCC	126	0	0	0	1	1	32	59	11	22

The third field in the totals structure, `algorithm`, indicates that we are working with the `cohort` method (`duration` is also supported, although the information in `totalsVec` and `totalsMat` would be different). These structures are obtained as optional outputs from `transprob`, but this example shows how you can define these structures directly.

We now group ratings `'AAA'` to `'BBB'` (ratings 1 to 4) into the IG category and ratings `'BB'` to `'CCC'` (ratings 5 to 7) into the SG category. We use `transprobgrouptotals` for this. The `edges` argument tells the function which ratings are to be grouped together (1 to 4, and 5 to 7). We also group all non-default ratings into one category. These are preliminary steps to get the IG, SG, and overall default rates for each year.

```
edgesIGSG = [4 7];
totalsIGSG = transprobgrouptotals(totalsByRtg,edgesIGSG);
edgesAll = 7; % could also use edgesAll = 2 with totalsIGSG
totalsAll = transprobgrouptotals(totalsByRtg,edgesAll);
```

Here are the 2005 totals grouped at IG/SG level, and the corresponding transition matrix, recovered using `transprobbytotals`.

```
fprintf('\nTransition counts for 2005 at IG/SG level:\n\n')
```

```
Transition counts for 2005 at IG/SG level:
```

```
Example_DisplayTransitions(totalsIGSG(end).totalsMat,...
   totalsIGSG(end).totalsVec,...
   {'IG','SG'},...
   {'IG','SG','D','NR'})
```

```
        Init     IG     SG      D     NR
 IG     3264   3035     54      1    174
 SG     2151     66   1780     29    276
```

```
fprintf('\nTransition matrix for 2005 at IG/SG level:\n\n')
```

```
Transition matrix for 2005 at IG/SG level:
```

```
Example_DisplayTransitions(transprobbytotals(totalsIGSG(end)),[],...
   {'IG','SG'},...
   {'IG','SG','D','NR'})
```

```
           IG     SG      D     NR
 IG     92.98   1.65   0.03   5.33
 SG      3.07  82.75   1.35  12.83
```

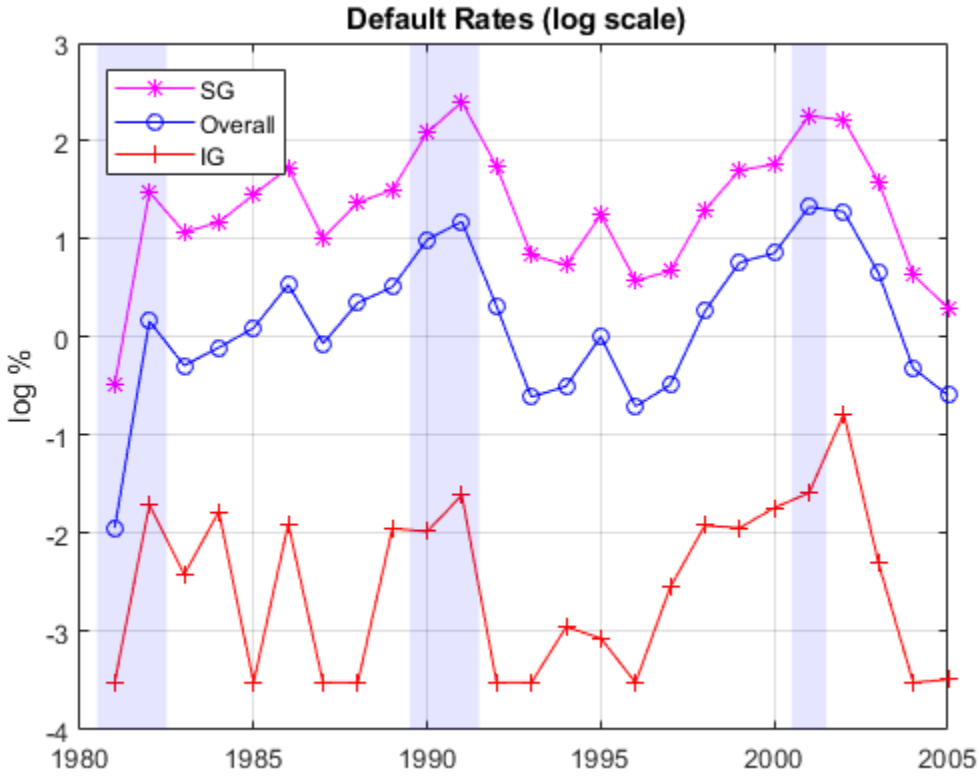
We now get transition matrices for every year both at IG/SG and non-default/default levels and store the default rates only (we do not use the rest of the transition probabilities).

```
DefRateIG = zeros(nYears,1);
DefRateSG = zeros(nYears,1);
DefRate = zeros(nYears,1);
for t=1:nYears
   % Get transition matrix at IG/SG level and extract IG default rate and
   % SG default rate for year t
   tmIGSG = transprobbytotals(totalsIGSG(t));
   DefRateIG(t) = tmIGSG(1,3);
   DefRateSG(t) = tmIGSG(2,3);
   % Get transition matrix at most aggregate level and extract overall
   % corporate default rate for year t
   tmAll = transprobbytotals(totalsAll(t));
   DefRate(t) = tmAll(1,2);
end
```

Here is a visualization of the dynamics of IG, SG, and overall corporate default rates together. To emphasize their patterns, rather than their magnitudes, a log scale is used. The shaded bands indicate recession years. The patterns of SG and IG are slightly different. For example, the IG rate is higher in 1994 than in 1995, but the opposite is

true for SG. More noticeably, the IG default rate peaked after the 2001 recession, in 2002, whereas the peak for SG is in 2001. This suggests that models for the dynamics of the IG and SG default rates could have important differences, a common situation when working with different segments. The overall corporate default rate is by construction a combination of the other two, and its pattern is closer to SG, most likely due to the relative magnitude of SG versus IG.

```
minIG = min(DefRateIG(DefRateIG~=0));
figure
plot(Years,log(DefRateSG),'m-*')
hold on
plot(Years,log(DefRate),'b-o')
plot(Years,log(max(DefRateIG,minIG-0.001)),'r-+')
Example_RecessionBands
hold off
grid on
title('{\bf Default Rates (log scale)}')
ylabel('log %')
legend({'SG','Overall','IG'},'location','NW')
```



Getting Default Rates for Different Time Periods

The default rates obtained are examples of point-in-time (PIT) rates, only the most recent information is used to estimate them. On the other extreme, we can use all the migrations observed in the 25 years spanned by the dataset to estimate long-term, or through-the-cycle (TTC) default rates. Other rates of interest are the average default rates over recession or expansion years.

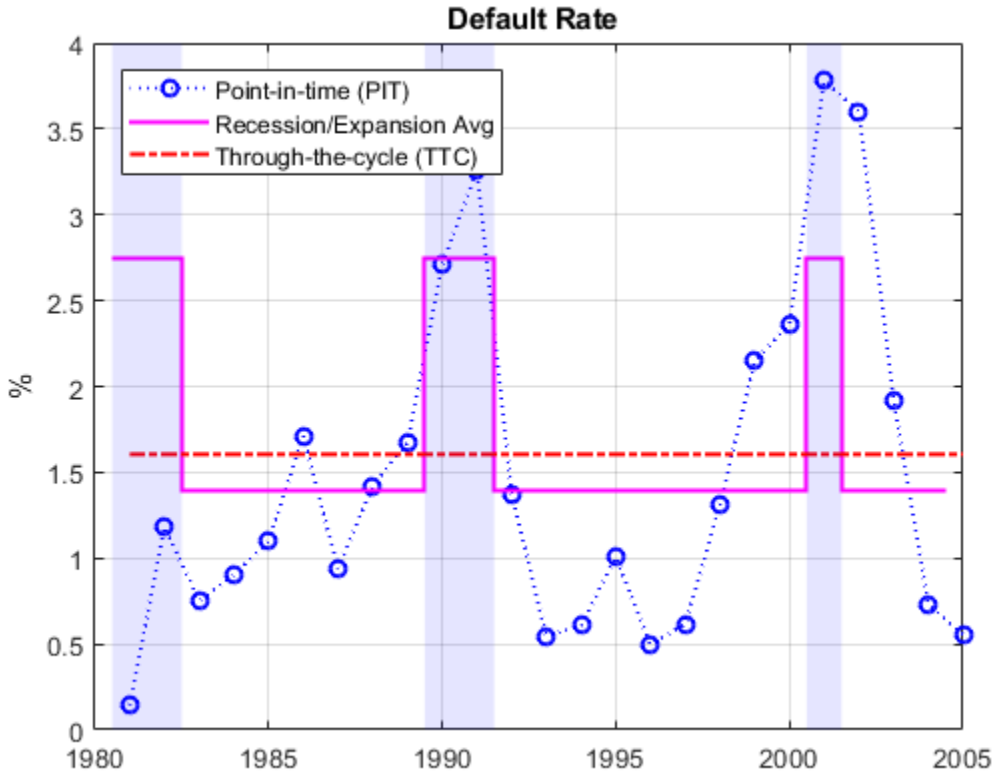
All of these are easy to estimate with the data we have and the same tools. For example, to estimate the average transition probabilities over recession years, pass to `transprobbytotals` the totals structures corresponding to the recession years only. We use logical indexing below, taking advantage of the `Recession` variable.

`transprobbytotals` aggregates the information over time and returns the corresponding transition matrix.

```
tmAllRec = transprobbytotals(totalsAll(Recession));
DefRateRec = tmAllRec(1,2);

tmAllExp = transprobbytotals(totalsAll(~Recession));
DefRateExp = tmAllExp(1,2);

tmAllTTC = transprobbytotals(totalsAll);
DefRateTTC = tmAllTTC(1,2);
```

The following figure shows the estimated PIT rates, TTC rates, and recession and expansion rates.

```
DefRateTwoValues = DefRateExp*ones(nYears,1);
DefRateTwoValues(Recession) = DefRateRec;

figure
plot(Years,DefRate,'bo:','LineWidth',1.2)
hold on
stairs(Years-0.5,DefRateTwoValues,'m-','LineWidth',1.5)
plot(Years,DefRateTTC*ones(nYears,1),'r-.','LineWidth',1.5)
Example_RecessionBands
hold off
grid on
title('{\bf Default Rate}')
ylabel('%')
legend({'Point-in-time (PIT)','Recession/Expansion Avg',...
   'Through-the-cycle (TTC)'},'location','NW')
```



Some analyses (see, for example, [11]) use simulations where the default rate is conditional on the general state of the economy, for example, recession v. expansion. The recession and expansion estimates obtained can be useful in such a framework. These are all historical averages, however, and may not work well if used as predictions for the actual default rates expected on any particular year. In the second part of this example, we revisit the use of these types of historical averages as forecasting tools in a backtesting exercise.

Building Predictors Using Credit Ratings Data

Using the credit data, you can build new time series of interest. We start with an age proxy that is used as predictor in the forecasting model in the second part of this example.

Age is known to be an important factor in predicting default rates; see, e.g., [1] and [5]. Age here means the number of years since a bond was issued. By extension, the age of a portfolio is the average age of its bonds. Certain patterns have been observed historically. Many low-quality borrowers default just a few years after issuing a bond. When troubled companies issue bonds, the amount borrowed helps them make payments for a year or two. Beyond that point, their only source of money is their cash flows, and if they are insufficient, default occurs.

We cannot calculate the exact age of the portfolio, because there is no information at issuer level in the dataset. We follow [6], however, and use the number of new issuers in year t-3 divided by the total number of issuers at the end of year t as an age proxy. Because of the lag, the age proxy starts in 1984. For the numerator, we have explicit information on the number of new issuers. For the denominator, the number of issuers at the end of a year equals the number of issuers at the beginning of next year. This is known for all years but the last one, which is set to the total transitions into a non-default rating plus the number of new issuers on that year.

```
% Total number of issuers at the end of the year
nEOY = zeros(nYears,1);
% nIssuers is number of issuers per ratings at the beginning of the year
% nEOY ( 1981 ) = sum nIssuers ( 1982 ), etc until 2004
nEOY(1:end-1) = sum(nIssuers(2:end,:),2);
% nEOY ( 2005 ) = issuers in non-default state at end of 2005 plus
% new issuers in 2005
nEOY(end) = totalsAll(end).totalsMat(1,1) + sum(nNewIssuers(end,:));
% Age proxy
AGE = 100*[nan(3,1); sum(nNewIssuers(1:end-3,:),2)./nEOY(4:end)];
```

Examples of other time series of interest are the proportion of SG issuers at the end of each year, or an age proxy for SG.

```
% nSGEOY: Number of SG issuers at the end of the year
% nSGEOY is similar to nEOY, but for SG only, from 5 ('BB') to 7 ('CCC')
indSG = 5:7;
nSGEOY = zeros(nYears,1);
nSGEOY(1:end-1) = sum(nIssuers(2:end,indSG),2);
nSGEOY(end) = sum(totalsIGSG(end).totalsMat(:,2)) +...
   sum(nNewIssuers(end,indSG));
% Proportion of SG issuers
SG = 100*nSGEOY./nEOY;
% SG age proxy: new SG issuers in t-3 / total issuers at the end of year t
AGESG = 100*[nan(3,1); sum(nNewIssuers(1:end-3,indSG),2)./nEOY(4:end)];
```

Part II: A Forecasting Model for Default Rates

We work with the following linear regression model for corporate default rates

$$DefRate = \beta_0 + \beta_{age} AGE + \beta_{cpf} CPF + \beta_{spr} SPR$$

where

- AGE: Age proxy defined above
- CPF: Corporate profits forecast
- SPR: Corporate spread over treasuries

This is the same model as in [6], except the model in [6] is for IG only.

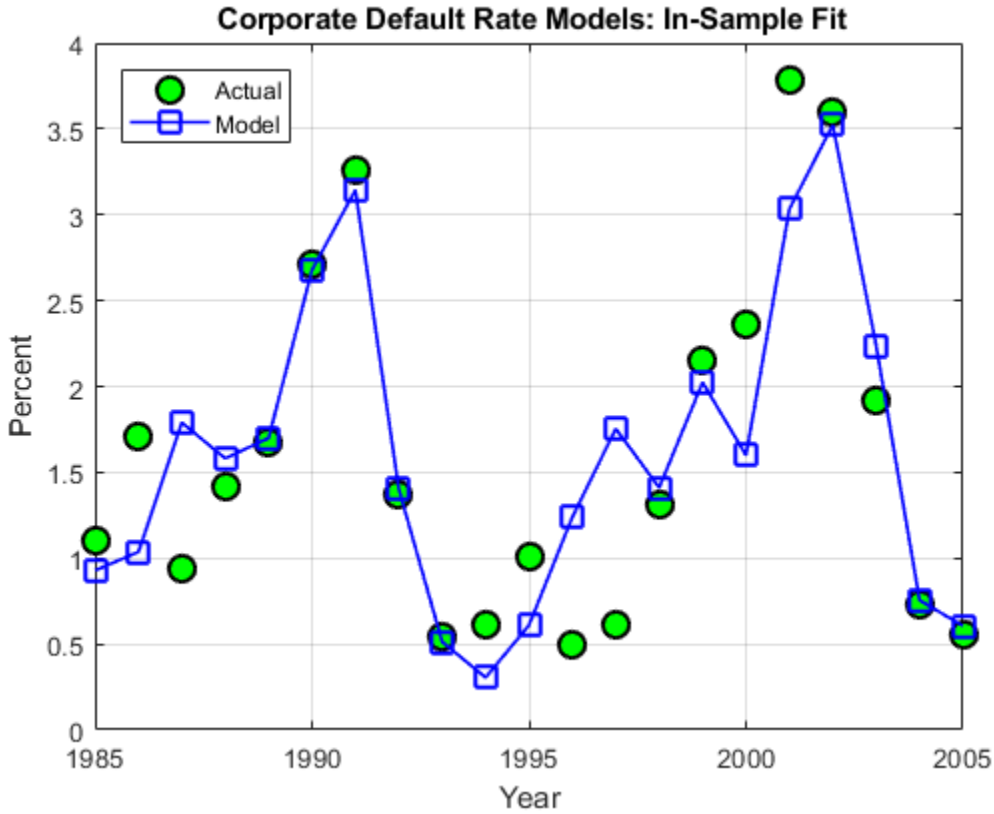
As previously discussed, age is known to be an important factor regarding default rates. The corporate profits provide information on the economic environment. The corporate spread is a proxy for credit quality. Age, environment, and quality are three dimensions frequently found in credit analysis models.

```
inSample = 4:nYears-1;
T = length(inSample);
varNames = {'AGE','CPF','SPR'};
X = [AGE CPF SPR];
X = X(inSample,:);
y = DefRate(inSample+1); % DefaultRate, year t+1
stats = regstats(y,X);

fprintf('\nConst   AGE   CPF   SPR   adjR^2\n')

Const   AGE   CPF   SPR   adjR^2

fprintf('%1.2f  %1.2f %1.2f  %1.2f   %1.4f\n',...
   [stats.beta;stats.adjrsquare])

-1.19  0.15 -0.10  0.71   0.7424
```

The coefficients have the expected sign: default rates tend to increase with a higher proportion of 3-year issuers, decrease with good corporate profits, and increase when the corporate yields are higher. The adjusted R square shows a good fit.

The in-sample fit, or how close the model predictions are from the sample points used to fit the model, is shown in the following figure.

```
bHat = stats.beta;
yHat = [ones(T,1),X]*bHat;

figure
plot(Years(inSample+1),DefRate(inSample+1),'ko','LineWidth',1.5,...
   'MarkerSize',10,'MarkerFaceColor','g')
hold on
plot(Years(inSample+1),yHat,'b-s','LineWidth',1.2,'MarkerSize',10)
hold off
grid on
legend({'Actual','Model'},'location','NW')
title('{\bf Corporate Default Rate Models: In-Sample Fit}')
xlabel('Year')
ylabel('Percent')
```



It can be shown that there is no strong statistical evidence to conclude that the linear regression assumptions are violated. It is apparent that default rates are not normally distributed. The model, however, does not make that assumption. The only normality assumption in the model is that, given the predictors values, the error between the predicted and the observed default rates is normally distributed. By looking at the in-sample fit, this does not seem unreasonable. The magnitude of the errors certainly seems independent of whether the default rates are high or low. Year 2001 has a high default rate and a high error, but years 1991 or 2002 also have high rates and yet very small errors. Likewise, low default rate years like 1996 and 1997 show considerable errors, but years 2004 or 2005 have similarly low rates and tiny errors.

A thorough statistical analysis of the model is out of scope here, but there are several detailed examples in Statistics and Machine Learning Toolbox™ and Econometrics Toolbox™.

Backtesting

To evaluate how this model performs out-of-sample, we set up a backtesting exercise. Starting at the end of 1995, we fit the linear regression model with the information available up to that date, and compare the model prediction to the actual default rate observed the following year. We repeat the same for all subsequent years until the end of the sample.

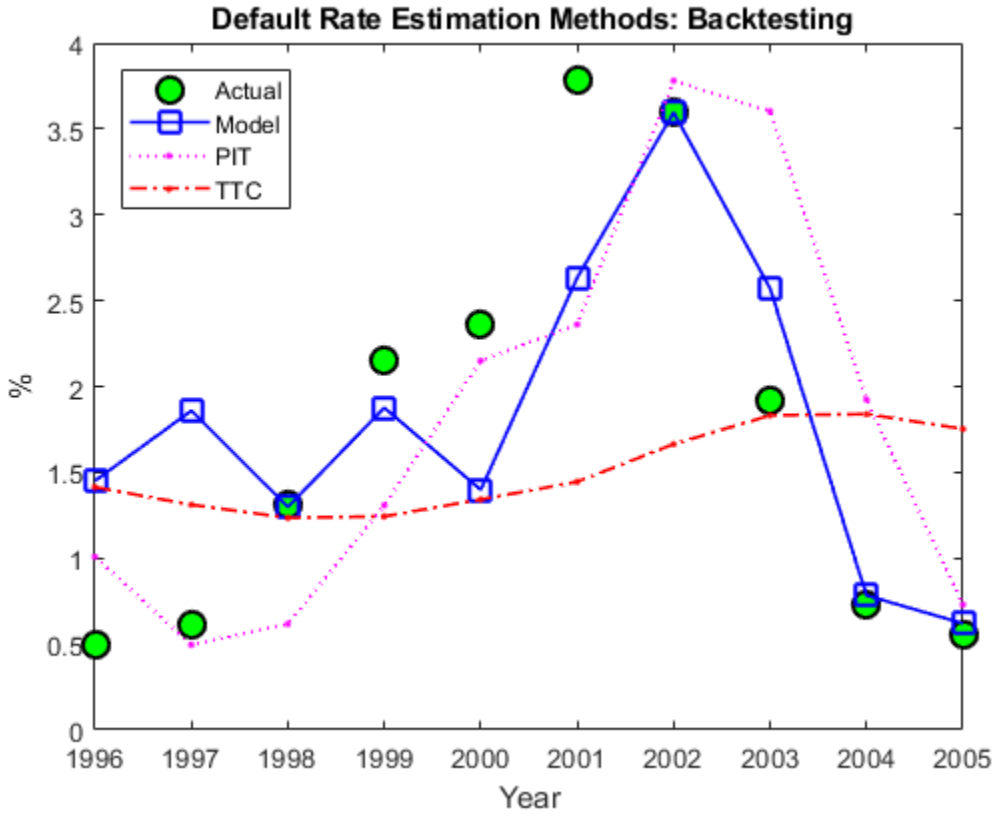
For backtesting, relative performance of a model, when compared to alternatives, is easier to assess than the performance of a model in isolation. Here we include two alternatives to determine next year's default rate, both likely candidates in practice. One is the TTC default rate, estimated with data from the beginning of the sample to the current year, a very stable default rate estimate. The other is the PIT rate, estimated using data from the most recent year only, much more sensitive to recent events.

```
XBT = [AGE,CPF,SPR];
yBT = DefRate;

iYear0 = find(Years==1984); % index of first year in sample, 1984
T = find(Years==1995); % ind "current" year, start at 1995, updated in loop
YearsBT = 1996:2005; % years predicted in BT exercise
iYearsBT = find(Years==1996):find(Years==2005); % corresponding indices
nYearsBT = length(YearsBT); % number of years in BT exercise

MethodTags = {'Model','PIT','TTC'};
nMethods = length(MethodTags);
PredDefRate = zeros(nYearsBT,nMethods);
```

```
ErrorBT = zeros(nYearsBT,nMethods);

alpha = 0.05;
PredDefLoBnd = zeros(nYearsBT,1);
PredDefUpBnd = zeros(nYearsBT,1);

for k=1:nYearsBT
   % In sample years for predictors, from 1984 to "last" year (T-1)
   inSampleBT = iYear0:T-1;

   % Method 1: Linear regression model
   %   Fit regression model with data up to "current" year (T)
   s = regstats(yBT(inSampleBT+1),XBT(inSampleBT,:));
   %   Predict default rate for "next" year (T+1)
   PredDefRate(k,1) = [1 XBT(T,:)]*s.beta;
   %   Compute prediction intervals
   tCrit = tinv(1-alpha/2,s.tstat.dfe);
   PredStd = sqrt([1 XBT(T,:)]*s.covb*[1 XBT(T,:)]'+s.mse);
   PredDefLoBnd(k) = max(0,PredDefRate(k,1) - tCrit*PredStd);
   PredDefUpBnd(k) = PredDefRate(k,1) + tCrit*PredStd;

   % Method 2: Point-in-time (PIT) default rate
   PredDefRate(k,2) = DefRate(T);

   % Method 3: Through-the-cycle (TTC) default rate
   tmAll = transprobbytotals(totalsAll(iYear0:T));
   PredDefRate(k,3) = tmAll(1,2);

   % Update error
   ErrorBT(k,:) = PredDefRate(k,:) - DefRate(T+1);

   % Move to next year
   T = T + 1;
end
```

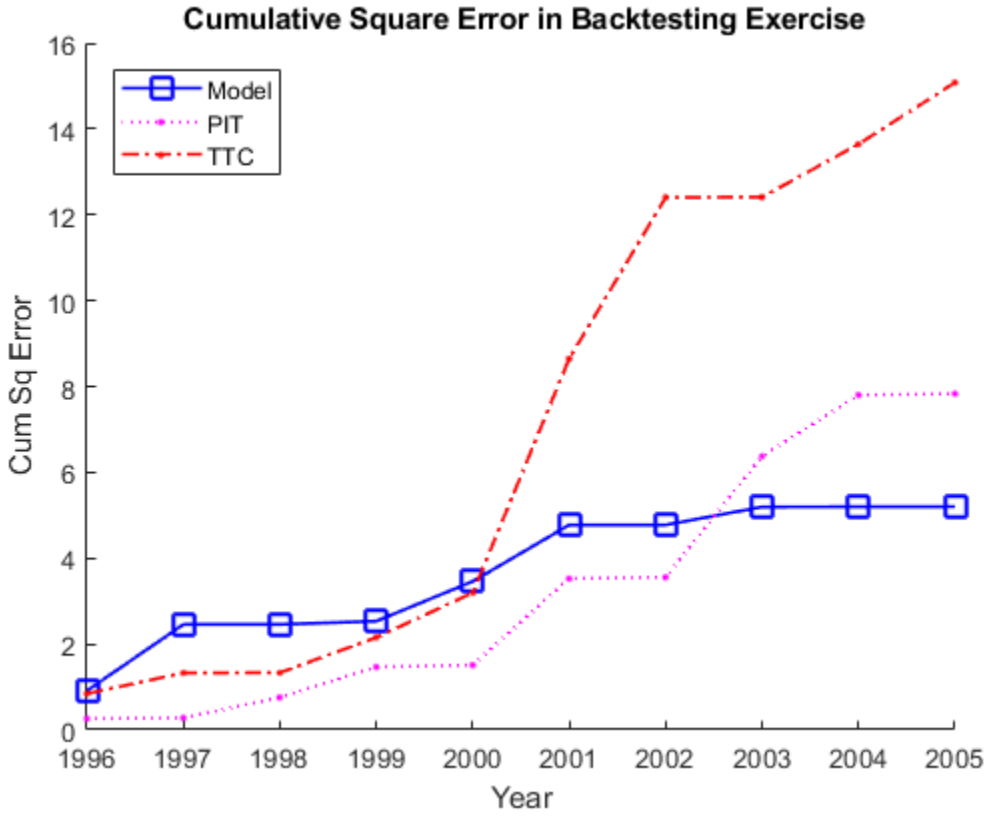
Here are the predictions of the three alternative approaches, compared to the actual default rates observed. Unsurprisingly, TTC shows a very poor predictive power. However, it is not obvious whether PIT or the linear regression model makes better predictions in this 10-year time span.

```
Example_BacktestPlot(YearsBT,DefRate(iYearsBT),PredDefRate,'Year','%',...
   '{\bf Default Rate Estimation Methods: Backtesting}',...
   ['Actual' MethodTags],'NW')
```



The following plot keeps track of cumulative square error, a measure often used for comparisons in backtesting exercises. This confirms TTC as a poor alternative. PIT shows lower cumulative error than the linear regression model in the late nineties, but after the 2001 recession the situation is reversed. Cumulative square error, however, is not an intuitive measure, it is hard to get a sense of what the difference between these alternatives means in practical terms.

```
CumSqError = cumsum(ErrorBT.^2);
Example_BacktestPlot(YearsBT,[],CumSqError,'Year','Cum Sq Error',...
   '{\bf Cumulative Square Error in Backtesting Exercise}',...
   MethodTags,'NW')
```



It makes sense to translate the prediction errors into a monetary measure. Here we measure the impact of the prediction error on a simplified framework for generating loss reserves in an institution.

We assume a homogeneous portfolio, where all credits have the same probability of default, the same loss given default (LGD), and the same exposure at default (EAD). Both LGD and EAD are assumed to be known. For simplicity, we keep these values constant for the 10 years of the exercise. We set LGD at 45%, and EAD per bond at 100 million. The portfolio is assumed to have a thousand bonds, so the total value of the portfolio, the total EAD, is 100 billion.

The predicted default rate for year t, determined at the end of year t-1, is used to calculate the expected loss for year t

$$EL_t = EAD_t \times LGD_t \times PredictedDefaultRate_t$$

This is the amount added to the loss reserves at the start of year *t*. At the end of the year, the actual losses are known

$$AL_t = EAD_t \times LGD_t \times ObservedDefaultRate_t$$

We assume that unused loss reserves remain in the reserves fund. The starting balance in reserves at the beginning of the exercise is set to zero. If the actual losses surpass the expected loss, unused reserves accumulated over the years are used first, and only if these run out, capital is used to cover a shortfall. All this translates into the following formula

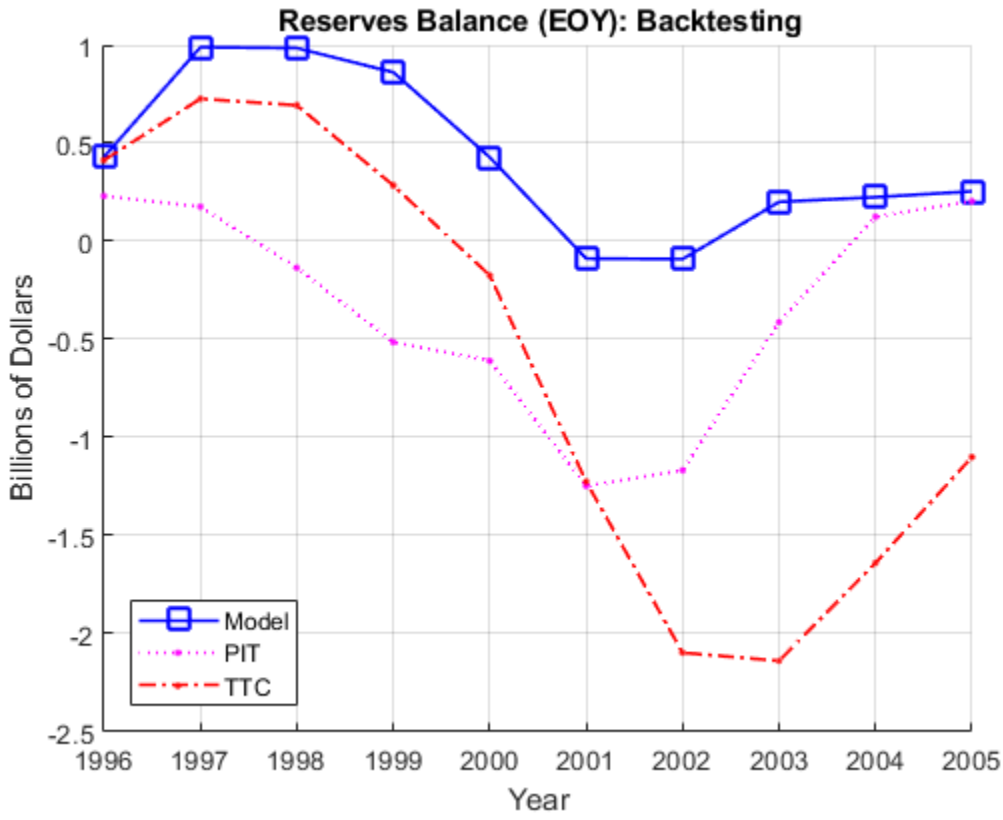
$$Reserves_t = Reserves_{t-1} + (EL_t - AL_t)$$

or equivalently

$$Reserves_t = \sum_{s=1}^{t} (EL_t - AL_t)$$

The following figure shows the loss reserves balance for each of the three alternatives in the backtesting exercise.

```
EAD = 100*ones(nYearsBT,1); % in billions
LGD = 0.45*ones(nYearsBT,1); % Loss given default, 45%
% Reserves excess or shortfall for each year, in billions
ReservesExcessShortfall = bsxfun(@times,EAD.*LGD,ErrorBT/100);
% Cumulative reserve balance for each year, in billions
ReservesBalanceEOY = cumsum(ReservesExcessShortfall);

Example_BacktestPlot(YearsBT,[],ReservesBalanceEOY,'Year',...
   'Billions of Dollars',...
   '{\bf Reserves Balance (EOY): Backtesting}',...
   MethodTags,'SW')
grid on
```



Using the linear regression model we only observe a deficit in reserves in two out of ten years, and the maximum deficit, in 2001, is 0.09 billion, only nine basis points of the portfolio value.

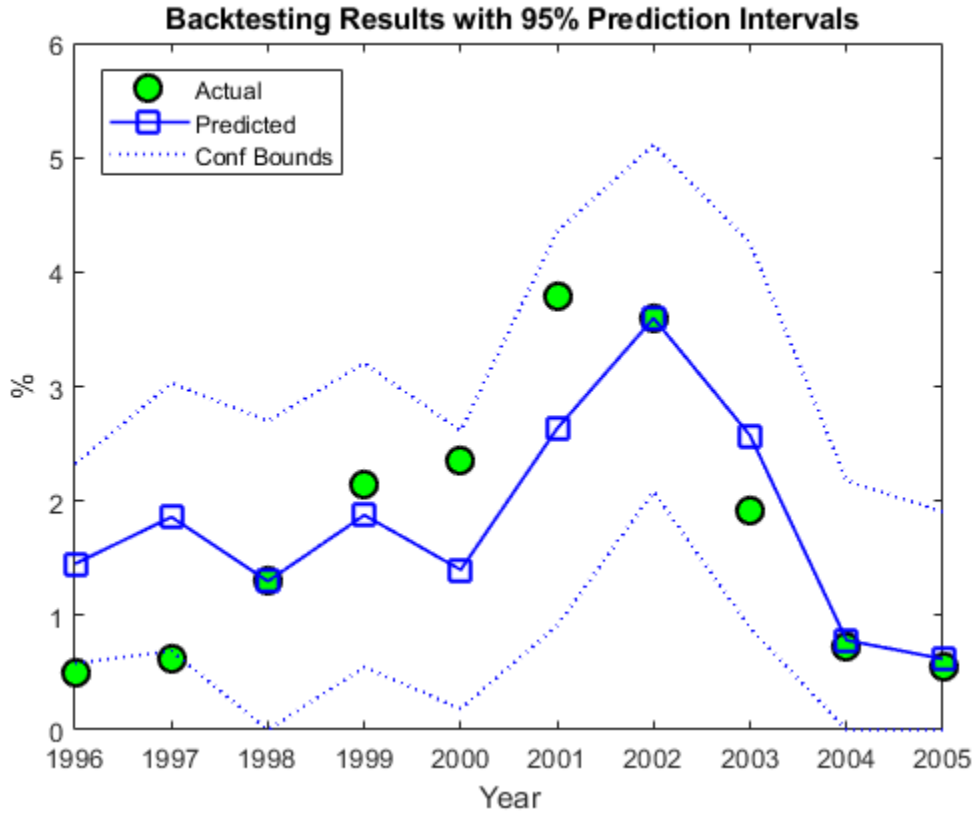
In contrast, both TTC and PIT reach a deficit of 1.2 billion by 2001. Things get worse for TTC in the next two years, reaching a deficit of 2.1 billion by 2003. PIT does make a correction quickly after 2001, and by 2004 the reserves have a surplus. Yet, both TTC and PIT lead to more deficit years than surplus years in this exercise.

The linear regression model shows more of a counter-cyclical effect than the alternatives in this exercise. The money set aside using the linear regression model reaches close to a billion in 1997 and 1998. High levels of unused reserves translate into a slower pace of lending (not reflected in the exercise, because we exogenously impose the portfolio value).

Moreover, capital is only slightly impacted during the 2001 recession thanks to the reserves accumulated over the previous expansion. This translates into more capital available to back up further lending, if desired, during the economic recovery.

The last backtesting tool we discuss is the use of prediction intervals. Linear regression models provide standard formulas to compute confidence intervals for the values of new observations. These intervals are shown in the next figure for the 10 years spanned in the backtesting exercise.

```
figure
plot(YearsBT,DefRate(iYearsBT),'ko','LineWidth',1.5,'MarkerSize',10,...
   'MarkerFaceColor','g')
hold on
plot(YearsBT,PredDefRate(:,1),'b-s','LineWidth',1.2,'MarkerSize',10)
plot(YearsBT,[PredDefLoBnd PredDefUpBnd],'b:','LineWidth',1.2)
hold off
strConf = num2str((1-alpha)*100);
title(['{\bf Backtesting Results with ' strConf '% Prediction Intervals}'])
xlabel('Year');
ylabel('%');
legend({'Actual','Predicted','Conf Bounds'},'location','NW');
```



The observed default rates fall outside the prediction intervals for two years, 1996 and 1997, where very low default rates are observed. For a 95% confidence level, two out of 10 seems high. Yet, the observed values in these cases fall barely outside the prediction interval, which is a positive sign for the model. It is also positive that the prediction intervals contain the observed values around the 2001 recession.

Stress Testing

Stress testing is a broad area that reaches far beyond computational tools; see, for example, [3]. We show some tools that can be incorporated into a comprehensive stress testing framework. We build on the linear regression model presented above, but the concepts and tools are compatible with other forecasting methodologies.

The first tool is the use of prediction intervals to define a worst-case scenario forecasts. This is to account for uncertainty in the model only, not in the value of the predictors.

We take a baseline scenario of predictors, in our case, the latest known values of our age proxy AGE, corporate profits forecast, CPF, and corporate spread, SPR. We then use the linear regression model to compute a 95% confidence upper bound for the predicted default rate. The motivation for this is illustrated in the last plot of the backtesting section, where the 95% confidence upper limit acts as a conservative bound when the prediction underestimates the actual default rates.

```
tCrit = tinv(1-alpha/2,stats.tstat.dfe);
XLast = [AGE(end),CPF(end),SPR(end)];

yPred = [1 XLast]*stats.beta;
PredStd = sqrt([1 XLast]*stats.covb*[1 XLast]'+stats.mse);
yPredUB = yPred + tCrit*PredStd;

fprintf('\nPredicted default rate:\n');
```

```
Predicted default rate:
```

```
fprintf('     Baseline: %4.2f%%\n',yPred);
```

```
     Baseline: 1.18%
```

```
fprintf('     %g%% Upper Bound: %4.2f%%\n',(1-alpha)*100,yPredUB);
```

```
     95% Upper Bound: 2.31%
```

The next step is to incorporate stressed scenarios of the predictors in the analysis. CPF and SPR can change in the short term, whereas AGE cannot. This is important. The corporate profits forecast and the corporate spread are influenced by world events, including, for example, natural disasters. These predictors can significantly change overnight. On the other hand, AGE depends on managerial decisions that can alter the proportion of old and new loans in time, but these decisions take months, if not years, to reflect in the AGE time series. Scenarios for AGE are compatible with longer term analyses. Here we look at one year ahead only, and keep AGE fixed for the remainder of this section.

It is convenient to define the predicted default rate and the confidence bounds as functions of CPF and SPR to simplify the scenario analysis.

```
yPredFn = @(cpf,spr) [1 AGE(end) cpf spr]*stats.beta;
PredStdFn = @(cpf,spr) sqrt([1 AGE(end) cpf spr]*stats.covb*...
```

```
   [1 AGE(end) cpf spr]'+stats.mse);
yPredUBFn = @(cpf,spr) (yPredFn(cpf,spr) + tCrit*PredStdFn(cpf,spr));
yPredLBFn = @(cpf,spr) (yPredFn(cpf,spr) - tCrit*PredStdFn(cpf,spr));
```

Two extreme scenarios of interest can be a drop in the corporate profits forecast of 4% relative to the baseline, and an increase in the corporate spread of 100 basis points over the baseline.

Moving one predictor at a time is not unreasonable in this case, because the correlation between `CPF` and `SPR` is very low. Moderate correlation levels may require perturbing predictors together to get more reliable results. Highly correlated predictors usually do not coexist in the same model, since they offer redundant information.

```
fprintf('\n\n         What-if Analysis\n');
```

```
         What-if Analysis
```

```
fprintf('Scenario         LB    Pred    UB\n');
```

```
Scenario         LB    Pred    UB
```

```
cpf = CPF(end)-4;
spr = SPR(end);
yPredRange = [yPredLBFn(cpf,spr),yPredFn(cpf,spr),yPredUBFn(cpf,spr)];
fprintf('CPF drops 4%%    %4.2f%%  %4.2f%%  %4.2f%%\n',yPredRange);
```

```
CPF drops 4%    0.42%  1.57%  2.71%
```

```
cpf = CPF(end);
spr = SPR(end)+1;
yPredRange = [yPredLBFn(cpf,spr),yPredFn(cpf,spr),yPredUBFn(cpf,spr)];
fprintf('SPR rises 1%%    %4.2f%%  %4.2f%%  %4.2f%%\n',yPredRange);
```

```
SPR rises 1%    0.71%  1.88%  3.05%
```

```
cpf = CPF(end);
spr = SPR(end);
yPredRange = [yPredLBFn(cpf,spr),yPredFn(cpf,spr),yPredUBFn(cpf,spr)];
fprintf('    Baseline    %4.2f%%  %4.2f%%  %4.2f%%\n',yPredRange);
```

```
    Baseline    0.04%  1.18%  2.31%
```

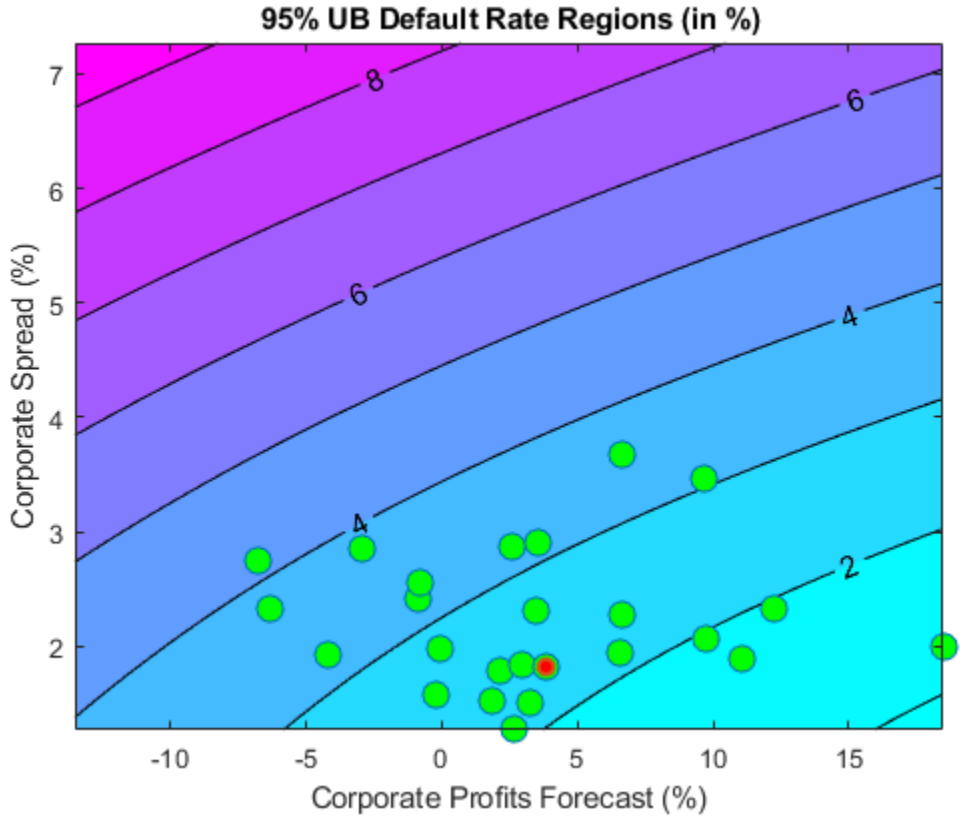
```
fprintf('\nCorrelation between CPF and SPR: %4.3f\n',corr(CPF,SPR));
```

```
Correlation between CPF and SPR: 0.012
```

We now take a more global view of the scenario analysis. Instead of analyzing one scenario at a time, we visualize the default rate forecasts as a function of `CPF` and `SPR`. More precisely, we plot default rate contours over a whole grid of `CPF` and `SPR` values. We use the conservative 95% upper bound.

If we assumed a particular bivariate distribution for the values of `CPF` and `SPR`, we could plot the contours of their distribution in the same figure. That would give visual information on the probability of falling on each region. Lacking such a distribution, we simply add to the plot the `CPF - SPR` pairs observed in our sample, as a historical, empirical distribution. The last observation in the sample, the baseline scenario, is marked in red.

```
gridCPF = 2*min(CPF):0.1:max(CPF);
gridSPR = min(SPR):0.1:2*max(SPR);
nGridCPF = length(gridCPF);
nGridSPR = length(gridSPR);

DefRateUB = zeros(nGridCPF,nGridSPR);
for i=1:nGridCPF
   for j=1:nGridSPR
      DefRateUB(i,j) = yPredUBFn(gridCPF(i),gridSPR(j));
   end
end
Example_StressTestPlot(gridCPF,gridSPR,DefRateUB,CPF,SPR,...
   'Corporate Profits Forecast (%)','Corporate Spread (%)',...
   ['{\bf ' strConf '% UB Default Rate Regions (in %)}'])
```



Very different predictor values result in similar default rate levels. For example, consider a profits forecast around 10% with a spread of 3.5%, and a profits forecast of -2.5% with a spread of 2%, they both result in a default rate slightly above 3%. Also, only one point in the available history yields a default rate higher than 4%.

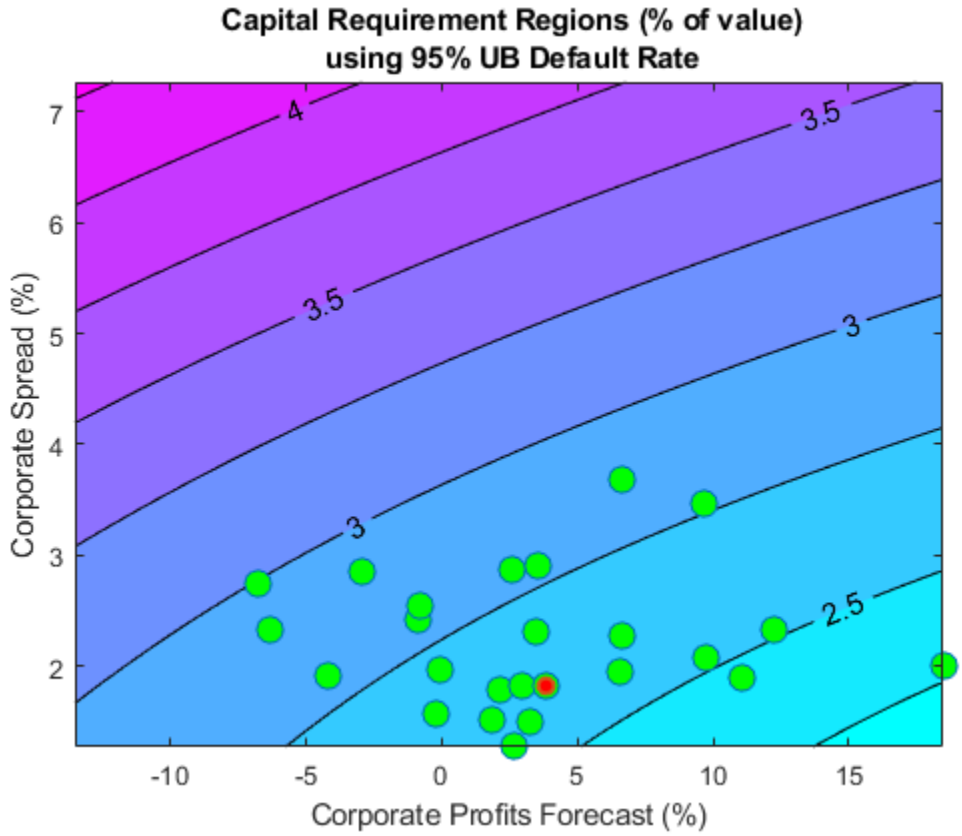
Monetary terms, once again, may be more meaningful. We use Basel II's capital requirements formula (see [2]) to translate the default rates into a monetary measure. Basel II's formula is convenient because it is analytic (there is no need to simulate to estimate the capital requirements), but also because it depends only on the probabilities of default. We define Basel II's capital requirements as a function `K`.

```
% Correlation as a function of PD
w = @(pd) (1-exp(-50*pd))/(1-exp(-50)); % weight
```

```
R = @(pd) (0.12*w(pd)+0.24*(1-w(pd))); % correlation
% Vasicek formula
V = @(pd) normcdf(norminv(pd)+R(pd).*norminv(0.999)./sqrt(1-R(pd)));
% Parameter b for maturity adjustment
b = @(pd) (0.11852-0.05478*log(pd)).^2;
% Basel II capital requirement with LGD=45% and maturity M=2.5 (numerator
% in maturity adjustment term becomes 1)
K = @(pd) 0.45*(V(pd)-pd).*(1./(1-1.5*b(pd)));
```

Worst-case default rates for a whole grid of `CPF` - `SPR` pairs are stored in `DefRateUB`. By applying the function `K` to `DefRateUB`, we can visualize the capital requirements over the same grid.

```
CapReq = 100*K(DefRateUB/100);
Example_StressTestPlot(gridCPF,gridSPR,CapReq,CPF,SPR,...
   'Corporate Profits Forecast (%)','Corporate Spread (%)',...
   {'{\bf Capital Requirement Regions (% of value)}';...
   ['{\bf using ' strConf '% UB Default Rate}']})
```



The contour levels now indicate capital requirements as a percentage of portfolio value. The two scenarios above, profits of 10% with spread of 3.5%, and profits of -2.5% and spread of 2%, result in capital requirements near 2.75%. The worst-case point from the historical data yields a capital requirement of about 3%.

This visualization can also be used, for example, as part of a reverse stress test analysis. Critical levels of capital can be determined first, and the figure can be used to determine regions of risk factor values (in this case `CPF` and `SPR`) that lead to those critical levels.

Instead of historical observations of `CPF` and `SPR`, an empirical distribution for the risk factors can be simulated using, for example, a vector autoregressive (VAR) model from Econometrics Toolbox™. The capital requirements corresponding to each default probability level can be found by simulation if a closed form formula is not available, and

the same plots can be generated. For large simulations, a distributed computing implementation using Parallel Computing Toolbox™ or MATLAB Distributed Computing Server™ can make the process more efficient.

Appendix: Modeling Full Transition Matrices

Transition matrices change in time, and a full description of their dynamics requires working with multi-dimensional time series. There are, however, techniques that exploit the particular structure of transition matrices to reduce the dimensionality of the problem. In [8], for example, a single parameter related to the proportion of downgrades is used, and both [6] and [8] describe a method to shift transition probabilities using a single parameter. The latter approach is shown in this appendix.

The method takes the TTC transition matrix as a baseline.

```
tmTTC = transprobbytotals(totalsByRtg);
Example_DisplayTransitions(tmTTC,[],...
   {'AAA','AA','A','BBB','BB','B','CCC'},...
   {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	88.2	7.67	0.49	0.09	0.06	0	0	0	3.49
AA	0.58	87.16	7.63	0.58	0.06	0.11	0.02	0.01	3.85
A	0.05	1.9	87.24	5.59	0.42	0.15	0.03	0.04	4.58
BBB	0.02	0.16	3.85	84.13	4.27	0.76	0.17	0.27	6.37
BB	0.03	0.04	0.25	5.26	75.74	7.36	0.9	1.12	9.29
B	0	0.05	0.19	0.31	5.52	72.67	4.21	5.38	11.67
CCC	0	0	0.28	0.41	1.24	10.92	47.06	27.02	13.06

An equivalent way to represent this matrix is by transforming it into credit quality thresholds, that is, critical values of a standard normal distribution that yield the same transition probabilities (row by row).

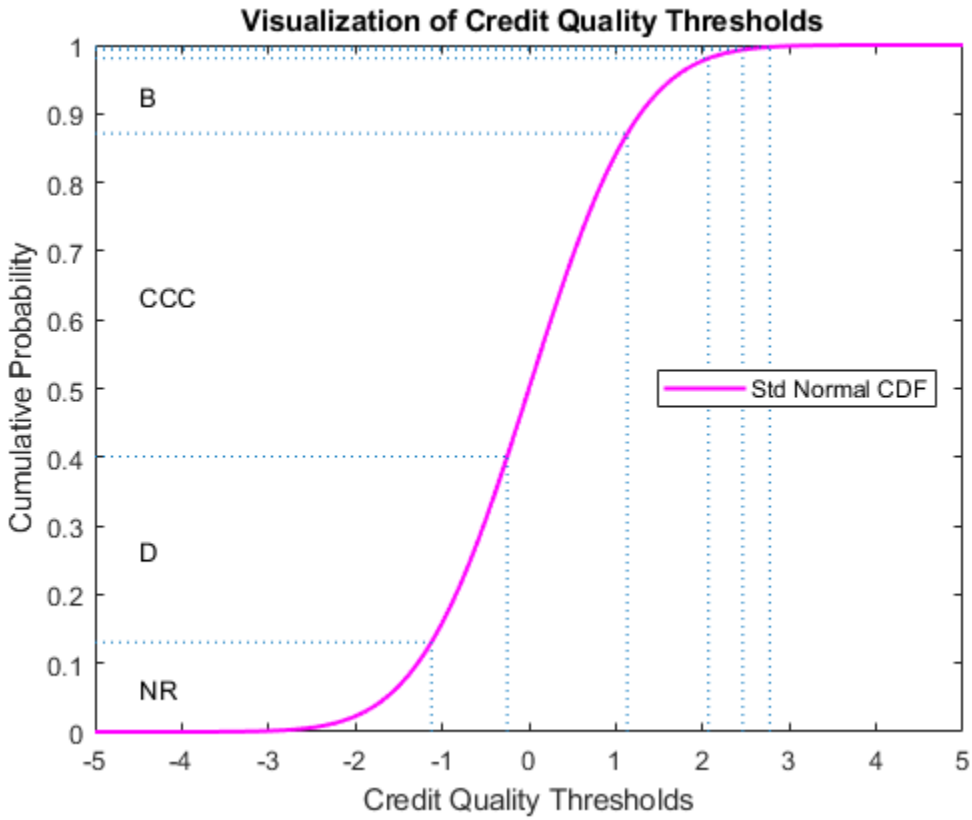
```
thresholdMat = transprobtothresholds(tmTTC);
Example_DisplayTransitions(thresholdMat,[],...
   {'AAA','AA','A','BBB','BB','B','CCC'},...
   {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	Inf	-1.19	-1.74	-1.8	-1.81	-1.81	-1.81	-1.81	-1.81
AA	Inf	2.52	-1.16	-1.68	-1.75	-1.75	-1.76	-1.77	-1.77
A	Inf	3.31	2.07	-1.24	-1.62	-1.66	-1.68	-1.68	-1.69
BBB	Inf	3.57	2.91	1.75	-1.18	-1.43	-1.49	-1.5	-1.52
BB	Inf	3.39	3.16	2.72	1.59	-0.89	-1.21	-1.26	-1.32

```
     B    Inf    Inf   3.28   2.82   2.54   1.55   -0.8  -0.95  -1.19
   CCC    Inf    Inf    Inf   2.77   2.46   2.07   1.13  -0.25  -1.12
```

Credit quality thresholds are illustrated in the following figure. The segments in the vertical axis represent transition probabilities, and the boundaries between them determine the critical values in the horizontal axis, via the standard normal distribution. Each row in the transition matrix determines a set of thresholds. The figure shows the thresholds for the `'CCC'` rating.

```
xliml = -5;
xlimr = 5;
step = 0.1;
x=xliml:step:xlimr;
thresCCC = thresholdMat(7,:);
centersY = (normcdf([thresCCC(2:end) xliml])+...
   normcdf([xlimr thresCCC(2:end)]))/2;
labels = {'AAA','AA','A','BBB','BB','B','CCC','D','NR'};

figure
plot(x,normcdf(x),'m','LineWidth',1.5)
for i=2:length(labels)
   val = thresCCC(i);
   line([val val],[0 normcdf(val)],'LineStyle',':');
   line([x(1) val],[normcdf(val) normcdf(val)],'LineStyle',':');
   if (centersY(i-1)-centersY(i))>0.05
      text(-4.5,centersY(i),labels{i});
   end
end
xlabel('Credit Quality Thresholds')
ylabel('Cumulative Probability')
title('{\bf Visualization of Credit Quality Thresholds}')
legend('Std Normal CDF','Location','E')
```



Shifting the critical values to the right or left changes the transition probabilities. For example, here is the transition matrix obtained by shifting the TTC thresholds by 0.5 to the right. Note that default probabilities increase.

```
shiftedThresholds = thresholdMat+0.5;
Example_DisplayTransitions(transprobfromthresholds(shiftedThresholds),...
   [],{'AAA','AA','A','BBB','BB','B','CCC'},...
   {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

	AAA	AA	A	BBB	BB	B	CCC	D	NR
AAA	75.34	13.84	1.05	0.19	0.13	0	0	0	9.45
AA	0.13	74.49	13.53	1.21	0.12	0.22	0.04	0.02	10.24
A	0.01	0.51	76.4	10.02	0.83	0.31	0.06	0.08	11.77
BBB	0	0.03	1.2	74.03	7.22	1.39	0.32	0.51	15.29

```
    BB      0   0.01   0.05   1.77  63.35  10.94   1.47   1.88  20.52
     B      0   0.01   0.04   0.07   1.91  59.67   5.74    8.1  24.46
   CCC      0      0   0.05    0.1   0.36   4.61  35.06  33.18  26.65
```

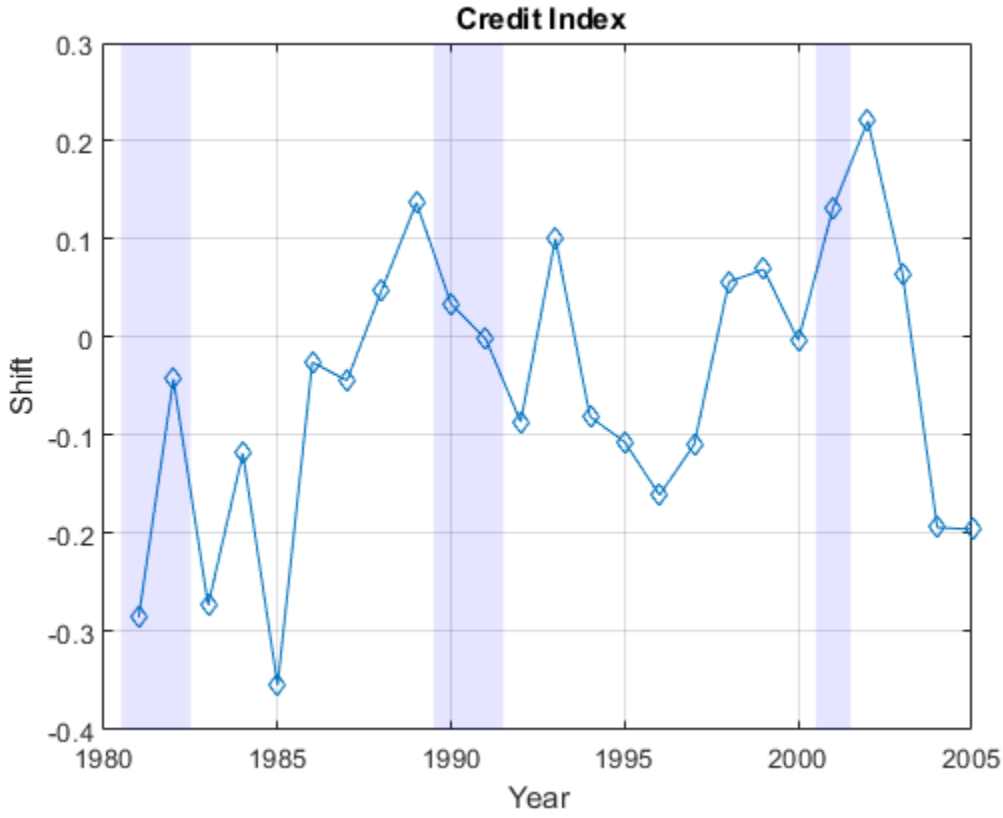
Given a particular PIT matrix, the idea in [6] and [8] is to vary the shifting parameter applied to the TTC thresholds so that the resulting transition matrix is as close as possible to the PIT matrix. Closeness is measured as the sum of squares of differences between corresponding transition probabilities. The optimal shifting value is called credit index. A credit index is determined for every PIT transition matrix in the sample.

Here we use `fminunc` from Optimization Toolbox™ to find the credit indices.

```
CreditIndex = zeros(nYears,1);
ExitFlag = zeros(nYears,1);
options = optimset('LargeScale','Off','Display','Off');
for i=1:nYears
   errorfun = @(z)norm(squeeze(TransMat(:,:,i))-...
      transprobfromthresholds(...
      transprobtothresholds(tmTTC)+z),'fro');
   [CreditIndex(i),~,ExitFlag(i)] = fminunc(errorfun,0,options);
end
```

In general, one expects that higher credit indices correspond to riskier years. The series of credit indices found does not entirely match this pattern. There may be different reasons for this. First, transition probabilities may deviate from their long-term averages in different ways that may lead to confounding effects in the single parameter trying to capture these differences, the credit index. Having separate credit indices for IG and SG, for example, may help separate confounding effects. Second, a difference of five basis points may be very significant for the `'BBB'` default rate, but not as important for the `'CCC'` default rate, yet the norm used weights them equally. Other norms can be considered. Also, it is always a good idea to check the exit flags of optimization solvers, in case the algorithm could not find a solution. Here we get valid solutions for each year (all exit flags are `1`).

```
figure
plot(Years,CreditIndex,'-d')
hold on
Example_RecessionBands
hold off
grid on
xlabel('Year')
ylabel('Shift')
title('{\bf Credit Index}')
```



The workflow above can be adapted to work with the series of credit indices instead of the series of corporate default rates. A model can be fit to predict a credit index for the following year, and a predicted transition matrix can be inferred and used for risk analyses.

References

[1] Altman, E., and E. Hotchkiss, *Corporate Financial Distress and Bankruptcy*, third edition, New Jersey: Wiley Finance, 2006.

[2] Basel Committee on Banking Supervision, "International Convergence of Capital Measurement and Capital Standards: A Revised Framework," Bank for International

Settlements (BIS), comprehensive version, June 2006. Available at: http://www.bis.org/publ/bcbsca.htm.

[3] Basel Committee on Banking Supervision, "Principles for Sound Stress Testing Practices and Supervision - Final Paper," Bank for International Settlements (BIS), May 2009. Available at: http://www.bis.org/publ/bcbs155.htm.

[4] FRED, St. Louis Federal Reserve, Federal Reserve Economic Database, http://research.stlouisfed.org/fred2/.

[5] Helwege, J., and P. Kleiman, "Understanding Aggregate Default Rates of High Yield Bonds," Federal Reserve Bank of New York, Current Issues in Economics and Finance, Volume 2, Number 6, May 1996.

[6] Loeffler, G., and P. N. Posch, *Credit Risk Modeling Using Excel and VBA*, West Sussex, England: Wiley Finance, 2007.

[7] NBER, National Bureau of Economic Research, Business Cycle Expansions and Contractions, http://www.nber.org/cycles/.

[8] Otani, A., S. Shiratsuka, R. Tsurui, and T. Yamada, "Macro Stress-Testing on the Loan Portfolio of Japanese Banks," Bank of Japan Working Paper Series No.09-E-1, March 2009.

[9] Survey of Professional Forecasters, Federal Reserve Bank of Philadelphia, http://www.philadelphiafed.org/.

[10] Vazza, D., D. Aurora, and R. Schneck, "Annual 2005 Global Corporate Default Study And Rating Transitions," Standard & Poor's, Global Fixed Income Research, New York, January 2006.

[11] Wilson, T. C., "Portfolio Credit Risk," FRBNY Economic Policy Review, October 1998.

See Also

`bootstrp` | `transprob` | `transprobbytotals` | `transprobfromthresholds` | `transprobgrouptotals` | `transprobprep` | `transprobtothresholds`

Related Examples

- “Credit Quality Thresholds” on page 8-52
- “Credit Rating by Bagging Decision Trees” (Statistics and Machine Learning Toolbox)

External Websites

- Credit Risk Modeling with MATLAB (53 min 09 sec)
- Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Credit Quality Thresholds

In this section...

Introduction

An equivalent way to represent transition probabilities is by transforming them into credit quality thresholds. These are critical values of a standard normal distribution that yield the same transition probabilities.

An `M`-by-`N` matrix of transition probabilities `TRANS` and the corresponding `M`-by-`N` matrix of credit quality thresholds `THRESH` are related as follows. The thresholds `THRESH`(*i*,*j*) are critical values of a standard normal distribution *z*, such that

```
TRANS(i,N) = P[z < THRESH(i,N)],

TRANS(i,j) = P[z < THRESH(i,j)] - P[z < THRESH(i,j+1)], for 1<=j<N
```

Financial Toolbox supports the transformation between transition probabilities and credit quality thresholds with the functions `transprobtothresholds` and `transprobfromthresholds`.

Compute Credit Quality Thresholds

To compute credit quality thresholds, transition probabilities are required as input. Here is a transition matrix estimated from credit ratings data:

```
load Data_TransProb
trans = transprob(data)

trans =

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001    0.0017
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004    0.0396
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028    0.0753
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642    0.2193
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919    0.7050
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169    2.4399
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927   12.7167
         0         0         0         0         0         0         0  100.0000
```

Convert the transition matrix to credit quality thresholds using `transprobtothresholds`:

```
thresh = transprobtothresholds(trans)
```

```
thresh =

       Inf   -1.4846   -2.3115   -2.8523   -3.3480   -4.0083   -4.1276   -4.1413
       Inf    2.1403   -1.6228   -2.3788   -2.8655   -3.3166   -3.3523   -3.3554
       Inf    3.0264    1.8773   -1.6690   -2.4673   -2.9800   -3.1631   -3.1736
       Inf    3.4963    2.8009    1.6201   -1.6897   -2.4291   -2.7663   -2.8490
       Inf    3.5195    2.9999    2.4225    1.5089   -1.7010   -2.3275   -2.4547
       Inf    4.2696    3.8015    3.0477    2.3320    1.3838   -1.6491   -1.9703
       Inf    4.6241    4.2097    3.6472    2.7803    2.1199    1.5556   -1.1399
       Inf       Inf       Inf       Inf       Inf       Inf       Inf       Inf
```

Conversely, given a matrix of thresholds, you can compute transition probabilities using `transprobfromthresholds`. For example, take the thresholds computed previously as input to recover the original transition probabilities:

```
trans1 = transprobfromthresholds(thresh)
```

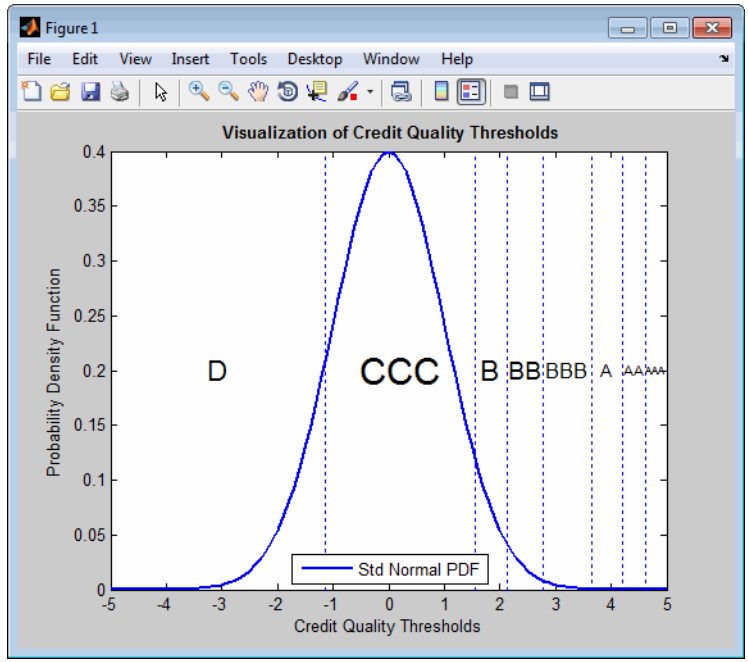
```
trans1 =

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001    0.0017
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004    0.0396
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028    0.0753
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642    0.2193
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919    0.7050
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169    2.4399
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927   12.7167
         0         0         0         0         0         0         0  100.0000
```

Visualize Credit Quality Thresholds

You can graphically represent the relationship between credit quality thresholds and transition probabilities. Here, this example shows the relationship for the `'CCC'` credit rating. In the plot, the thresholds are marked by the vertical lines and the transition probabilities are the area below the standard normal density curve:

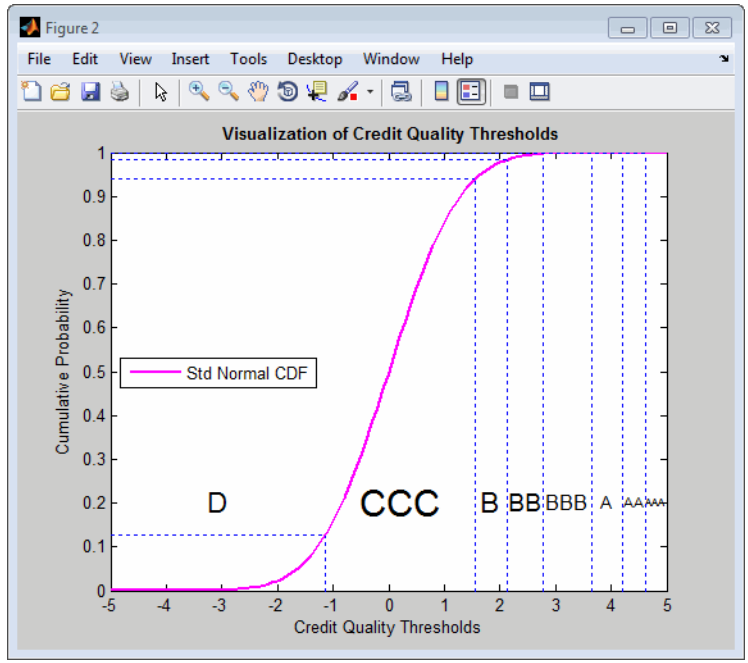
```
load Data_TransProb
trans = transprob(data);
thresh = transprobtothresholds(trans);

xliml = -5;
xlimr = 5;
step = 0.1;
x=xliml:step:xlimr;
thresCCC = thresh(7,:);
labels = {'AAA','AA','A','BBB','BB','B','CCC','D'};
```

```
centersX = ([5 thresCCC(2:end)]+[thresCCC(2:end) -5])*0.5;
omag = round(log10(trans(7,:)));
omag(omag>0)=omag(omag>0).^2;
fs = 14+2*omag;

figure
plot(x,normpdf(x),'LineWidth',1.5)
text(centersX(1),0.2,labels{1},'FontSize',fs(1),...
   'HorizontalAlignment','center')
for i=2:length(labels)
   val = thresCCC(i);
   line([val val],[0 0.4],'LineStyle',':')
   text(centersX(i),0.2,labels{i},'FontSize',fs(i),...
      'HorizontalAlignment','center')
end
xlabel('Credit Quality Thresholds')
ylabel('Probability Density Function')
title('{\bf Visualization of Credit Quality Thresholds}')
legend('Std Normal PDF','Location','S')
```



The second plot uses the cumulative density function instead. The thresholds are represented by vertical lines. The transition probabilities are given by the distance between horizontal lines.

```
figure
plot(x,normcdf(x),'m','LineWidth',1.5)
text(centersX(1),0.2,labels{1},'FontSize',fs(1),...
   'HorizontalAlignment','center')
for i=2:length(labels)
   val = thresCCC(i);
   line([val val],[0 normcdf(val)],'LineStyle',':');
   line([x(1) val],[normcdf(val) normcdf(val)],'LineStyle',':');
   text(centersX(i),0.2,labels{i},'FontSize',fs(i),...
      'HorizontalAlignment','center')
end
xlabel('Credit Quality Thresholds')
ylabel('Cumulative Probability')
title('{\bf Visualization of Credit Quality Thresholds}')
legend('Std Normal CDF','Location','W')
```



See Also

`bootstrp` | `transprob` | `transprobbytotals` | `transprobfromthresholds` | `transprobgrouptotals` | `transprobprep` | `transprobtothresholds`

Related Examples

- "Estimation of Transition Probabilities" on page 8-2
- "Credit Rating by Bagging Decision Trees" (Statistics and Machine Learning Toolbox)
- "Forecasting Corporate Default Rates" on page 8-20

External Websites

- Credit Risk Modeling with MATLAB (53 min 09 sec)
- Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

About Credit Scorecards

In this section...

What Is a Credit Scorecard?

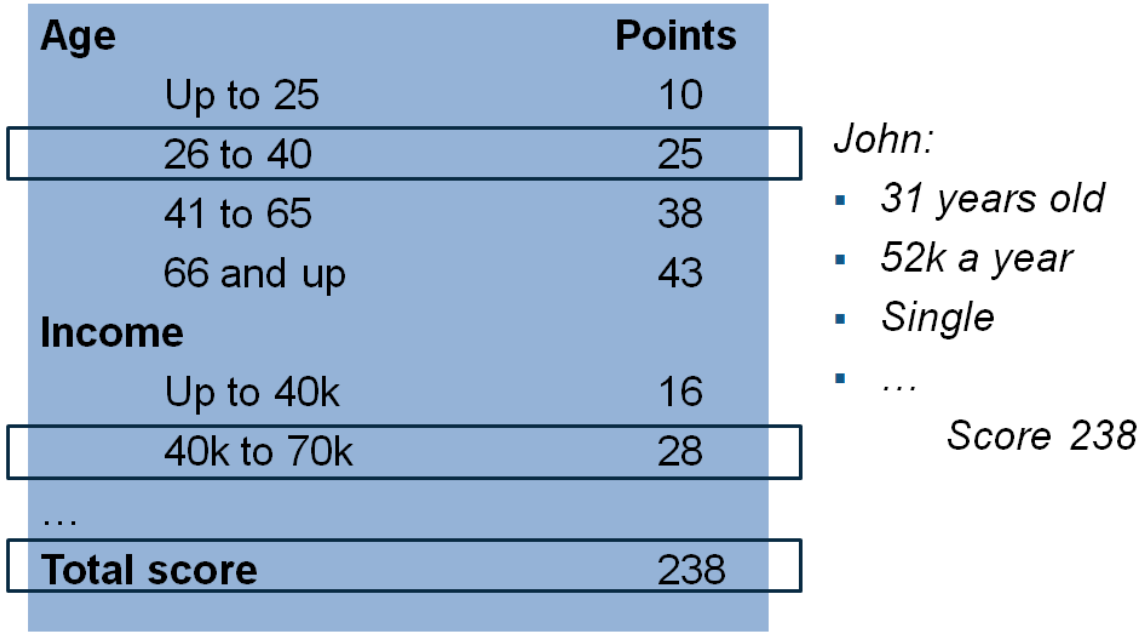
Credit scoring is one of the most widely used credit risk analysis tools. The goal of credit scoring is ranking borrowers by their credit worthiness. In the context of retail credit (credit cards, mortgages, car loans, etc.), credit scoring is performed using a credit scorecard. Credit scorecards represent different characteristics of a customer (age, residential status, time at current address, time at current job, and so on) translated into points and the total number of points becomes the credit score. The credit worthiness of customers is summarized by their credit score; high scores usually correspond to low-risk customers, and conversely. Scores are also used for corporate credit analysis of small and medium enterprises, and, large corporations.

A credit scorecard is a lookup table that maps specific characteristics of a borrower into points. The total number of points becomes the credit score. Credit scorecards are a widely used type of credit scoring model. As such, the goal of a credit scorecard is to distinguish between customers who repay their loans (“good” customers), and customers who will not (“bad” customers). Like other credit scoring models, credit scorecards quantify the risk that a borrower will not repay a loan in the form of a score and a probability of default.

For example, a credit scorecard can give individual borrowers points for their age and income according to the following table. Other characteristics such as residential status, employment status, might also be included, although, for brevity, they are not shown in this table.

Age	Points
Up to 25	10
26 to 40	25
41 to 65	38
66 and up	43
Income	
Up to 40k	16
40k to 70k	28
…	
Total score	(Sum of Points)

Using the credit scorecard in this example, a particular customer who is 31 and has an income of $52,000 a year, is placed into the second age group (26–40) and receives 25 points for their age, and similarly, receives 28 points for their income. Other characteristics (not shown here) might contribute additional points to their score. The total score is the sum of all points, which in this example is assumed to give the customer a total of 238 points (this is a fictitious example on an arbitrary scoring scale).



Technically, to determine the credit scorecard points, start out by selecting a set of potential predictors (column 1 in the next figure). Then, bin data into groups (for example, ages 'Up to 25', '25 to 40' (column 2 in the figure). This grouping helps to distinguish between "good" and "bad" customers. The Weight of Evidence (WOE) is a way to measure how well the distribution of "good" and "bad" are separated across bins or groups for each individual predictor (column 3 in the figure). By fitting a logistic regression model, you can identify which predictors, when put together, do a better job distinguishing between "good" and "bad" customers. The model is summarized by its coefficients (column 4 in the figure). Finally, the combination of WOE's and model coefficients (commonly scaled, shifted, and rounded) make up the scorecard points (column 5 in the figure).

Predictor	Bin	WOE	Model	Points*
Age			β_{age}	
	'Up to 25'	WOE_{age}('Up to 25')		β_{age}*WOE_{age}('Up to 25')
	'26 to 40'	WOE_{age}('26 to 40')		β_{age}*WOE_{age}('26 to 40')
	'41 to 65'	WOE_{age}('41 to 65')		β_{age}*WOE_{age}('41 to 65')
	'66 and up'	WOE_{age}('66 and up')		β_{age}*WOE_{age}('66 and up')
Income			β_{income}	
	'Up to 40k'	WOE_{income}('Up to 40k')		β_{income}*WOE_{income}('Up to 40k')
	'40k to 70k'	WOE_{income}('40k to 70k')		β_{income}*WOE_{income}('40k to 70k')
	...			

* Points may include a constant and may be scaled and rounded.

Credit Scorecard Development Process

1 Data gathering and preparation phase

This includes data gathering and integration, such as querying, merging, aligning. It also includes treatment of missing information and outliers. There is a prescreening step based on reports of association measures between the predictors and the response variable. Finally, there is a sampling step, to produce a training set, sometimes called the modeling view, and usually a validation set, too. The training set, in the form of a table, is the required data input to the `creditscorecard` object, and this training set table must be prepared before creating a `creditscorecard` object in the Modeling phase.

2 Modeling phase

Use the `creditscorecard` object and associated object functions to develop a credit scorecard model. You can bin the data, apply the Weight of Evidence (WOE) transformation, and compute other statistics, such as the Information Value. You can fit a logistic regression model and also review the resulting scorecard points and format their scaling and rounding. For details on using the `creditscorecard` object, see `creditscorecard`.

3 Deployment phase

Deployment entails integrating a credit scorecard model into an IT production environment and keeping tracking logs, performance reports, and so on.

The `creditscorecard` object is designed for the Modeling phase of the credit scorecard workflow. Support for all three phases requires other MathWorks® products.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Related Examples

- "Troubleshooting Credit Scorecard Results" on page 8-68
- "Case Study for a Credit Scorecard Analysis" on page 8-78

More About

- "Credit Scorecard Modeling Workflow" on page 8-62
- "Credit Scorecard Modeling Using Observation Weights" on page 8-65

Credit Scorecard Modeling Workflow

Create, model, and analyze credit scorecards as follows.

1. Create a `creditscorecard` object.

   Create a `creditscorecard` object for credit scorecard analysis by specifying "training" data in table format. The training data, sometimes called the modeling view, is the result of multiple data preparation tasks (see "About Credit Scorecards" on page 8-57) that must be performed before creating a `creditscorecard` object.

   You can use optional input arguments for `creditscorecard` to specify scorecard properties such as the response variable and the `GoodLabel`. Perform some initial data exploration when the `creditscorecard` object is created, although data analysis is usually done in combination with data binning (see step 2). For more information and examples, see `creditscorecard` and step 1 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

2. Bin the data.

   Perform manual or automatic binning of the data loaded into the `creditscorecard` object.

   A common starting point is to apply automatic binning to all or selected variables using `autobinning`, report using `bininfo`, and visualize bin information with respect to bin counts and statistics or association measures such as Weight of Evidence (WOE) using `plotbins`. The bins can be modified or fine-tuned either manually using `modifybins` or with a different automatic binning algorithm using `autobinning`. Bins that show a close-to-linear trend in the WOE are frequently desired in the credit scorecard context.

   Alternatively, with Risk Management Toolbox™, you can use the **Binning Explorer** app to interactively bin. The **Binning Explorer** enables you to interactively apply a binning algorithm and modify bins. For more information, see **Binning Explorer**.

   For more information and examples, see `autobinning`, `modifybins`, `bininfo`, and `plotbins` and step 2 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

3. Fit a logistic regression model.

   Fit a logistic regression model to the WOE data from the `creditscorecard` object. The `fitmodel` function internally bins the training data, transforms it into WOE

values, maps the response variable so that `'Good'` is `1`, and fits a linear logistic regression model.

By default, `fitmodel` uses a stepwise procedure to determine which predictors should be in the model, but optional input arguments can also be used, for example, to fit a full model. For more information and examples, see `fitmodel` and step 3 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

4 Review and format credit scorecard points.

After fitting the logistic model, use `displaypoints` to summarize the scorecard points. By default, the points are unscaled and come directly from the combination of Weight of Evidence (WOE) values and model coefficients.

The `formatpoints` function lets you control scaling and rounding of scorecard points. For more information and examples, see `displaypoints` and `formatpoints` and step 4 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

5 Score the data.

The `score` function computes the scores for the training data.

An optional data input can also be passed to `score`, for example, validation data. The points per predictor for each customer are also provided as an optional output. For more information and examples, see `score` and step 5 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

6 Calculate the probability of default for credit scorecard scores.

The `probdefault` function to calculate the probability of default for training data.

In addition, you can compute likelihood of default for a different dataset (for example, a validation data set) using the `probdefault` function. For more information and examples, see `probdefault` and step 6 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

7 Validate the credit scorecard model.

Use the `validatemodel` function to validate the quality of the credit scorecard model.

You can obtain the Cumulative Accuracy Profile (CAP), Receiver Operating Characteristic (ROC), and Kolmogorov-Smirnov (KS) plots and statistics for a given

dataset using the `validatemodel` function. For more information and examples, see `validatemodel` and step 7 in "Case Study for a Credit Scorecard Analysis" on page 8-78.

For an example of this workflow, see "Case Study for a Credit Scorecard Analysis" on page 8-78.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Related Examples

- "Troubleshooting Credit Scorecard Results" on page 8-68
- "Case Study for a Credit Scorecard Analysis" on page 8-78

More About

- "About Credit Scorecards" on page 8-57
- "Credit Scorecard Modeling Using Observation Weights" on page 8-65

Credit Scorecard Modeling Using Observation Weights

When creating a `creditscorecard` object, the table used for the input `data` argument either defines or does not define observational weights. If the `data` does not use weights, then the "counts" for `Good`, `Bad`, and `Odds` are used by credit score card functions. However, if the optional `WeightsVar` argument is specifed when creating a `creditscorecard` object, then the "counts" for `Good`, `Bad`, and `Odds` are the sum of weights.

For example,here is a snippet of an input table that does not define observational weights:

CustID	CustAge	TmAtAddress	ResStatus	EmpStatus	CustIncome	status
11	52	24	'Tenant'	'Unknown'	34000	1
12	48	91	'Other'	'Unknown'	44000	0
13	65	63	'Home Owner'	'Unknown'	48000	0
14	44	75	'Other'	'Unknown'	41000	0
15	46	82	'Other'	'Employed'	46000	0
16	33	47	'Home Owner'	'Employed'	36000	0
17	39	10	'Tenant'	'Employed'	34000	1
18	24	7	'Home Owner'	'Employed'	22000	0
19	53	14	'Home Owner'	'Employed'	51000	1
20	52	55	'Other'	'Unknown'	42000	0

If you bin the customer age predictor data, with customers up to 45 years old in one bin, and 46 and up in another bin, you get these statistics:

Bin	Good	Bad	Odds
'[-Inf,46)'	381	241	1.581
'[46,Inf]'	422	156	2.705
'Totals'	803	397	2.023

`Good` means the total number of rows with a `0` value in the `status` response variable. `Bad` the number of `1`'s in the `status` column. `Odds` is the ratio of `Good` to `Bad`. The `Good`, `Bad`, and `Odds` is reported for each bin. This means that there are 381 people in the sample who are 45 and under who paid their loans, 241 in the same age range who defaulted, and therefore, the odds of being good for that age range is `1.581`.

Suppose that the modeler thinks that people 45 and younger are underrepresented in this sample. The modeler wants to give all rows with ages up to 45 a higher weight. Assume that the modeler thinks the up to 45 age group should have 50% more weight

than rows with ages 46 and up. The table data is expanded to include the observation weights. A `Weight` column is added to the table, where all rows with ages 45 and under have a weight of `1.5`, and all other rows a weight of `1`. There are other reasons to use weights, for example, recent data points may be given higher weights than older data points.

CustID	CustAge	TmAtAddress	ResStatus	EmpStatus	CustIncome	status	Weight
11	52	24	'Tenant'	'Unknown'	34000	1	1.0
12	48	91	'Other'	'Unknown'	44000	0	1.0
13	65	63	'Home Owner'	'Unknown'	48000	0	1.0
14	44	75	'Other'	'Unknown'	41000	0	1.5
15	46	82	'Other'	'Employed'	46000	0	1.0
16	33	47	'Home Owner'	'Employed'	36000	0	1.5
17	39	10	'Tenant'	'Employed'	34000	1	1.5
18	24	7	'Home Owner'	'Employed'	22000	0	1.5
19	53	14	'Home Owner'	'Employed'	51000	1	1.0
20	52	55	'Other'	'Unknown'	42000	0	1.0

If you bin the weighted data based on age (45 and under, versus 46 and up) the expectation is that each row with age 45 and under must count as 1.5 observations, and therefore the `Good` and `Bad` "counts" are increased by 50%:

Bin	Good	Bad	Odds
'[-Inf,46)'	571.5	361.5	1.581
'[46,Inf]'	422	156	2.705
'Totals'	993.5	517.5	1.920

The "counts" are now "weighted frequencies" and are no longer integer values. The `Odds` do not change for the first bin. The particular weights given in this example have the effect of scaling the total `Good` and `Bad` counts in the first bin by the same scaling factor, therefore their ratio does not change. However, the `Odds` value of the total sample does change; the first bin now carries a higher weight, and because the odds in that bin are lower, the total `Odds` are now lower, too. Other credit scorecard statistics not shown here, such as `WOE` and `Information Value` are affected in a similar way.

In general, the effect of weights is not simply to scale frequencies in a particular bin, because members of that bin will have different weights. The goal of this example is to demonstrate the concept of switching from counts to the sum of weights.

See Also

`autobinning` | `bininfo` | `creditscorecard` | `fitmodel` | `validatemodel`

More About

- "About Credit Scorecards" on page 8-57
- "Credit Scorecard Modeling Workflow" on page 8-62

Troubleshooting Credit Scorecard Results

In this section...

This topic shows some of the results when using credit scorecards that need troubleshooting. These examples cover the full range of the credit score card workflow. For details on the overall process of creating and developing credit scorecards, see "Credit Scorecard Modeling Workflow" on page 8-62.

Predictor Name Is Unspecified and the Parser Returns an Error

If you attempt to use `modifybins`, `bininfo`, or `plotbins` and omit the predictor's name, the parser returns an error.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
modifybins(sc,'CutPoints',[20 30 50 65])
```

```
Error using creditscorecard/modifybins (line 79)
Expected a string for the parameter name, instead the input type was 'double'.
```

Solution: Make sure to include the predictor's name when using these functions. Use this syntax to specify the `PredictorName` when using `modifybins`.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
modifybins(sc,'CustIncome','CutPoints',[20 30 50 65]);
```

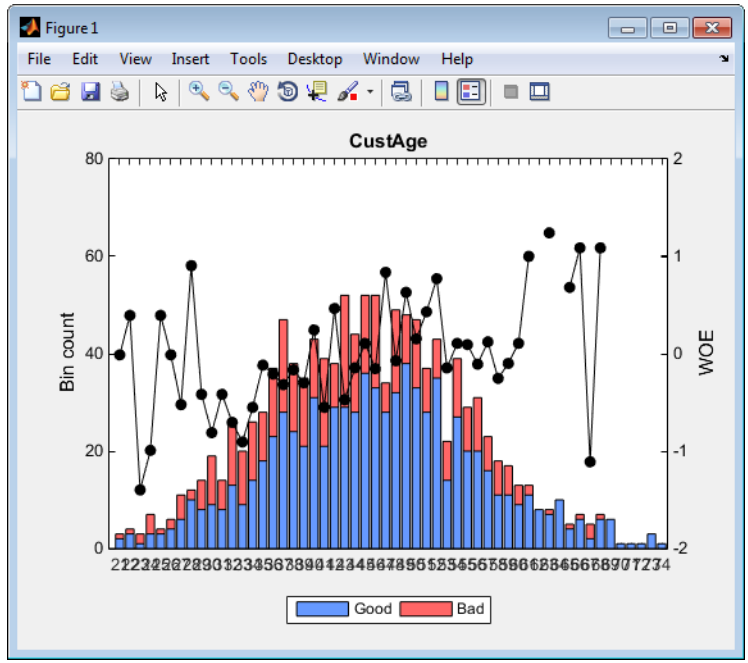
Using bininfo or plotbins Before Binning

If you use `bininfo` or `plotbins` before binning, the results might be unusable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
bininfo(sc,'CustAge')
plotbins(sc,'CustAge')
```

```
ans =
```

Bin	Good	Bad	Odds	WOE	InfoValue
'21'	2	1	2	-0.011271	3.1821e-07
'22'	3	1	3	0.39419	0.00047977
'23'	1	2	0.5	-1.3976	0.0053002
'24'	3	4	0.75	-0.9921	0.0062895
'25'	3	1	3	0.39419	0.00047977
'26'	4	2	2	-0.011271	6.3641e-07
'27'	6	5	1.2	-0.5221	0.0026744
'28'	10	2	5	0.90502	0.0067112
'29'	8	6	1.3333	-0.41674	0.0021465
'30'	9	10	0.9	-0.80978	0.011321
'31'	8	6	1.3333	-0.41674	0.0021465
'32'	13	13	1	-0.70442	0.011663
'33'	9	11	0.81818	-0.90509	0.014934
'34'	14	12	1.1667	-0.55027	0.0070391
'35'	18	10	1.8	-0.11663	0.00032342
'36'	23	14	1.6429	-0.20798	0.0013772
'37'	28	19	1.4737	-0.31665	0.0041132
'38'	24	14	1.7143	-0.16542	0.0008894
'39'	21	14	1.5	-0.29895	0.0027242
'40'	31	12	2.5833	0.24466	0.0020499
'41'	21	18	1.1667	-0.55027	0.010559
'42'	29	9	3.2222	0.46565	0.0062605
'43'	29	23	1.2609	-0.47262	0.010312
'44'	28	16	1.75	-0.1448	0.00078672
'45'	36	16	2.25	0.10651	0.00048246
'46'	33	19	1.7368	-0.15235	0.0010303
'47'	28	6	4.6667	0.83603	0.016516
'48'	32	17	1.8824	-0.071896	0.00021357
'49'	38	10	3.8	0.63058	0.013957
'50'	33	14	2.3571	0.15303	0.00089239
'51'	28	9	3.1111	0.43056	0.0052525
'52'	35	8	4.375	0.77149	0.01808
'53'	14	8	1.75	-0.1448	0.00039336
'54'	27	12	2.25	0.10651	0.00036184
'55'	20	9	2.2222	0.094089	0.00021044
'56'	20	11	1.8182	-0.10658	0.00029856
'57'	16	7	2.2857	0.12226	0.00028035
'58'	11	7	1.5714	-0.25243	0.00099297
'59'	11	6	1.8333	-0.098283	0.00013904
'60'	9	4	2.25	0.10651	0.00012061
'61'	11	2	5.5	1.0003	0.0086637
'62'	8	0	Inf	Inf	Inf
'63'	7	1	7	1.2415	0.0076953
'64'	10	0	Inf	Inf	Inf
'65'	4	1	4	0.68188	0.0016791
'66'	6	1	6	1.0873	0.0053857
'67'	2	3	0.66667	-1.1099	0.0056227
'68'	6	1	6	1.0873	0.0053857
'69'	6	0	Inf	Inf	Inf
'70'	1	0	Inf	Inf	Inf
'71'	1	0	Inf	Inf	Inf
'72'	1	0	Inf	Inf	Inf
'73'	3	0	Inf	Inf	Inf
'74'	1	0	Inf	Inf	Inf
'Totals'	803	397	2.0227	NaN	Inf

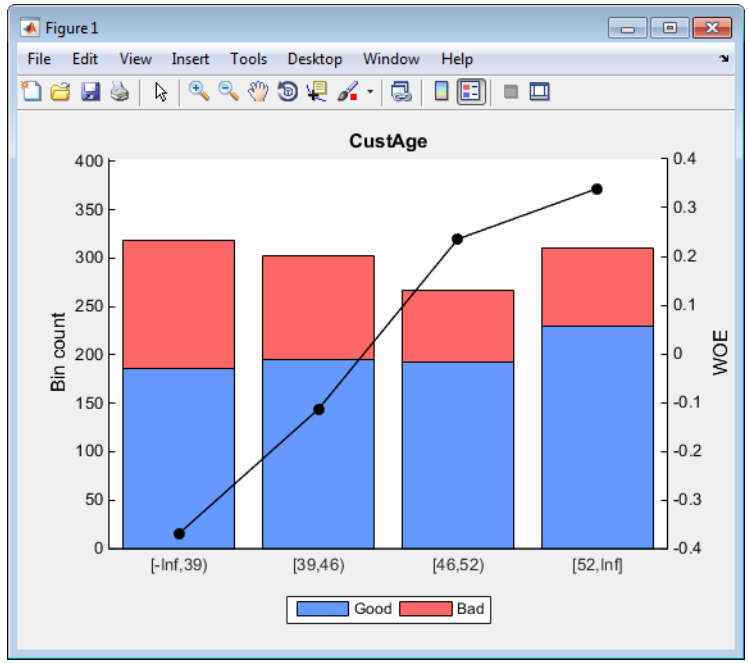


The plot for `CustAge` is not readable because it has too many bins. Additionally, `bininfo` returns data that have `Inf` values for the WOE due to zero observations for either **Good** or **Bad**.

Solution: Bin the data using `autobinning` or `modifybins` before plotting or inquiring about the bin statistics, to avoid having too many bins or having `NaN`s and `Inf`s. For example, you can use the name-value pair argument for `AlgoOptions` with the `autobinning` function to define the number of bins.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
AlgoOptions = {'NumBins',4};
sc = autobinning(sc,'CustAge','Algorithm','EqualFrequency',...
'AlgorithmOptions',AlgoOptions);
bininfo(sc,'CustAge','Totals','off')
plotbins(sc,'CustAge')

ans =

        Bin          Good    Bad      Odds       WOE        InfoValue
    _____________    ____    ___    _______    _________    _________

    '[-Inf,39)'      186     133    1.3985     -0.36902       0.03815
    '[39,46)'        195     108    1.8056     -0.11355     0.0033158
```

```
'[46,52)'      192      75      2.56      0.23559      0.011823
'[52,Inf]'     230      81     2.8395     0.33921       0.02795
```



If Categorical Data Is Given as Numeric

Categorical data is often recorded using numeric values, and can be stored in a numeric array. Although you know that the data should be interpreted as categorical information, for `creditscorecard` this predictor looks like a numeric array.

To show the case where categorical data is given as numeric data, the data for the variable `ResStatus` is intentionally converted to numeric values.

```
load CreditCardData
data.ResStatus = double(data.ResStatus);
sc = creditscorecard(data,'IDVar','CustID')

sc =

  creditscorecard with properties:
```

```
                GoodLabel: 0
             ResponseVar: 'status'
                VarNames: {1x11 cell}
        NumericPredictors: {1x7 cell}
    CategoricalPredictors: {'EmpStatus'  'OtherCC'}
                  IDVar: 'CustID'
           PredictorVars: {1x9 cell}
```

Note that `'ResStatus'` appears as part of the `NumericPredictors` property. If we applied automatic binning, the resulting bin information raises flags regarding the predictor type.

```
sc = autobinning(sc,'ResStatus');
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi =

      Bin        Good    Bad     Odds        WOE        InfoValue
    _________    ____    ___    ______    _________    _________

    '[-Inf,2)'    365    177    2.0621     0.019329     0.0001682
    '[2,Inf]'     438    220    1.9909    -0.015827    0.00013772
    'Totals'      803    397    2.0227          NaN    0.00030592


cg =

     2
```

The numeric ranges in the bin labels show that `'ResStatus'` is being treated as a numeric variable. This is also confirmed by the fact that the optional output from `bininfo` is a numeric array of cut points, as opposed to a table with category groupings. Moreover, the output from `predictorinfo` confirms that the credit scorecard is treating the data as numeric.

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T =

                PredictorType        LatestBinning
                _____________    ____________________

    ResStatus    'Numeric'        'Automatic / Monotone'


Stats =

            Value
           _______

    Min          1
    Max          3
```

```
    Mean     1.7017
    Std     0.71863
```

Solution: For `creditscorecard`, `'Categorical'` means a MATLAB categorical data type. For more information, see `categorical`. To treat `'ResStatus'` as categorical, change the `'PredictorType'` of the `PredictorName` `'ResStatus'` from `'Numeric'` to `'Categorical'` using `modifypredictor`.

```
sc = modifypredictor(sc,'ResStatus','PredictorType','Categorical')
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
sc =

  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
               VarNames: {1x11 cell}
      NumericPredictors: {1x6 cell}
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                  IDVar: 'CustID'
          PredictorVars: {1x9 cell}


T =

                PredictorType    Ordinal     LatestBinning
                _____________    _______    _______________

    ResStatus    'Categorical'    false      'Original Data'


Stats =

          Count
          _____

    C1    542
    C2    474
    C3    184
```

Note that `'ResStatus'` now appears as part of the Categorical predictors. Also, `predictorinfo` now describes `'ResStatus'` as categorical and displays the category counts.

If you apply `autobinning`, the categories are now reordered, as shown by calling `bininfo`, which also shows the category labels, as opposed to numeric ranges. The optional output of `bininfo` is now a category grouping table.

```
sc = autobinning(sc,'ResStatus');
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi =
```

```
      Bin        Good    Bad     Odds        WOE        InfoValue
    _________    ____    ___    ______    __________    _________

    'C2'         307     167    1.8383    -0.095564     0.0036638
    'C1'         365     177    2.0621     0.019329     0.0001682
    'C3'         131      53    2.4717      0.20049     0.0059418
    'Totals'     803     397    2.0227          NaN     0.0097738

cg =

    Category    BinNumber
    ________    _________

    'C2'        1
    'C1'        2
    'C3'        3
```

NaNs Returned When Scoring a "Test" Dataset

When applying a `creditscorecard` model to a "test" dataset using the `score` function, if an observation in the "test" dataset has a `NaN` or `<undefined>` value, a `NaN` total score is returned for each of these observations. For example, a `creditscorecard` object is created using "training" data.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
sc = autobinning(sc);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized Linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBalance
    Distribution = Binomial

Estimated Coefficients:
                    Estimate       SE       tStat       pValue
                    ________    ________    ______    __________

    (Intercept)     0.70239     0.064001    10.975    5.0538e-28
    CustAge         0.60833      0.24932      2.44      0.014687
    ResStatus         1.377      0.65272    2.1097      0.034888
    EmpStatus       0.88565        0.293    3.0227     0.0025055
    CustIncome      0.70164      0.21844    3.2121     0.0013179
    TmWBank          1.1074      0.23271    4.7589    1.9464e-06
    OtherCC          1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792
```

```
1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Suppose that a missing observation (`Nan`) is added to the data and then `newdata` is scored using the `score` function. By default, the points and score assigned to the missing value is `NaN`.

```
newdata = data(1:10,:);
newdata.CustAge(1) = NaN;
[Scores,Points] = score(sc,newdata)
```

```
Scores =

       NaN
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
   -0.0872
    0.9207
    1.0399
    0.8252


Points =

    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBalance
    _______    _________    _________    __________    _________    ________    _________

        NaN    -0.031252    -0.076317       0.43693       0.39607     0.15842    -0.017472
      0.479      0.12696      0.31449       0.43693      -0.033752    0.15842    -0.017472
    0.21445    -0.031252      0.31449      0.081611       0.39607    -0.19168    -0.017472
    0.23039      0.12696      0.31449       0.43693      -0.044811    0.15842      0.35551
      0.479      0.12696      0.31449       0.43693      -0.044811    0.15842    -0.017472
      0.479      0.12696      0.31449       0.43693       0.39607     0.15842    -0.017472
    -0.14036      0.12696    -0.076317      -0.10466      -0.033752    0.15842    -0.017472
    0.23039      0.37641      0.31449       0.43693      -0.033752    -0.19168     -0.21206
    0.23039    -0.031252    -0.076317       0.43693      -0.033752    0.15842      0.35551
    0.23039      0.12696    -0.076317       0.43693      -0.033752    0.15842    -0.017472
```

Additionally, notice that because the `CustAge` predictor for the first observation is `NaN`, the corresponding `Scores` output is `NaN` also.

Solution: To resolve this issue, use the `formatpoints` function with the name-value pair argument `Missing`. When using `Missing`, you can replace a predictor's `NaN` value according to three alternative criteria (`'ZeroWoe'`, `'MinPoints'`, or `'MaxPoints'`).

For example, use `Missing` to replace the missing value with the `'MinPoints'` option. The row with the missing data now has a score corresponding to assigning it the minimum possible points for `CustAge`.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,newdata)
PointsTable = displaypoints(sc);
PointsTable(1:7,:)
```

```
Scores =

    0.7074
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
   -0.0872
    0.9207
    1.0399
    0.8252


Points =

    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBalance
    ________    _________    _________    __________    _________    ________    _________

    -0.15894    -0.031252    -0.076317     0.43693        0.39607     0.15842    -0.017472
       0.479      0.12696      0.31449     0.43693       -0.033752    0.15842    -0.017472
     0.21445    -0.031252      0.31449    0.081611        0.39607    -0.19168    -0.017472
     0.23039      0.12696      0.31449     0.43693       -0.044811    0.15842      0.35551
       0.479      0.12696      0.31449     0.43693       -0.044811    0.15842    -0.017472
       0.479      0.12696      0.31449     0.43693        0.39607     0.15842    -0.017472
    -0.14036      0.12696    -0.076317    -0.10466       -0.033752    0.15842    -0.017472
     0.23039      0.37641      0.31449     0.43693       -0.033752    -0.19168    -0.21206
     0.23039    -0.031252    -0.076317     0.43693       -0.033752    0.15842      0.35551
     0.23039      0.12696    -0.076317     0.43693       -0.033752    0.15842    -0.017472


ans =

    Predictors        Bin          Points
    __________    ___________    _________

    'CustAge'     '[-Inf,33)'     -0.15894
    'CustAge'     '[33,37)'       -0.14036
    'CustAge'     '[37,40)'      -0.060323
    'CustAge'     '[40,46)'       0.046408
    'CustAge'     '[46,48)'        0.21445
    'CustAge'     '[48,58)'        0.23039
    'CustAge'     '[58,Inf]'         0.479
```

Notice that the `Scores` output has a value for the first customer record because `CustAge` now has a value and the score can be calculated for the first customer record.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Related Examples

- “Case Study for a Credit Scorecard Analysis” on page 8-78

More About

- “About Credit Scorecards” on page 8-57
- “Credit Scorecard Modeling Workflow” on page 8-62

Case Study for a Credit Scorecard Analysis

This example shows how to create a `creditscorecard` object, bin data, display, and plot binned data information. This example also shows how to fit a logistic regression model, obtain a score for the scorecard model, and determine the probabilities of default and validate the credit scorecard model using three different metrics.

Step 1. Create a creditscorecard object.

Use the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). By default, `'ResponseVar'` is set to the last column in the data (`'status'` in this example) and the `'GoodLabel'` to the response value with the highest count (`0` in this example). The syntax for `creditscorecard` indicates that `'CustID'` is the `'IDVar'` to remove from the list of predictors. Also, while not demonstrated in this example, when creating a `creditscorecard` object using `creditscorecard`, you can use the optional name-value pair argument `WeightsVar` to specify observation (sample) weights.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')
```

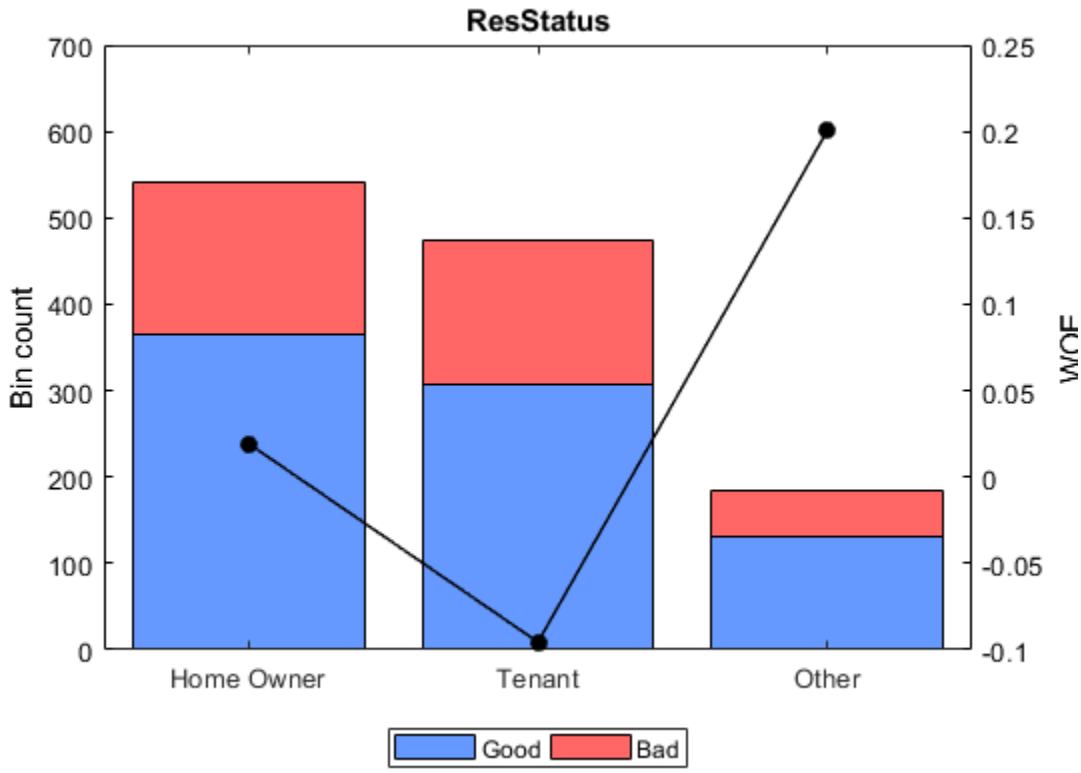
```
sc =

  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                 VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                     Data: [1200x11 table]
```

Perform some initial data exploration. Inquire about predictor statistics for the categorical variable `'ResStatus'` and plot the bin information for `'ResStatus'`.

```
bininfo(sc,'ResStatus')
plotbins(sc,'ResStatus')
```

```
ans =
```

```
4x6 table

      Bin          Good    Bad     Odds        WOE        InfoValue
    ___________    ____    ___    ______    _________    _________

    'Home Owner'    365     177    2.0621     0.019329    0.0001682
    'Tenant'        307     167    1.8383    -0.095564    0.0036638
    'Other'         131      53    2.4717      0.20049    0.0059418
    'Totals'        803     397    2.0227          NaN    0.0097738
```



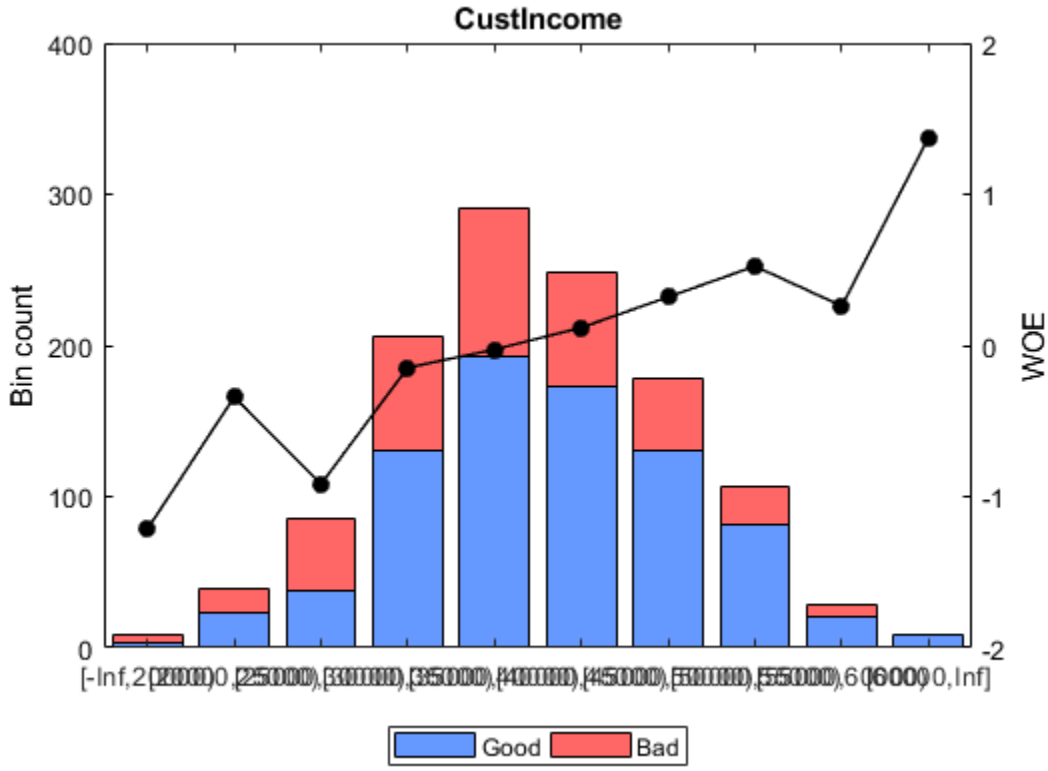
This bin information contains the frequencies of “Good” and “Bad,” and bin statistics. Avoid having bins with frequencies of zero because they lead to infinite or undefined

(NaN) statistics. Use the `modifybins` or `autobinning` functions to bin the data accordingly.

For numeric data, a common first step is "fine classing." This means binning the data into several bins, defined with a regular grid. To illustrate this point, use the predictor `'CustIncome'`.

```
cp = 20000:5000:60000;

sc = modifybins(sc,'CustIncome','CutPoints',cp);

bininfo(sc,'CustIncome')
plotbins(sc,'CustIncome')
```

```
ans =

  11x6 table

         Bin           Good    Bad     Odds        WOE        InfoValue
    _______________    ____    ___    _______    _________    __________

    '[-Inf,20000)'       3       5        0.6      -1.2152      0.010765
    '[20000,25000)'     23      16     1.4375      -0.34151     0.0039819
    '[25000,30000)'     38      47    0.80851      -0.91698      0.065166
    '[30000,35000)'    131      75     1.7467      -0.14671      0.003782
    '[35000,40000)'    193      98     1.9694     -0.026696     0.00017359
    '[40000,45000)'    173      76     2.2763       0.11814     0.0028361
    '[45000,50000)'    131      47     2.7872       0.32063      0.014348
    '[50000,55000)'     82      24     3.4167       0.52425      0.021842
    '[55000,60000)'     21       8      2.625       0.26066     0.0015642
    '[60000,Inf]'        8       1          8         1.375      0.010235
    'Totals'           803     397     2.0227           NaN       0.13469
```



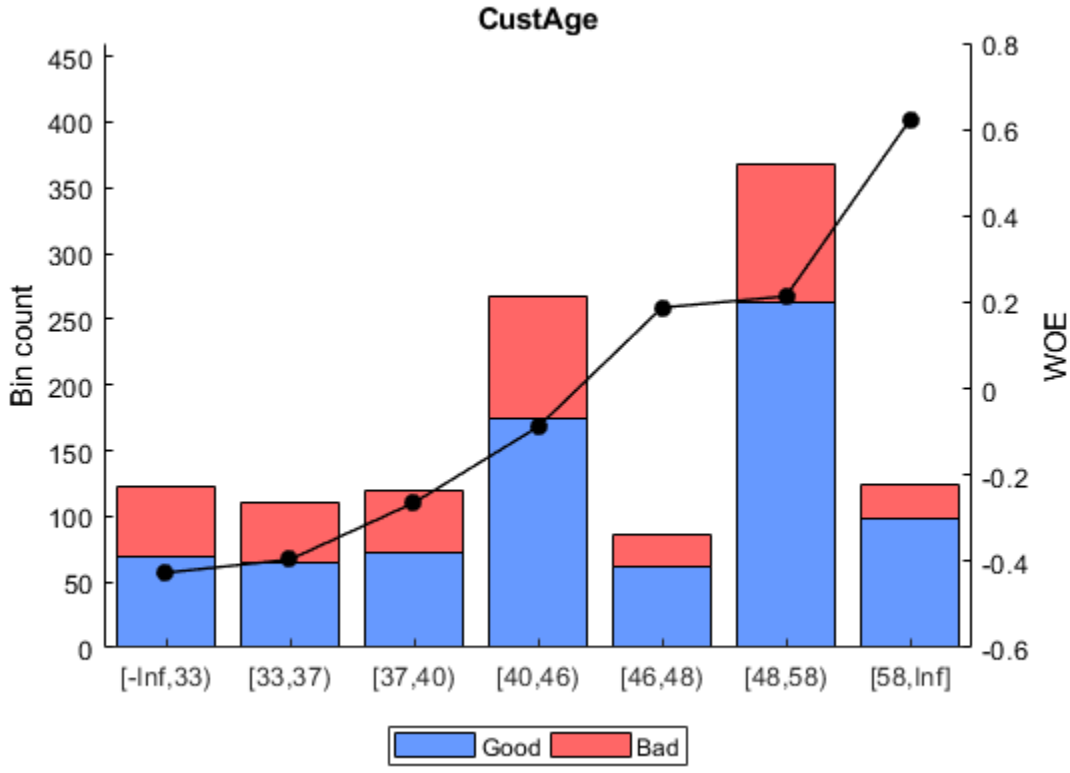
Step 2a. Automatically bin the data.

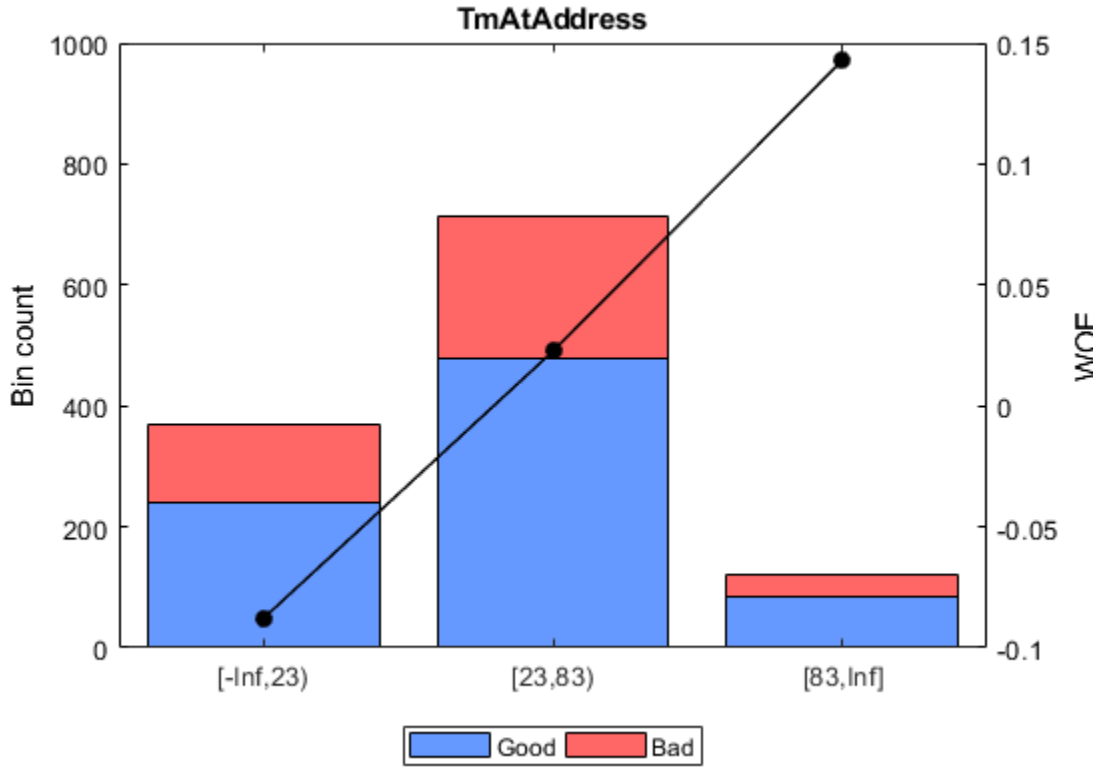
Use the `autobinning` function to perform automatic binning for every predictor variable, using the default `'Monotone'` algorithm with default algorithm options.

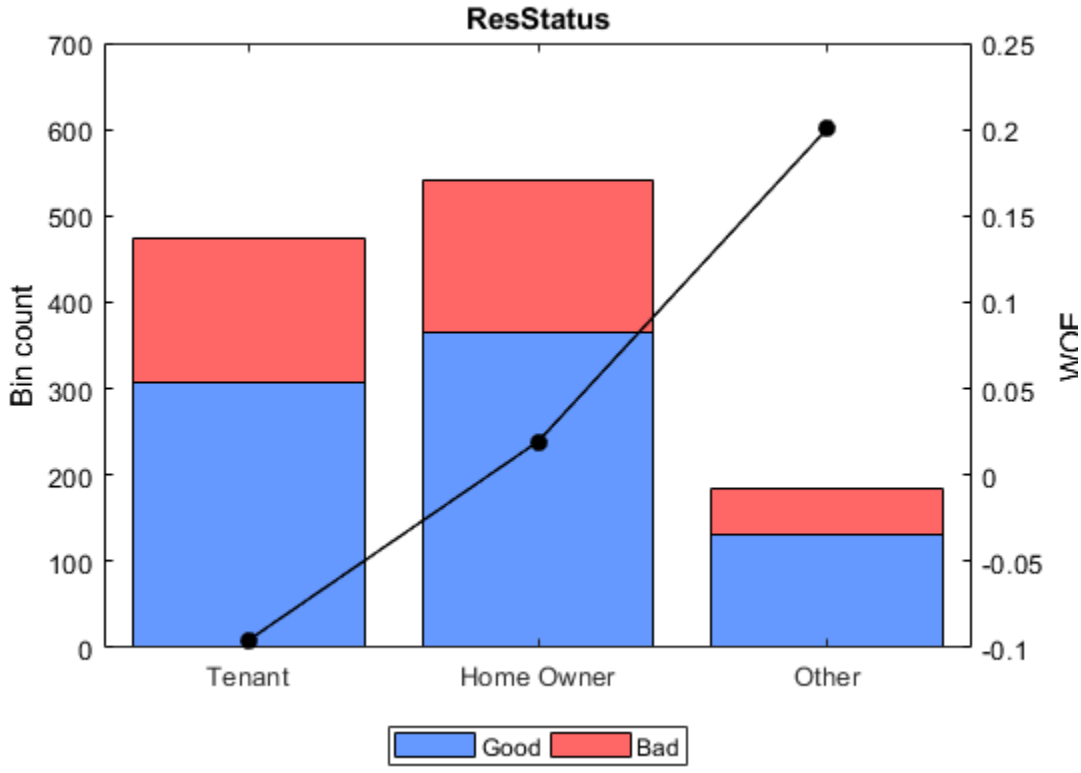
```
sc = autobinning(sc);
```

After the automatic binning step, every predictor bin must be reviewed using the `bininfo` and `plotbins` functions and fine-tuned. A monotonic, ideally linear trend in the Weight of Evidence (WOE) is desirable for credit scorecards because this translates into linear points for a given predictor. The WOE trends can be visualized using `plotbins`.

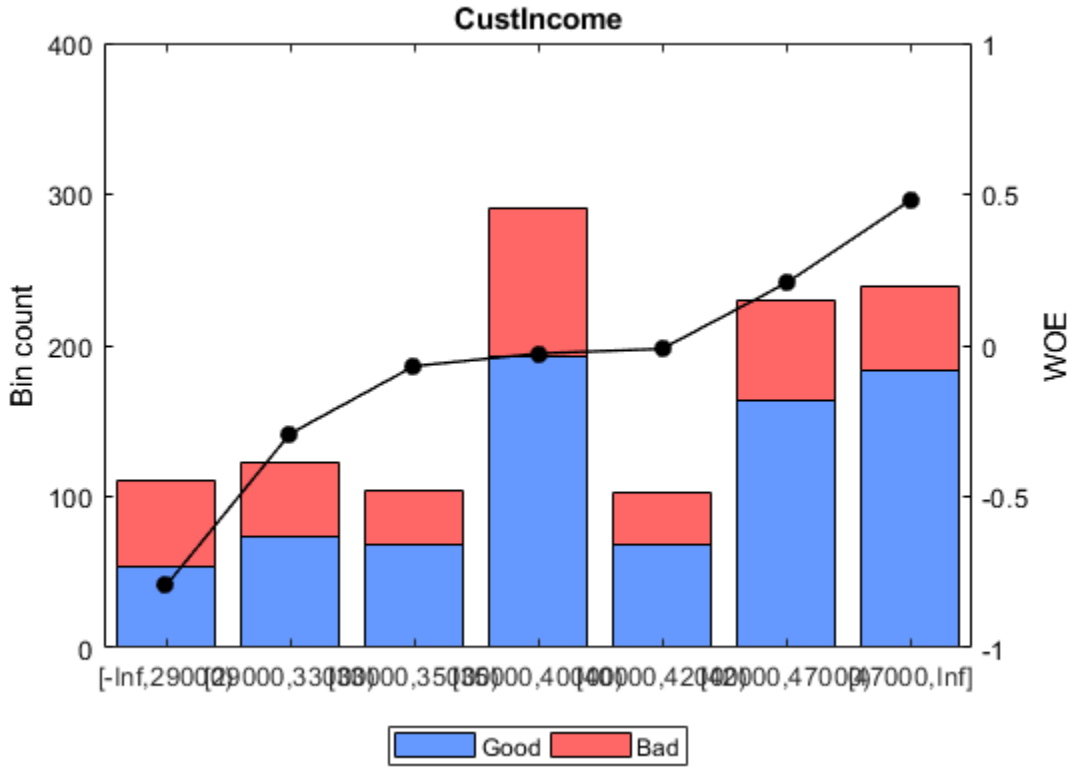
```
plotbins(sc,sc.PredictorVars)
```

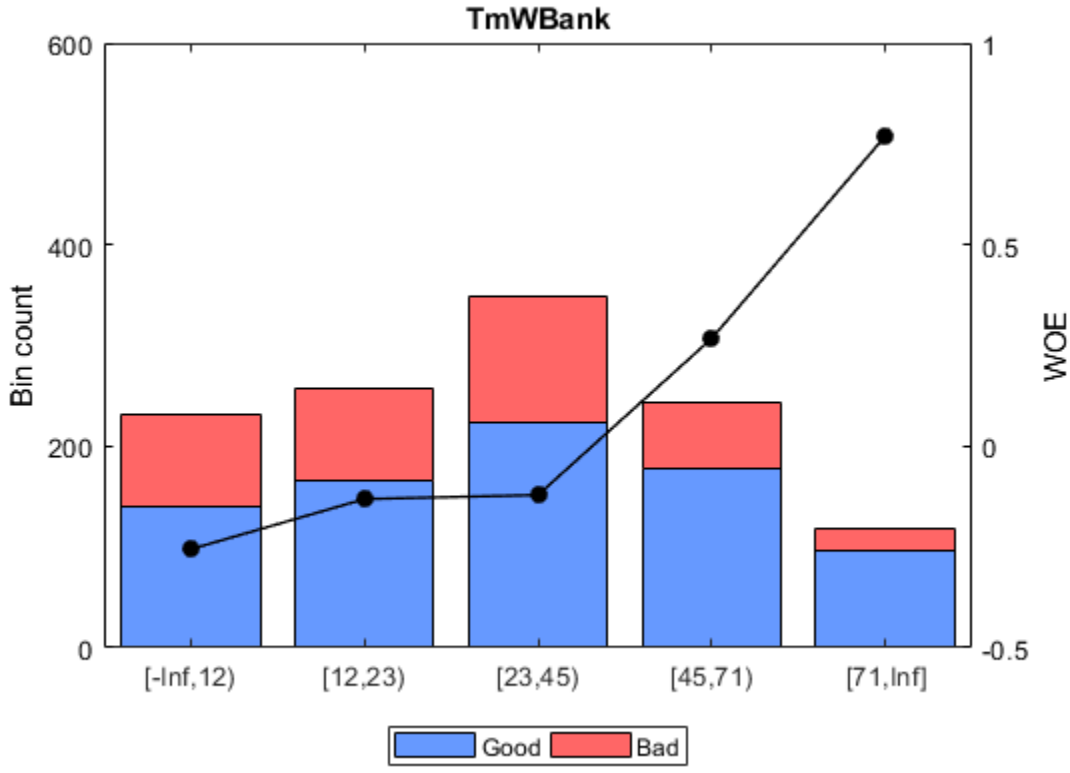

CustAge
Bin count
WOE
450
400
350
300
250
200
150
100
50
0
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
[-Inf,33)
[33,37)
[37,40)
[40,46)
[46,48)
[48,58)
[58,Inf]
Good
Bad

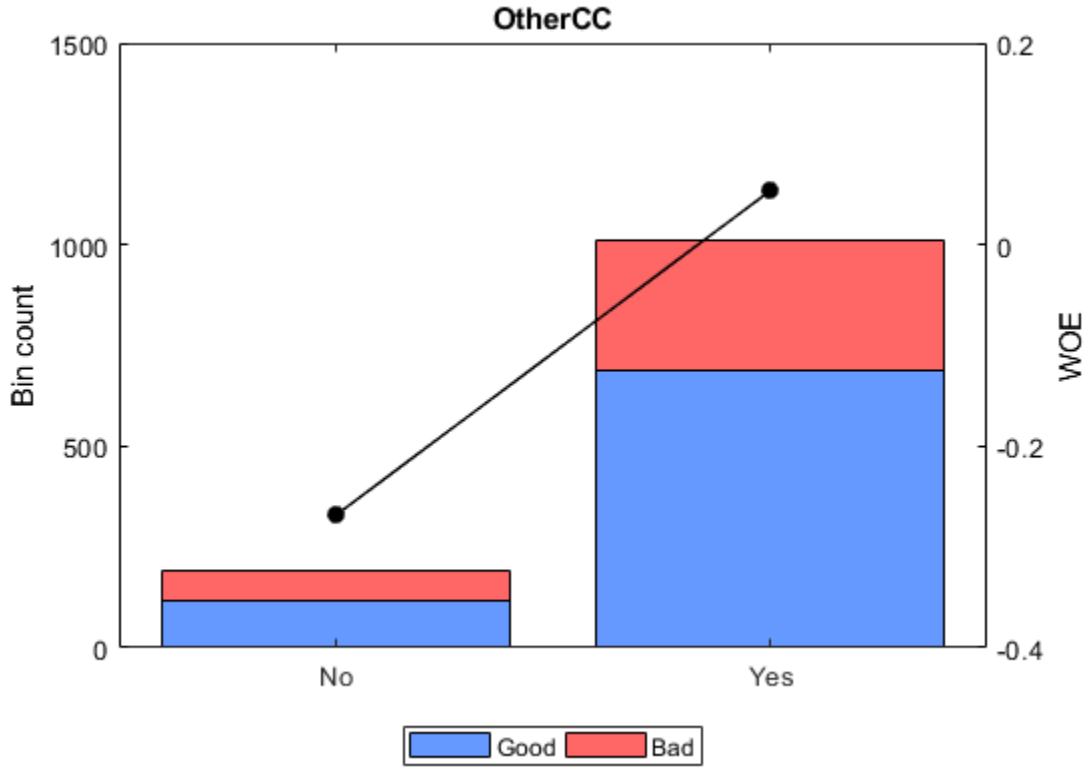

TmAtAddress
Bin count
WOE
1000
800
600
400
200
0
0.15
0.1
0.05
0
-0.05
-0.1
[-Inf,23)
[23,83)
[83,Inf]
Good
Bad

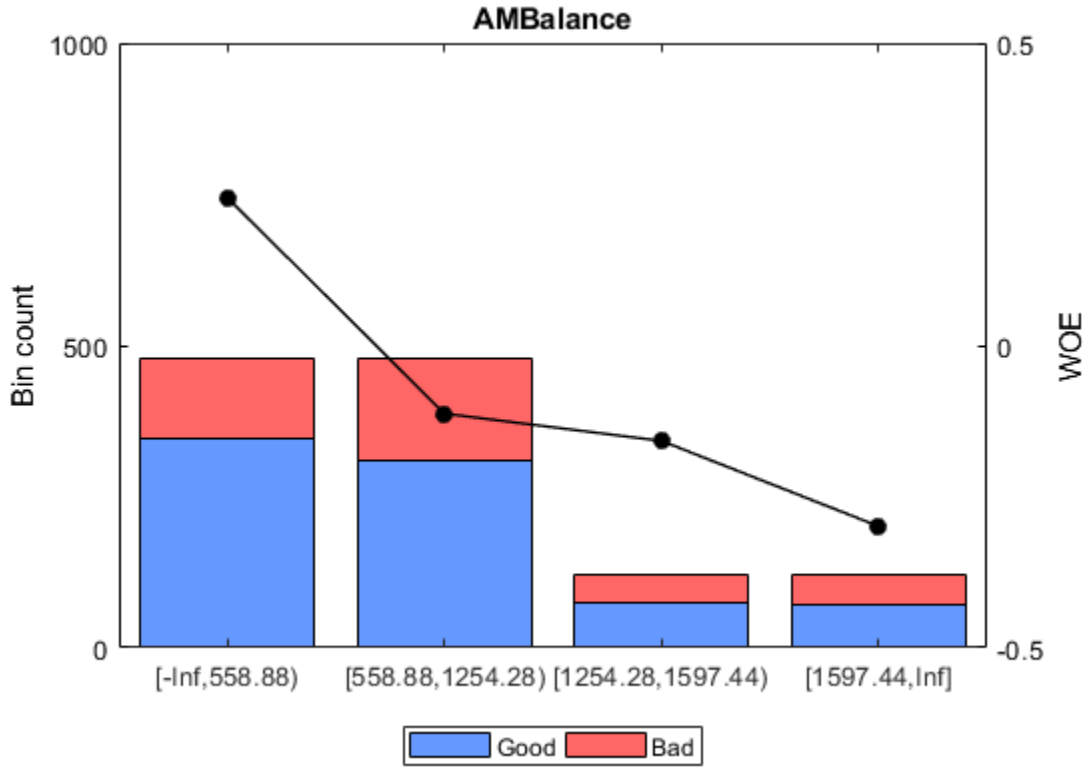

ResStatus
Bin count
WOE
700
600
500
400
300
200
100
0
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
Tenant
Home Owner
Other
Good
Bad

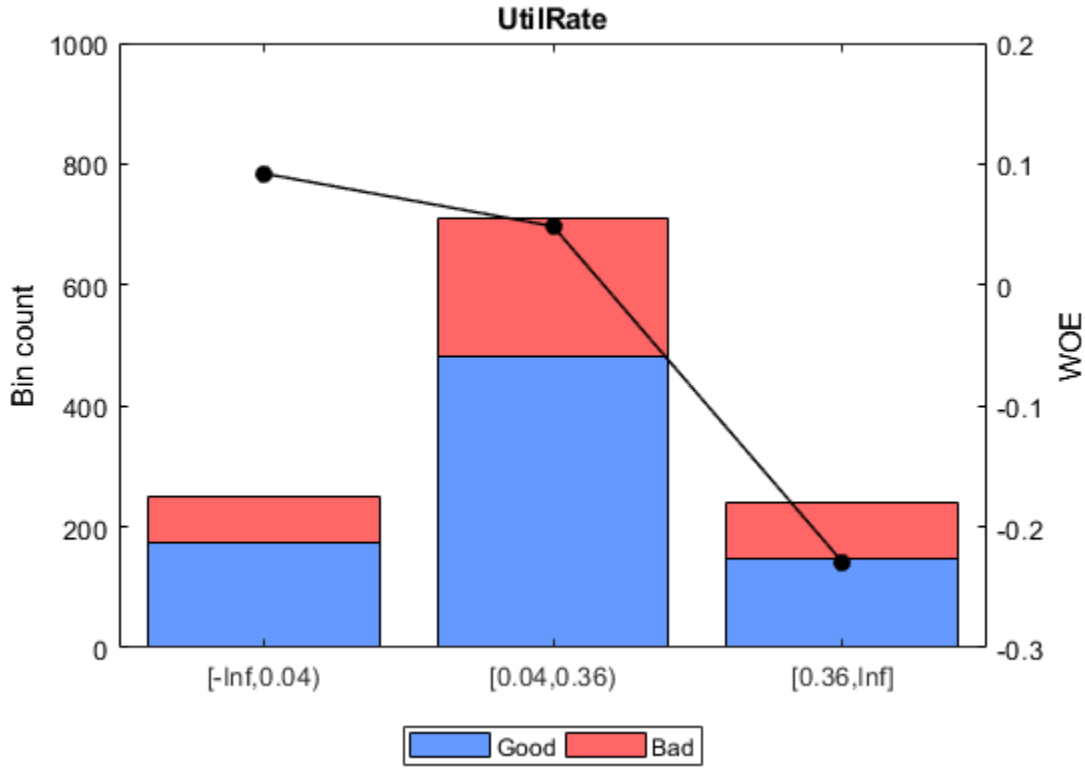

EmpStatus
1000
800
600
400
200
0
Bin count
0.3
0.2
0.1
0
-0.1
-0.2
WOE
Unknown
Employed
Good
Bad


CustIncome
Bin count
WOE
400
300
200
100
0
1
0.5
0
-0.5
-1
[-Inf,29000)
[29000,33000)
[33000,35000)
[35000,40000)
[40000,42000)
[42000,47000)
[47000,Inf]
Good
Bad


TmWBank
600
400
200
0
Bin count
1
0.5
0
-0.5
WOE
[-Inf,12)
[12,23)
[23,45)
[45,71)
[71,Inf]
Good
Bad


OtherCC
1500
1000
500
0
Bin count
0.2
0
-0.2
-0.4
WOE
No
Yes
Good
Bad


AMBalance
1000
500
0
Bin count
0.5
0
-0.5
WOE
[-Inf,558.88)
[558.88,1254.28)
[1254.28,1597.44)
[1597.44,Inf]
Good
Bad



Unlike the initial plot of `'ResStatus'` when the scorecard was created, the new plot for `'ResStatus'` shows an increasing WOE trend. This is because the `autobinning` function, by default, sorts the order of the categories by increasing odds.

These plots show that the `'Monotone'` algorithm does a good job finding monotone WOE trends for this dataset. To complete the binning process, it is necessary to make only a few manual adjustments for some predictors using the `modifybins` function.

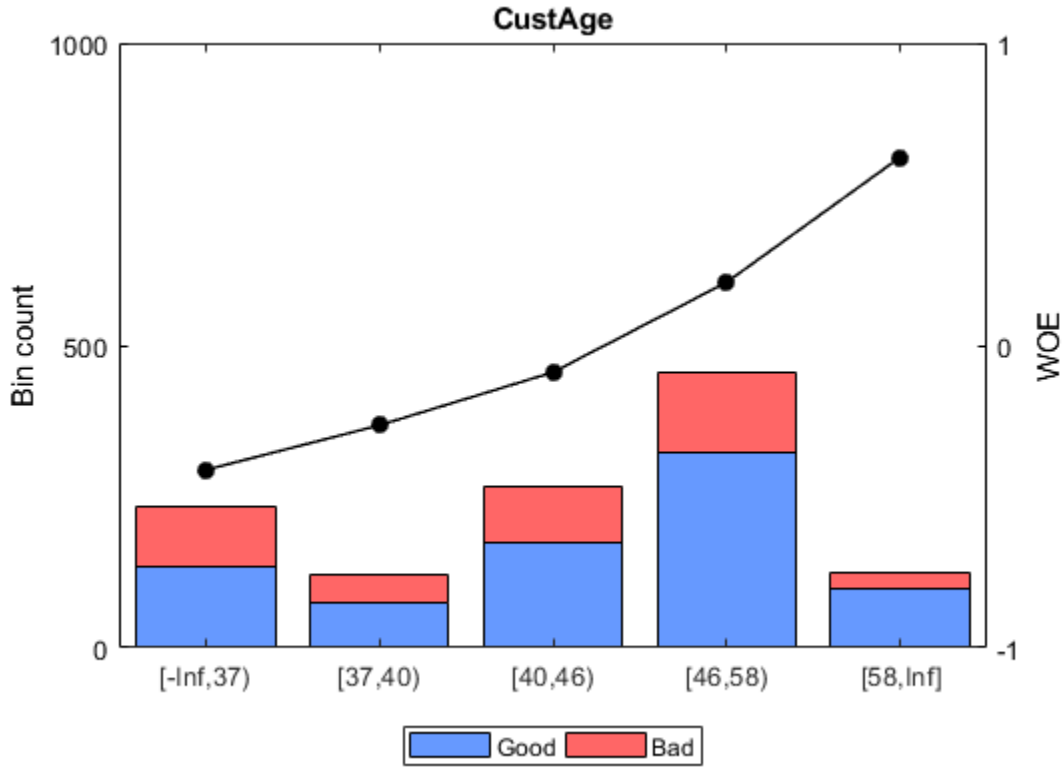
Step 2b. Fine-tune the bins using manual binning.

Common steps to manually modify bins are:

- Use the `bininfo` function with two output arguments where the second argument contains binning rules.
- Manually modify the binning rules using the second output argument from `bininfo`.
- Set the updated binning rules with `modifybins` and then use `plotbins` or `bininfo` to review the updated bins.

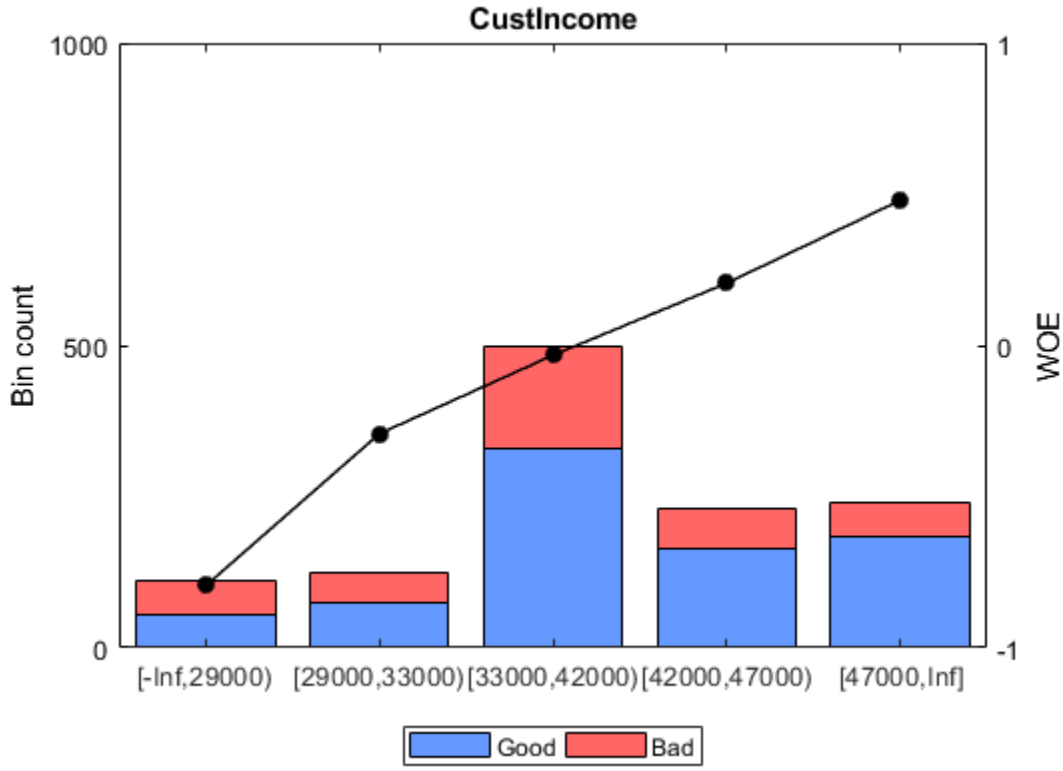
For example, based on the plot for `'CustAge'` in Step 2a, bins number 1 and 2 have similar WOE's as do bins number 5 and 6. To merge these bins using the steps outlined above:

```
[bi,cp] = bininfo(sc,'CustAge');
cp([1 5]) = []; % To merge bins 1 and 2, and bins 5 and 6
sc = modifybins(sc,'CustAge','CutPoints',cp);
plotbins(sc,'CustAge')
```



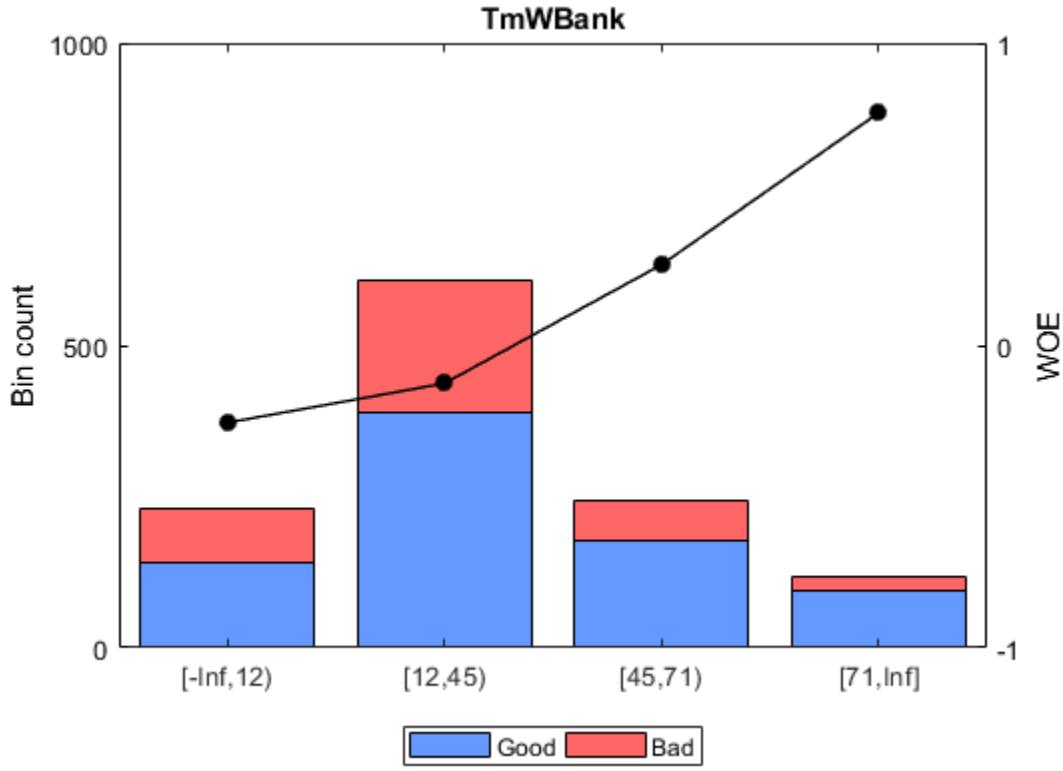
For `'CustIncome'`, based on the plot above, it is best to merge bins 3, 4 and 5 because they have similar WOE's. To merge these bins:

```
[bi,cp] = bininfo(sc,'CustIncome');
cp([3 4]) = [];
sc = modifybins(sc,'CustIncome','CutPoints',cp);
plotbins(sc,'CustIncome')
```



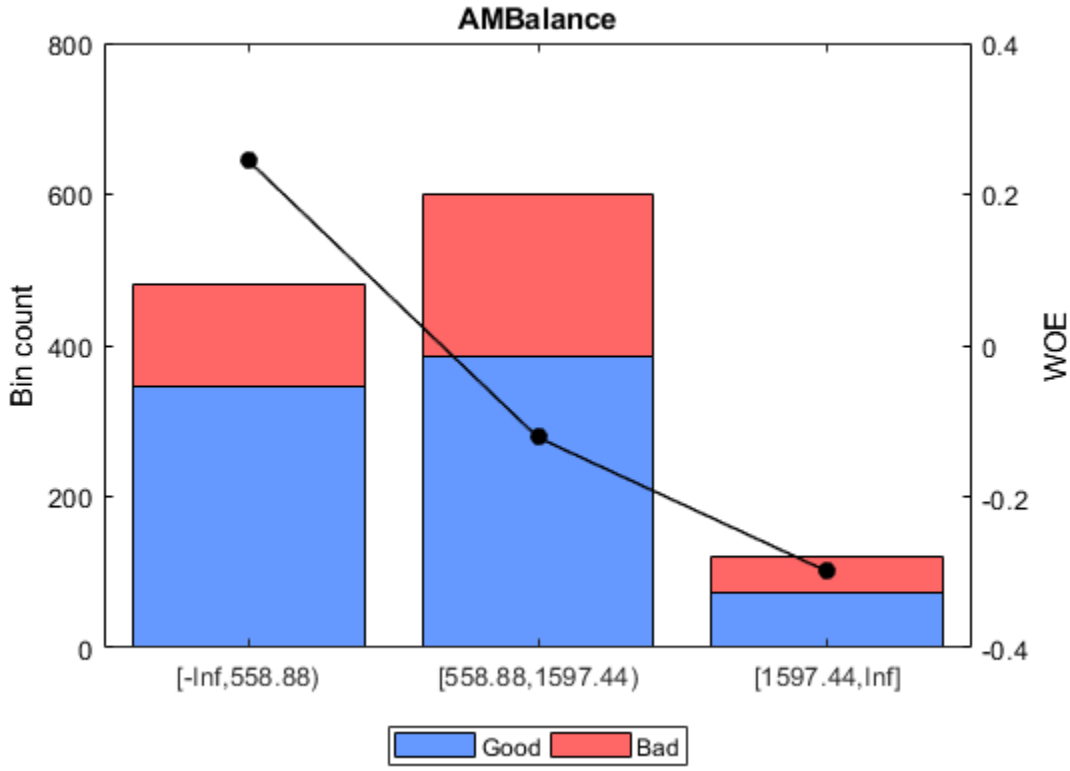
For `'TmWBank'`, based on the plot above, it is best to merge bins 2 and 3 because they have similar WOE's. To merge these bins:

```
[bi,cp] = bininfo(sc,'TmWBank');
cp(2) = [];
sc = modifybins(sc,'TmWBank','CutPoints',cp);
plotbins(sc,'TmWBank')
```



For `'AMBalance'`, based on the plot above, it is best to merge bins 2 and 3 because they have similar WOE's. To merge these bins:

```
[bi,cp] = bininfo(sc,'AMBalance');
cp(2) = [];
sc = modifybins(sc,'AMBalance','CutPoints',cp);
plotbins(sc,'AMBalance')
```



Now that the binning fine-tuning is completed, the bins for all predictors have close-to-linear WOE trends.

Step 3. Fit a logistic regression model.

The `fitmodel` function fits a logistic regression model to the WOE data. `fitmodel` internally bins the training data, transforms it into WOE values, maps the response variable so that `'Good'` is `1`, and fits a linear logistic regression model. By default, `fitmodel` uses a stepwise procedure to determine which predictors should be in the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8954, Chi2Stat = 32.545914, PValue = 1.1640961e-0
2. Adding TmWBank, Deviance = 1467.3249, Chi2Stat = 23.570535, PValue = 1.2041739e-06
3. Adding AMBalance, Deviance = 1455.858, Chi2Stat = 11.466846, PValue = 0.00070848829
4. Adding EmpStatus, Deviance = 1447.6148, Chi2Stat = 8.2432677, PValue = 0.0040903428
5. Adding CustAge, Deviance = 1442.06, Chi2Stat = 5.5547849, PValue = 0.018430237
6. Adding ResStatus, Deviance = 1437.9435, Chi2Stat = 4.1164321, PValue = 0.042468555
7. Adding OtherCC, Deviance = 1433.7372, Chi2Stat = 4.2063597, PValue = 0.040272676

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate      SE       tStat       pValue
                   ________    _______    ______    __________

    (Intercept)     0.7024       0.064    10.975    5.0407e-28
    CustAge        0.61562     0.24783    2.4841      0.012988
    ResStatus       1.3776     0.65266    2.1107      0.034799
    EmpStatus      0.88592     0.29296     3.024     0.0024946
    CustIncome     0.69836     0.21715     3.216     0.0013001
    TmWBank          1.106     0.23266    4.7538    1.9958e-06
    OtherCC         1.0933     0.52911    2.0662      0.038806
    AMBalance       1.0437     0.32292    3.2322     0.0012285


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.42e-16
```

Step 4. Review and format scorecard points.

After fitting the logistic model, by default the points are unscaled and come directly from the combination of WOE values and model coefficients. The `displaypoints` function summarizes the scorecard points.

```
p1 = displaypoints(sc);
disp(p1)
```

```
     Predictors            Bin             Points
    _____________    ___________________    _________

    'CustAge'        '[-Inf,37)'             -0.15314
    'CustAge'        '[37,40)'              -0.062247
    'CustAge'        '[40,46)'               0.045763
```

```
    'CustAge'       '[46,58)'              0.22888
    'CustAge'       '[58,Inf]'             0.48354
    'ResStatus'     'Tenant'              -0.031302
    'ResStatus'     'Home Owner'           0.12697
    'ResStatus'     'Other'                0.37652
    'EmpStatus'     'Unknown'             -0.076369
    'EmpStatus'     'Employed'             0.31456
    'CustIncome'    '[-Inf,29000)'        -0.45455
    'CustIncome'    '[29000,33000)'        -0.1037
    'CustIncome'    '[33000,42000)'       0.077768
    'CustIncome'    '[42000,47000)'        0.24406
    'CustIncome'    '[47000,Inf]'          0.43536
    'TmWBank'       '[-Inf,12)'           -0.18221
    'TmWBank'       '[12,45)'             -0.038279
    'TmWBank'       '[45,71)'              0.39569
    'TmWBank'       '[71,Inf]'             0.95074
    'OtherCC'       'No'                    -0.193
    'OtherCC'       'Yes'                  0.15868
    'AMBalance'     '[-Inf,558.88)'         0.3552
    'AMBalance'     '[558.88,1597.44)'    -0.026797
    'AMBalance'     '[1597.44,Inf]'        -0.21168
```

This is a good time to modify the bin labels, if this is something of interest for cosmetic reasons. To do so, use `modifybins` to change the bin labels.

```
sc = modifybins(sc,'CustAge','BinLabels',...
{'Up to 36' '37 to 39' '40 to 45' '46 to 57' '58 and up'});

sc = modifybins(sc,'CustIncome','BinLabels',...
{'Up to 28999' '29000 to 32999' '33000 to 41999' '42000 to 46999' '47000 and up'});

sc = modifybins(sc,'TmWBank','BinLabels',...
{'Up to 11' '12 to 44' '45 to 70' '71 and up'});

sc = modifybins(sc,'AMBalance','BinLabels',...
{'Up to 558.87' '558.88 to 1597.43' '1597.44 and up'});

p1 = displaypoints(sc);
disp(p1)
```

```
     Predictors             Bin             Points
    _____________    ___________________    _________

    'CustAge'        'Up to 36'              -0.15314
```

```
    'CustAge'       '37 to 39'             -0.062247
    'CustAge'       '40 to 45'              0.045763
    'CustAge'       '46 to 57'                0.22888
    'CustAge'       '58 and up'               0.48354
    'ResStatus'     'Tenant'               -0.031302
    'ResStatus'     'Home Owner'              0.12697
    'ResStatus'     'Other'                   0.37652
    'EmpStatus'     'Unknown'              -0.076369
    'EmpStatus'     'Employed'                0.31456
    'CustIncome'    'Up to 28999'           -0.45455
    'CustIncome'    '29000 to 32999'         -0.1037
    'CustIncome'    '33000 to 41999'        0.077768
    'CustIncome'    '42000 to 46999'          0.24406
    'CustIncome'    '47000 and up'            0.43536
    'TmWBank'       'Up to 11'              -0.18221
    'TmWBank'       '12 to 44'             -0.038279
    'TmWBank'       '45 to 70'                0.39569
    'TmWBank'       '71 and up'               0.95074
    'OtherCC'       'No'                       -0.193
    'OtherCC'       'Yes'                     0.15868
    'AMBalance'     'Up to 558.87'             0.3552
    'AMBalance'     '558.88 to 1597.43'    -0.026797
    'AMBalance'     '1597.44 and up'        -0.21168
```

Points are usually scaled and also often rounded. To do this, use the `formatpoints` function. For example, you can set a target level of points corresponding to a target odds level and also set the required points-to-double-the-odds (PDO).

```
TargetPoints = 500;
TargetOdds = 2;
PDO = 50; % Points to double the odds

sc = formatpoints(sc,'PointsOddsAndPDO',[TargetPoints TargetOdds PDO]);
p2 = displaypoints(sc);
disp(p2)
```

```
     Predictors             Bin            Points
    _____________    ____________________    ______

    'CustAge'       'Up to 36'             53.239
    'CustAge'       '37 to 39'             59.796
    'CustAge'       '40 to 45'             67.587
    'CustAge'       '46 to 57'             80.796
    'CustAge'       '58 and up'            99.166
```

```
    'ResStatus'     'Tenant'               62.028
    'ResStatus'     'Home Owner'           73.445
    'ResStatus'     'Other'                91.446
    'EmpStatus'     'Unknown'              58.777
    'EmpStatus'     'Employed'             86.976
    'CustIncome'    'Up to 28999'          31.497
    'CustIncome'    '29000 to 32999'       56.805
    'CustIncome'    '33000 to 41999'       69.896
    'CustIncome'    '42000 to 46999'       81.891
    'CustIncome'    '47000 and up'          95.69
    'TmWBank'       'Up to 11'             51.142
    'TmWBank'       '12 to 44'             61.524
    'TmWBank'       '45 to 70'             92.829
    'TmWBank'       '71 and up'            132.87
    'OtherCC'       'No'                   50.364
    'OtherCC'       'Yes'                  75.732
    'AMBalance'     'Up to 558.87'         89.908
    'AMBalance'     '558.88 to 1597.43'    62.353
    'AMBalance'     '1597.44 and up'       49.016
```

Step 5. Score the data.

The `score` function computes the scores for the training data. An optional `data` input can also be passed to `score`, for example, validation data. The points per predictor for each customer are provided as an optional output.

```
[Scores,Points] = score(sc);
disp(Scores(1:10))
disp(Points(1:10,:))
```

```
  528.2044
  554.8861
  505.2406
  564.0717
  554.8861
  586.1904
  441.8755
  515.8125
  524.4553
  508.3169
```

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBalance

```
    80.796    62.028    58.777    95.69     92.829    75.732    62.353
    99.166    73.445    86.976    95.69     61.524    75.732    62.353
    80.796    62.028    86.976    69.896    92.829    50.364    62.353
    80.796    73.445    86.976    95.69     61.524    75.732    89.908
    99.166    73.445    86.976    95.69     61.524    75.732    62.353
    99.166    73.445    86.976    95.69     92.829    75.732    62.353
    53.239    73.445    58.777    56.805    61.524    75.732    62.353
    80.796    91.446    86.976    95.69     61.524    50.364    49.016
    80.796    62.028    58.777    95.69     61.524    75.732    89.908
    80.796    73.445    58.777    95.69     61.524    75.732    62.353
```

Step 6. Calculate the probability of default.

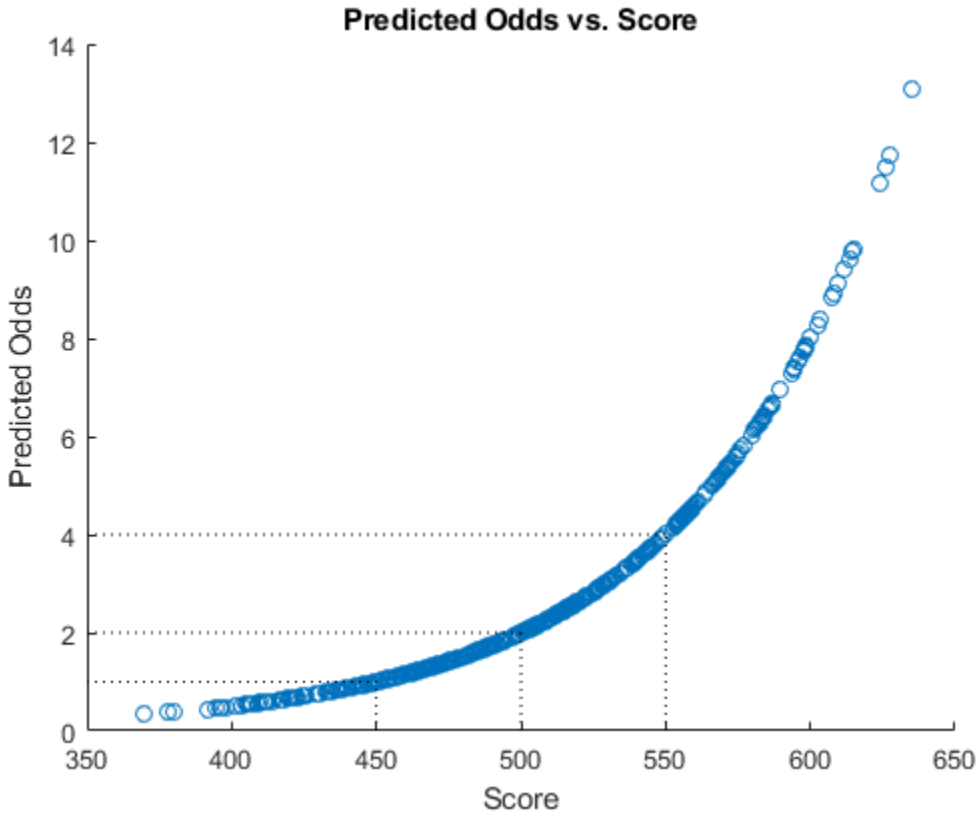
To calculate the probability of default, use the `probdefault` function.

```
pd = probdefault(sc);
```

Define the probability of being "Good" and plot the predicted odds versus the formatted scores. Visually analyze that the target points and target odds match and that the points-to-double-the-odds (PDO) relationship holds.

```
ProbGood = 1-pd;
PredictedOdds = ProbGood./pd;

figure
scatter(Scores,PredictedOdds)
title('Predicted Odds vs. Score')
xlabel('Score')
ylabel('Predicted Odds')

hold on

xLimits = xlim;
yLimits = ylim;

% Target points and odds
plot([TargetPoints TargetPoints],[yLimits(1) TargetOdds],'k:')
plot([xLimits(1) TargetPoints],[TargetOdds TargetOdds],'k:')

% Target points plus PDO
plot([TargetPoints+PDO TargetPoints+PDO],[yLimits(1) 2*TargetOdds],'k:')
plot([xLimits(1) TargetPoints+PDO],[2*TargetOdds 2*TargetOdds],'k:')

% Target points minus PDO
```

```
plot([TargetPoints-PDO TargetPoints-PDO],[yLimits(1) TargetOdds/2],'k:')
plot([xLimits(1) TargetPoints-PDO],[TargetOdds/2 TargetOdds/2],'k:')

hold off
```



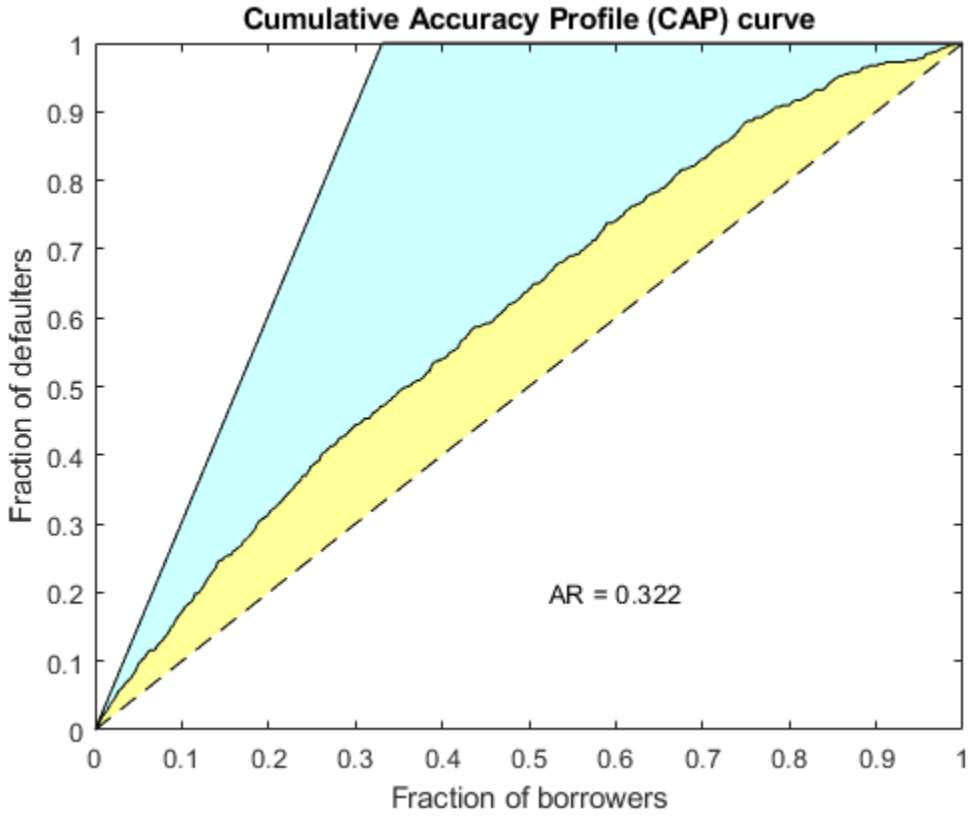
Step 7. Validate the credit scorecard model using the CAP, ROC, and Kolmogorov-Smirnov statistic

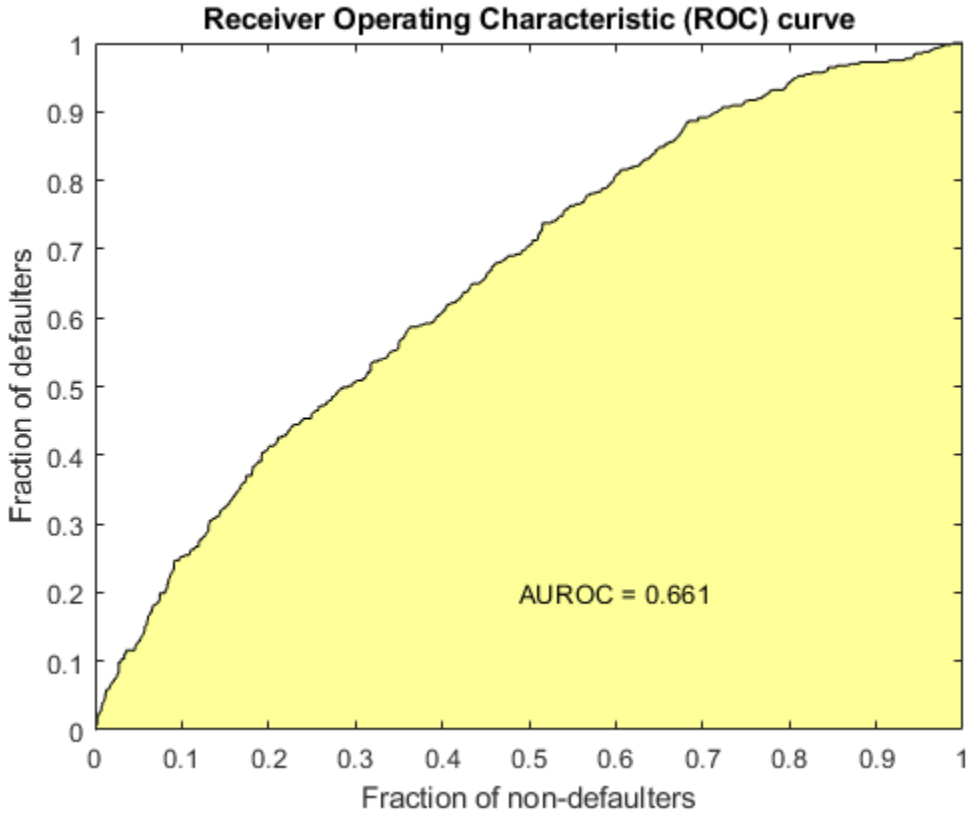
The `creditscorecard` class supports three validation methods, the Cumulative Accuracy Profile (CAP), the Receiver Operating Characteristic (ROC), and the Kolmogorov-Smirnov (K-S) statistic. For more information on CAP, ROC, and KS, see "Cumulative Accuracy Profile (CAP)" on page 18-2202, "Receiver Operating Characteristic (ROC)" on page 18-2203, and "Kolmogorov-Smirnov statistic (KS)" on page 18-2203.

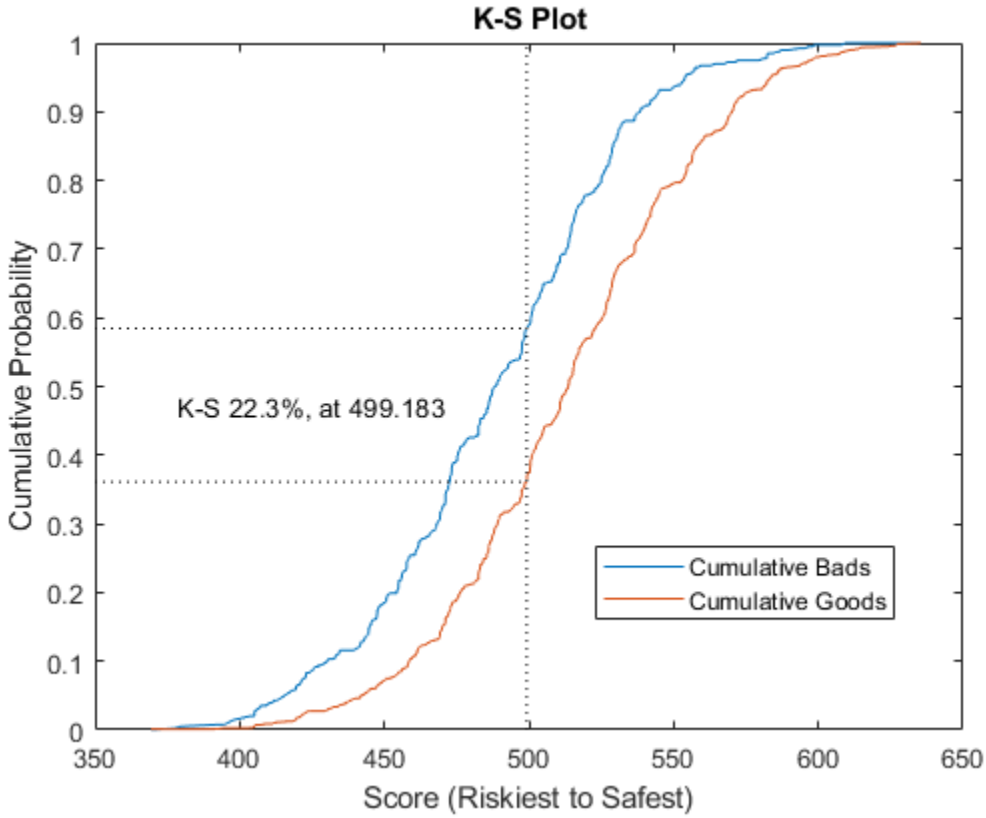
```
[Stats,T] = validatemodel(sc,'Plot',{'CAP','ROC','KS'});
disp(Stats)
disp(T(1:15,:))
```

Measure	Value
'Accuracy Ratio'	0.32225
'Area under ROC curve'	0.66113
'KS statistic'	0.22324
'KS score'	499.18

Scores	ProbDefault	TrueBads	FalseBads	TrueGoods	FalseGoods	Sensit
369.4	0.7535	0	1	802	397	
377.86	0.73107	1	1	802	396	0.0025
379.78	0.7258	2	1	802	395	0.0050
391.81	0.69139	3	1	802	394	0.0075
394.77	0.68259	3	2	801	394	0.0075
395.78	0.67954	4	2	801	393	0.010
396.95	0.67598	5	2	801	392	0.012
398.37	0.67167	6	2	801	391	0.015
401.26	0.66276	7	2	801	390	0.017
403.23	0.65664	8	2	801	389	0.020
405.09	0.65081	8	3	800	389	0.020
405.15	0.65062	11	5	798	386	0.027
405.37	0.64991	11	6	797	386	0.027
406.18	0.64735	12	6	797	385	0.030
407.14	0.64433	13	6	797	384	0.032


Cumulative Accuracy Profile (CAP) curve
Fraction of defaulters
Fraction of borrowers
AR = 0.322
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Receiver Operating Characteristic (ROC) curve
Fraction of defaulters
Fraction of non-defaulters
AUROC = 0.661
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Related Examples

- “Troubleshooting Credit Scorecard Results” on page 8-68
- “Credit Rating by Bagging Decision Trees” (Statistics and Machine Learning Toolbox)

More About

- “About Credit Scorecards” on page 8-57
- “Credit Scorecard Modeling Workflow” on page 8-62
- “Credit Scorecard Modeling Using Observation Weights” on page 8-65
- Monotone Adjacent Pooling Algorithm (MAPA) on page 18-2175

External Websites

- Credit Risk Modeling with MATLAB (53 min 10 sec)

Credit Default Swap (CDS)

A credit default swap (CDS) is a contract that protects against losses resulting from credit defaults. The transaction involves two parties, the protection buyer and the protection seller, and also a reference entity, usually a bond. The protection buyer pays a stream of premiums to the protection seller, who in exchange offers to compensate the buyer for the loss in the bond's value if a credit event occurs. The stream of premiums is called the premium leg, and the compensation when a credit event occurs is called the protection leg. Credit events usually include situations in which the bond issuer goes bankrupt, misses coupon payments, or enters a restructuring process. Financial Instruments Toolbox™ software supports:

CDS Functions

Function	Purpose
`cdsbootstrap`	Compute default probability parameters from CDS market quotes.
`cdsspread`	Compute breakeven spreads for the CDS contracts.
`cdsprice`	Compute the price for the CDS contracts.

See Also

`cdsbootstrap` | `cdsprice` | `cdsrpv01` | `cdsspread`

Related Examples

- "First-to-Default Swaps" (Financial Instruments Toolbox)
- "Credit Default Swap Option" (Financial Instruments Toolbox)
- "Counterparty Credit Risk and CVA" (Financial Instruments Toolbox)

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Bootstrapping a Default Probability Curve

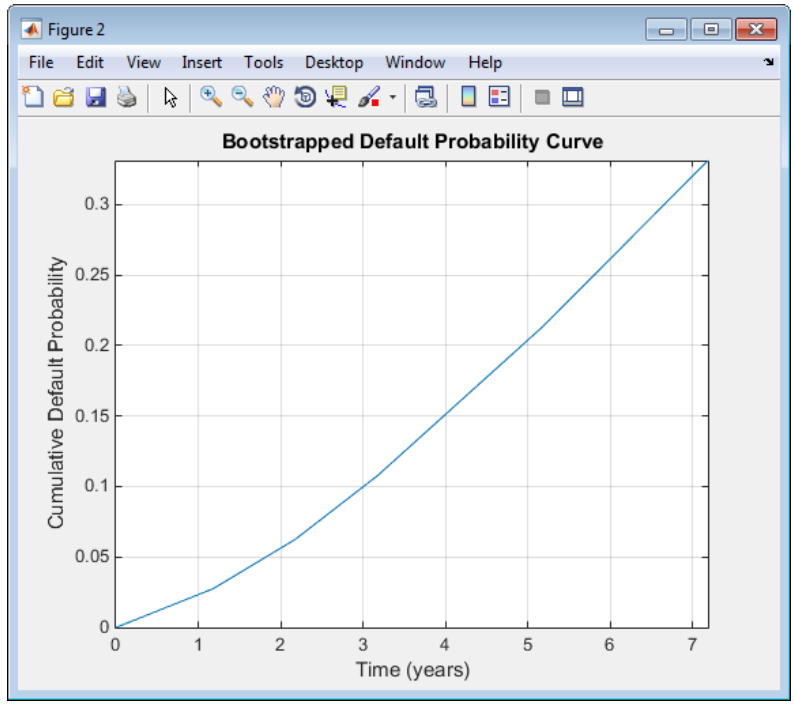
This example shows how to bootstrap default probabilities from CDS market quotes. To bootstrap default probabilities from bond market data, see `bondDefaultBootstrap`. In a typical workflow, pricing a new CDS contract involves first estimating a default probability term structure using `cdsbootstrap`. This requires market quotes of existing CDS contracts, or quotes of CDS indices (e.g., iTraxx). The estimated default probability curve is then used as input to `cdsspread` or `cdsprice`. If the default probability information is already known, `cdsbootstrap` can be bypassed and `cdsspread` or `cdsprice` can be called directly.

The market information in this example is provided in the form of running spreads of CDS contracts maturing on the CDS standard payment dates closest to 1, 2, 3, 5, and 7 years from the valuation date.

```
Settle = '17-Jul-2009'; % valuation date for the CDS
MarketDates = datenum({'20-Sep-10','20-Sep-11','20-Sep-12','20-Sep-14',...
'20-Sep-16'});
MarketSpreads = [140 175 210 265 310]';
MarketData = [MarketDates MarketSpreads];
ZeroDates = datenum({'17-Jan-10','17-Jul-10','17-Jul-11','17-Jul-12',...
'17-Jul-13','17-Jul-14'});
ZeroRates = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
ZeroData = [ZeroDates ZeroRates];

[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle);
```

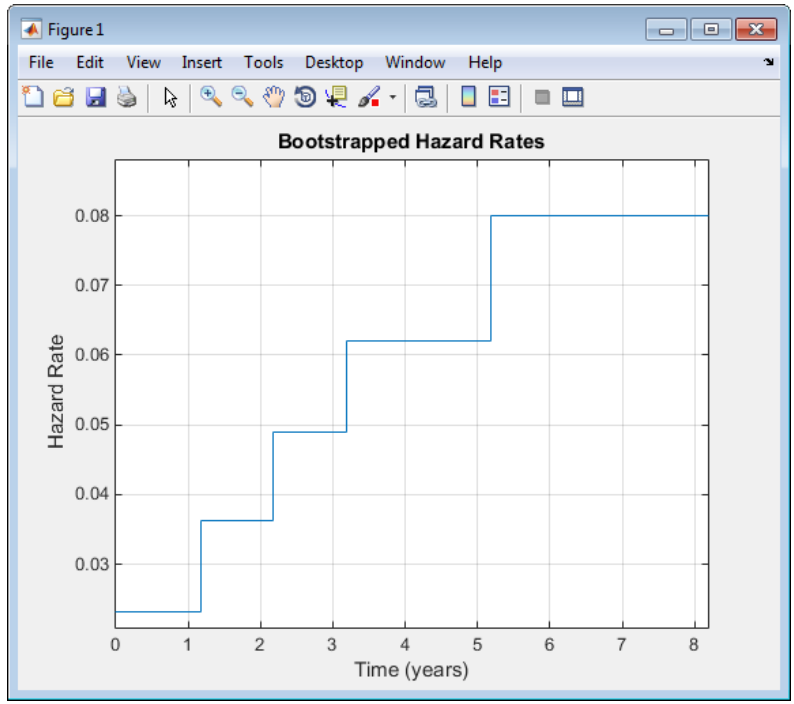
The bootstrapped default probability curve is plotted against time, in years, from the valuation date.

```
ProbTimes = yearfrac(Settle,ProbData(:,1));
figure
plot([0; ProbTimes],[0; ProbData(:,2)])
grid on
axis([0 ProbTimes(end,1) 0 ProbData(end,2)])
xlabel('Time (years)')
ylabel('Cumulative Default Probability')
title('Bootstrapped Default Probability Curve')
```



The associated hazard rates are returned as an optional output. The convention is that the first hazard rate applies from the settlement date to the first market date, the second hazard rate from the first to the second market date, etc., and the last hazard rate applies from the second-to-last market date onwards. The following plot displays the bootstrapped hazard rates, plotted against time, in years, from the valuation date:

```
HazTimes = yearfrac(Settle,HazData(:,1));
figure
stairs([0; HazTimes(1:end-1,1); HazTimes(end,1)+1],...
[HazData(:,2);HazData(end,2)])
grid on
axis([0 HazTimes(end,1)+1 0.9*HazData(1,2) 1.1*HazData(end,2)])
xlabel('Time (years)')
ylabel('Hazard Rate')
title('Bootstrapped Hazard Rates')
```



See Also

`bondDefaultBootstrap` | `cdsbootstrap` | `cdsprice` | `cdsrpv01` | `cdsspread`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Finding Breakeven Spread for New CDS Contract

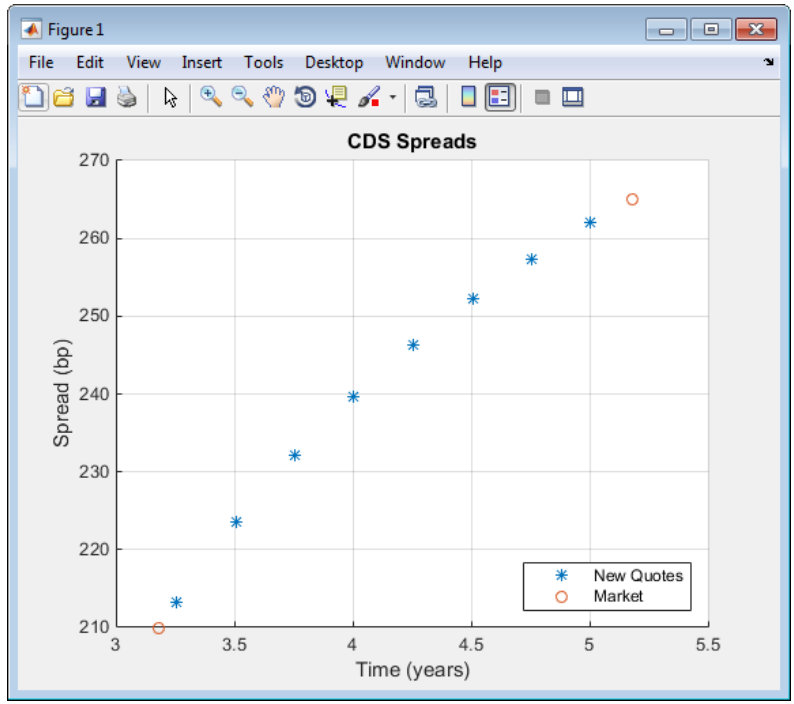
The breakeven, or running, spread is the premium a protection buyer must pay, with no upfront payments involved, to receive protection for credit events associated to a given reference entity. Spreads are expressed in basis points (bp). There is a notional amount associated to the CDS contract to determine the monetary amounts of the premium payments.

New quotes for CDS contracts can be obtained with `cdsspread`. After obtaining a default probability curve using `cdsbootstrap`, you get quotes that are consistent with current market conditions.

In this example, instead of standard CDS payment dates, define new maturity dates. Using the period from 3 and 5 years (CDS standard dates), maturities are defined within this range spaced at quarterly intervals (measuring time from the valuation date):

```
Settle = '17-Jul-2009';  % valuation date for the CDS
MarketDates = datenum({'20-Sep-10','20-Sep-11','20-Sep-12','20-Sep-14',...
'20-Sep-16'});
MarketSpreads = [140 175 210 265 310]';
MarketData = [MarketDates MarketSpreads];
ZeroDates = datenum({'17-Jan-10','17-Jul-10','17-Jul-11','17-Jul-12',...
'17-Jul-13','17-Jul-14'});
ZeroRates = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
ZeroData = [ZeroDates ZeroRates];

[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle);

Maturity1 = datestr(daysadd('17-Jul-09',360*(3.25:0.25:5),1));
Spread1 = cdsspread(ZeroData,ProbData,Settle,Maturity1);

figure
scatter(yearfrac(Settle,Maturity1),Spread1,'*')
hold on
scatter(yearfrac(Settle,MarketData(3:4,1)),MarketData(3:4,2))
hold off
grid on
xlabel('Time (years)')
ylabel('Spread (bp)')
title('CDS Spreads')
legend('New Quotes','Market','location','SouthEast')
```

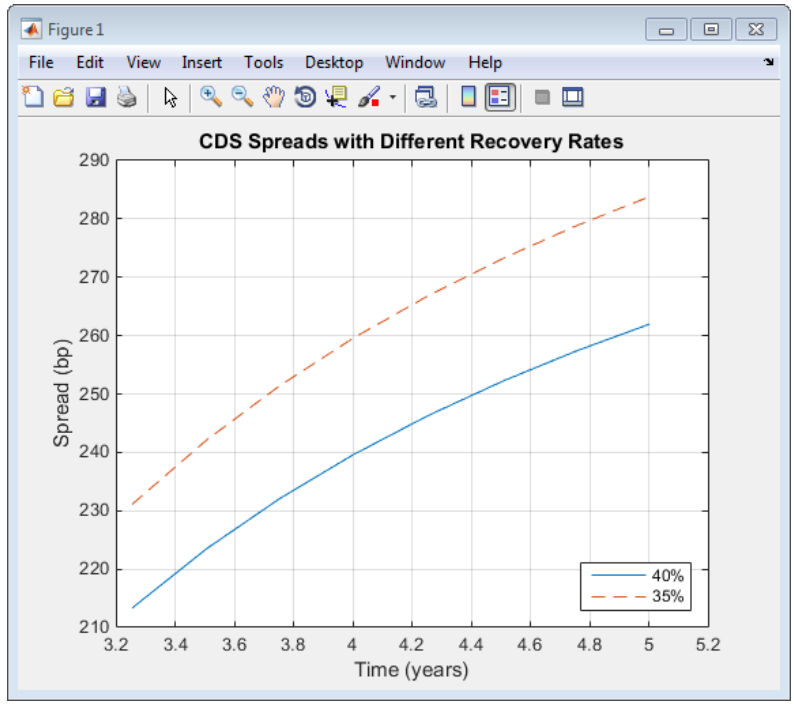
This plot displays the resulting spreads:



To evaluate the effect of the recovery rate parameter, instead of 40% (default value), use a recovery rate of 35%:

```
Spread1Rec35 = cdsspread(ZeroData,ProbData,Settle,Maturity1,...
'RecoveryRate',0.35);

figure
plot(yearfrac(Settle,Maturity1),Spread1,...
yearfrac(Settle,Maturity1),Spread1Rec35,'--')
grid on
xlabel('Time (years)')
ylabel('Spread (bp)')
title('CDS Spreads with Different Recovery Rates')
legend('40%','35%','location','SouthEast')
```

The resulting plot shows that smaller recovery rates produce higher premia, as expected, since in the event of default, the protection payments are higher:



See Also

`cdsbootstrap` | `cdsprice` | `cdsrpv01` | `cdsspread`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Valuing an Existing CDS Contract

The current value, or mark-to-market, of an existing CDS contract is the amount of money the contract holder would receive (if positive) or pay (if negative) to unwind this contract. The upfront of the contract is the current value expressed as a fraction of the notional amount of the contract, and it is commonly used to quote market values.

The value of existing CDS contracts is obtained with `cdsprice`. By default, `cdsprice` treats contracts as long positions. Whether a contract position is long or short is determined from a protection standpoint, that is, long means that protection was bought, and short means protection was sold. In the following example, an existing CDS contract pays a premium that is lower than current market conditions. The price is positive, as expected, since it would be more costly to buy the same type of protection today.

```
Settle = '17-Jul-2009';   % valuation date for the CDS
MarketDates = datenum({'20-Sep-10','20-Sep-11','20-Sep-12','20-Sep-14',...
'20-Sep-16'});
MarketSpreads = [140 175 210 265 310]';
MarketData = [MarketDates MarketSpreads];

ZeroDates = datenum({'17-Jan-10','17-Jul-10','17-Jul-11','17-Jul-12',...
'17-Jul-13','17-Jul-14'});
ZeroRates = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
ZeroData = [ZeroDates ZeroRates];

[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle);

Maturity2 = '20-Sep-2012';
Spread2 = 196;

[Price,AccPrem,PaymentDates,PaymentTimes,PaymentCF] = cdsprice(ZeroData,...
ProbData,Settle,Maturity2,Spread2);

fprintf('Dirty Price: %8.2f\n',Price);
fprintf('Accrued Premium: %8.2f\n',AccPrem);
fprintf('Clean Price: %8.2f\n',Price-AccPrem);
fprintf('\nPayment Schedule:\n\n');
fprintf('Date \t\t Time Frac \t Amount\n');
for k = 1:length(PaymentDates)
   fprintf('%s \t %5.4f \t %8.2f\n',datestr(PaymentDates(k)),...
      PaymentTimes(k),PaymentCF(k));
end
```

This resulting payment schedule is:

```
Dirty Price: 41628.50
Accrued Premium: 15244.44
Clean Price: 26384.05
```

```
Payment Schedule:

Date          Time Frac      Amount
20-Sep-2009      0.1806      35388.89
20-Dec-2009      0.2528      49544.44
20-Mar-2010      0.2500      49000.00
20-Jun-2010      0.2556      50088.89
20-Sep-2010      0.2556      50088.89
20-Dec-2010      0.2528      49544.44
20-Mar-2011      0.2500      49000.00
20-Jun-2011      0.2556      50088.89
20-Sep-2011      0.2556      50088.89
20-Dec-2011      0.2528      49544.44
20-Mar-2012      0.2528      49544.44
20-Jun-2012      0.2556      50088.89
20-Sep-2012      0.2556      50088.89
```

Additionally, you can use `cdsprice` to value a portfolio of CDS contracts. In the following example, a simple hedged position with two vanilla CDS contracts, one long, one short, with slightly different spreads is priced in a single call and the value of the portfolio is the sum of the returned prices:

```
[Price2,AccPrem2] = cdsprice(ZeroData,ProbData,Settle,...
repmat(datenum(Maturity2),2,1),[Spread2;Spread2+3],...
'Notional',[1e7; -1e7]);

fprintf('Contract \t Dirty Price \t Acc Premium \t  Clean Price\n');
fprintf('    Long \t $ %9.2f \t $ %9.2f \t $ %9.2f \t\n',...
   Price2(1), AccPrem2(1), Price2(1) - AccPrem2(1));
fprintf('   Short \t $ %8.2f \t $ %8.2f \t $ %8.2f \t\n',...
   Price2(2), AccPrem2(2), Price2(2) - AccPrem2(2));
fprintf('Mark-to-market of hedged position: $ %8.2f\n',sum(Price2));
```

This resulting value of the portfolio is:

```
Contract      Dirty Price      Acc Premium       Clean Price
    Long      $  41628.50      $  15244.44      $  26384.05
   Short      $ -32708.11      $ -15477.78      $ -17230.33
Mark-to-market of hedged position: $  8920.39
```

See Also

`cdsbootstrap` | `cdsprice` | `cdsrpv01` | `cdsspread`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Converting from Running to Upfront

A CDS market quote is given in terms of a standard spread (usually 100 bp or 500 bp) and an upfront payment, or in terms of an equivalent running or breakeven spread, with no upfront payment. The functions `cdsbootstrap`, `cdsspread`, and `cdsprice` perform upfront to running or running to upfront conversions.

For example, to convert the market quotes to upfront quotes for a standard spread of 100 bp:

```
Settle = '17-Jul-2009';  % valuation date for the CDS
MarketDates = datenum({'20-Sep-10','20-Sep-11','20-Sep-12','20-Sep-14',...
'20-Sep-16'});
MarketSpreads = [140 175 210 265 310]';
MarketData = [MarketDates MarketSpreads];

ZeroDates = datenum({'17-Jan-10','17-Jul-10','17-Jul-11','17-Jul-12',...
'17-Jul-13','17-Jul-14'});
ZeroRates = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
ZeroData = [ZeroDates ZeroRates];

[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle);

Maturity3 = MarketData(:,1);
Spread3Run = MarketData(:,2);
Spread3Std = 100*ones(size(Maturity3));
Price3 = cdsprice(ZeroData,ProbData,Settle,Maturity3,Spread3Std);
Upfront3 = Price3/10000000; % Standard notional of 10MM
display(Upfront3);
```

This resulting value is:

```
Upfront3 =

    0.0047
    0.0158
    0.0327
    0.0737
    0.1182
```

The conversion can be reversed to convert upfront quotes to market quotes:

```
ProbData3Upf = cdsbootstrap(ZeroData,[Maturity3 Upfront3 Spread3Std],Settle);
Spread3RunFromUpf = cdsspread(ZeroData,ProbData3Upf,Settle,Maturity3);
display([Spread3Run Spread3RunFromUpf]);
```

Comparing the results of this conversion to the original market spread demonstrates the reversal:

```
ans =

  140.0000  140.0000
  175.0000  175.0000
  210.0000  210.0000
  265.0000  265.0000
  310.0000  310.0000
```

Under the flat-hazard rate (FHR) quoting convention, a single market quote is used to calibrate a probability curve. This convention yields a single point in the probability curve, and a single hazard rate value. For example, assume a 4-year (standard dates) CDS contract with a current FHR-based running spread of 550 bp needs conversion to a CDS contract with a standard spread of 500 bp:

```
Maturity4 = datenum('20-Sep-13');
Spread4Run = 550;
ProbData4Run = cdsbootstrap(ZeroData,[Maturity4 Spread4Run],Settle);
Spread4Std = 500;
Price4 = cdsprice(ZeroData,ProbData4Run,Settle,Maturity4,Spread4Std);
Upfront4 = Price4/10000000;
fprintf('A running spread of %5.2f is equivalent to\n',Spread4Run);
fprintf('   a standard spread of %5.2f with an upfront of %8.7f\n',...
   Spread4Std,Upfront4);

A running spread of 550.00 is equivalent to
   a standard spread of 500.00 with an upfront of 0.0167576
```

To reverse the conversion:

```
ProbData4Upf = cdsbootstrap(ZeroData,[Maturity4 Upfront4 Spread4Std],Settle);
Spread4RunFromUpf = cdsspread(ZeroData,ProbData4Upf,Settle,Maturity4);
fprintf('A standard spread of %5.2f with an upfront of %8.7f\n',...
   Spread4Std,Upfront4);
fprintf('    is equivalent to a running spread of %5.2f\n',Spread4RunFromUpf);

A standard spread of 500.00 with an upfront of 0.0167576
    is equivalent to a running spread of 550.00
```

As discussed in Beumee et. al., 2009 (see “Credit Derivatives” on page A-7), the FHR approach is a quoting convention only, and leads to quotes inconsistent with market data. For example, calculating the upfront for the 3-year (standard dates) CDS contract with a standard spread of 100 bp using the FHR approach and comparing the results to the upfront amounts previously calculated, demonstrates that the FHR-based approach yields a different upfront amount:

```
Maturity5 = MarketData(3,1);
Spread5Run = MarketData(3,2);
ProbData5Run = cdsbootstrap(ZeroData,[Maturity5 Spread5Run],Settle);
```

```
Spread5Std = 100;
Price5 = cdsprice(ZeroData,ProbData5Run,Settle,Maturity5,Spread5Std);
Upfront5 = Price5/10000000;
fprintf('Relative error of FHR-based upfront amount: %3.1f%%\n',...
   ((Upfront5-Upfront3(3))/Upfront3(3))*100);

Relative error of FHR-based upfront amount: -0.8%
```

See Also

cdsbootstrap | cdsprice | cdsrpv01 | cdsspread

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Bootstrapping from Inverted Market Curves

The following two examples demonstrate the behavior of bootstrapping with inverted CDS market curves, that is, market quotes with higher spreads for short-term CDS contracts. The first example is handled normally by `cdsbootstrap`:

```
Settle = '17-Jul-2009';  % valuation date for the CDS
MarketDates = datenum({'20-Sep-10','20-Sep-11','20-Sep-12','20-Sep-14',...
'20-Sep-16'});

ZeroDates = datenum({'17-Jan-10','17-Jul-10','17-Jul-11','17-Jul-12',...
'17-Jul-13','17-Jul-14'});
ZeroRates = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
ZeroData = [ZeroDates ZeroRates];

MarketSpreadsInv1 = [750 650 550 500 450]';
MarketDataInv1 = [MarketDates MarketSpreadsInv1];
[ProbDataInv1,HazDataInv1] = cdsbootstrap(ZeroData,MarketDataInv1,Settle)

ProbDataInv1 =

   1.0e+05 *

    7.3440    0.0000
    7.3477    0.0000
    7.3513    0.0000
    7.3586    0.0000
    7.3659    0.0000


HazDataInv1 =

   1.0e+05 *

    7.3440    0.0000
    7.3477    0.0000
    7.3513    0.0000
    7.3586    0.0000
    7.3659    0.0000
```

In the second example, `cdsbootstrap` generates a warning:

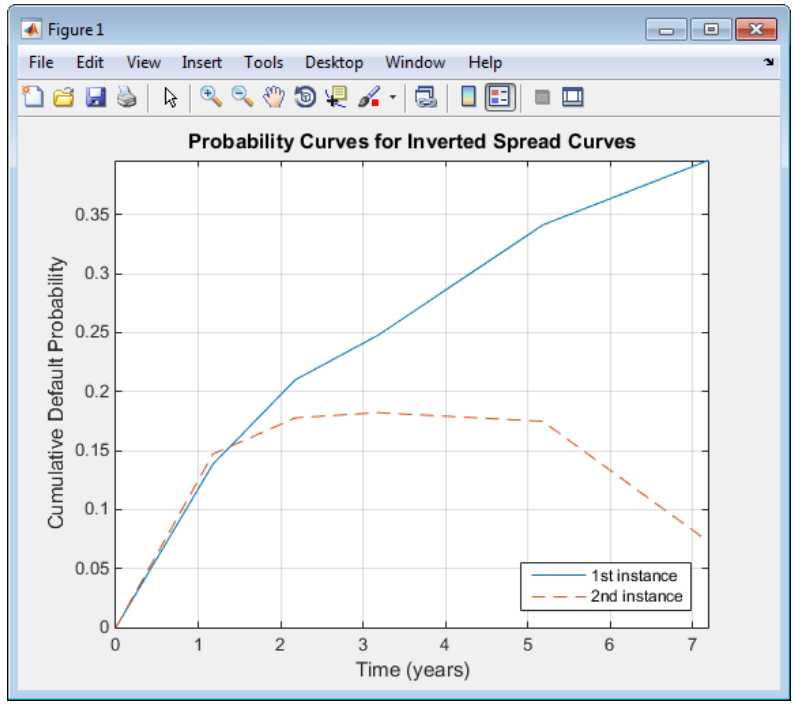
```
MarketSpreadsInv2 = [800 550 400 250 100]';
MarketDataInv2 = [MarketDates MarketSpreadsInv2];

[ProbDataInv2,HazDataInv2] = cdsbootstrap(ZeroData,MarketDataInv2,Settle);

Warning: Found non-monotone default probabilities (negative hazard rates)
```

A non-monotone bootstrapped probability curve implies negative default probabilities and negative hazard rates for certain time intervals. Extreme market conditions can lead to these types of situations. In these cases, you must assess the reliability and usefulness of the bootstrapped results.

The following plot illustrates these bootstrapped probability curves. The curves are concave, meaning that the marginal default probability decreases with time. This result is consistent with the market information that indicates a higher default risk in the short term. The second bootstrapped curve is non-monotone, as indicated by the warning.

```
ProbTimes = yearfrac(Settle, MarketDates);
figure
plot([0; ProbTimes],[0; ProbDataInv1(:,2)])
hold on
plot([0; ProbTimes],[0; ProbDataInv2(:,2)],'--')
hold off
grid on
axis([0 ProbTimes(end,1) 0 ProbDataInv1(end,2)])
xlabel('Time (years)')
ylabel('Cumulative Default Probability')
title('Probability Curves for Inverted Spread Curves')
legend('1st instance','2nd instance','location','SouthEast')
```

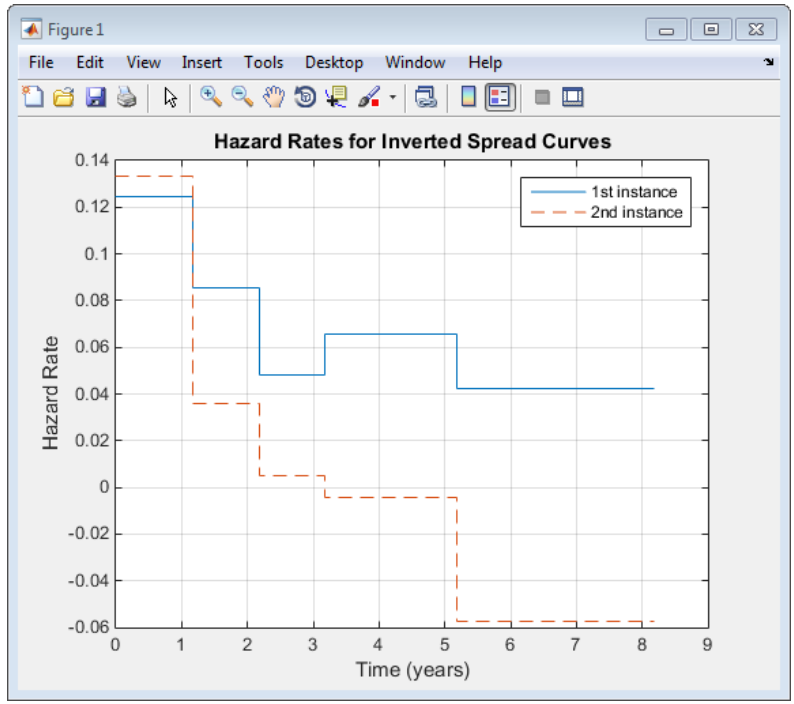
The resulting plot



Also in line with the previous plot, the hazard rates for these bootstrapped curves are decreasing because the short-term risk is higher. Some bootstrapped parameters in the second curve are negative, as indicated by the warning.

```
HazTimes = yearfrac(Settle, MarketDates);
figure
stairs([0; HazTimes(1:end-1,1); HazTimes(end,1)+1],...
   [HazDataInv1(:,2);HazDataInv1(end,2)])
hold on
stairs([0; HazTimes(1:end-1,1); HazTimes(end,1)+1],...
   [HazDataInv2(:,2);HazDataInv2(end,2)],'--')
hold off
grid on
xlabel('Time (years)')
ylabel('Hazard Rate')
title('Hazard Rates for Inverted Spread Curves')
legend('1st instance','2nd instance','location','NorthEast')
```

The resulting plot shows the hazard rates for both bootstrapped curves:



For further discussion on inverted curves, and their relationship to arbitrage, see O'Kane and Turnbull, 2003 (“Credit Derivatives” on page A-7).

See Also

`cdsbootstrap` | `cdsprice` | `cdsrpv01` | `cdsspread`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

External Websites

- Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Regression with Missing Data

Multivariate Normal Regression

In this section...

Introduction

This section focuses on using likelihood-based methods for multivariate normal regression. The parameters of the regression model are estimated via maximum likelihood estimation. For multiple series, this requires iteration until convergence. The complication due to the possibility of missing data is incorporated into the analysis with a variant of the EM algorithm known as the ECM algorithm.

The underlying theory of maximum likelihood estimation and the definition and significance of the Fisher information matrix can be found in Caines [1] and Cramér [2]. The underlying theory of the ECM algorithm can be found in Meng and Rubin [8] and Sexton and Swensen [9].

In addition, these two examples of maximum likelihood estimation are presented:

- "Portfolios with Missing Data" on page 9-27
- "Estimation of Some Technology Stock Betas" on page 9-35

Multivariate Normal Linear Regression

Suppose that you have a multivariate normal linear regression model in the form

$$\begin{bmatrix} Z_1 \\ \vdots \\ Z_m \end{bmatrix} \sim N\left(\begin{bmatrix} H_1 b \\ \vdots \\ H_m b \end{bmatrix}, \begin{bmatrix} C & & 0 \\ & \ddots & \\ 0 & & C \end{bmatrix} \right),$$

where the model has m observations of n-dimensional random variables $\mathbf{Z}_1$, ..., $\mathbf{Z}_m$ with a linear regression model that has a p-dimensional model parameter vector **b**. In addition, the model has a sequence of m design matrices $\mathbf{H}_1$, ..., $\mathbf{H}_m$, where each design matrix is a known n-by-p matrix.

Given a parameter vector **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 **0** and `n`-by-`n` covariance matrix **C** for each $k = 1, ..., m$.

A concise way to write this model is

$$Z_k \sim N(H_k b, C)$$

for $k = 1, ..., m$.

The goal of multivariate normal regression is to obtain maximum likelihood estimates for **b** and **C** given a collection of m observations $\mathbf{z}_1$, ..., $\mathbf{z}_m$ of the random variables $\mathbf{Z}_1$, ..., $\mathbf{Z}_m$. The estimated parameters are the p distinct elements of **b** and the $n\,(n + 1)/2$ distinct elements of **C** (the lower-triangular elements of **C**).

Note Quasi-maximum likelihood estimation works with the same models but with a relaxation of the assumption of normally distributed residuals. In this case, however, the parameter estimates are asymptotically optimal.

Maximum Likelihood Estimation

To estimate the parameters of the multivariate normal linear regression model using maximum likelihood estimation, it is necessary to maximize the log-likelihood function over the estimation parameters given observations $\mathbf{z}_1$, ... , $\mathbf{z}_m$.

Given the multivariate normal model to characterize residual errors in the regression model, the log-likelihood function is

$$L(z_1,\ldots,z_m; b, C) = \frac{1}{2}mn\log(2\pi) + \frac{1}{2}m\log(\det(C)) + \frac{1}{2}\sum_{k=1}^{m}(z_k - H_k b)^T C^{-1}(z_k - H_k b).$$

Although the cross-sectional residuals must be independent, you can use this log-likelihood function for quasi-maximum likelihood estimation. In this case, the estimates for the parameters **b** and **C** provide estimates to characterize the first and second moments of the residuals. See Caines [1] for details.

Except for a special case (see "Special Case of Multiple Linear Regression Model" on page 9-4), if both the model parameters in **b** and the covariance parameters in **C** are to be estimated, the estimation problem is intractably nonlinear and a solution must use iterative methods. Denote estimates for the parameters **b** and **C** for iteration $t = 0, 1, \ldots$ 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 **b** and **C** are obtained using a two-stage iterative process with

$$b^{(t+1)} = \left(\sum_{k=1}^{m} {H_k}^T \left(C^{(t)}\right)^{-1} H_k\right)^{-1} \left(\sum_{k=1}^{m} {H_k}^T \left(C^{(t)}\right)^{-1} z_k\right)$$

and

$$C^{(t+1)} = \frac{1}{m}\sum_{k=1}^{m}\left(z_k - H_k b^{(t+1)}\right)\left(z_k - H_k b^{(t+1)}\right)^T$$

for $t = 0, 1, \ldots$.

Special Case of Multiple Linear Regression Model

The special case mentioned in "Maximum Likelihood Estimation" on page 9-3 occurs if $n = 1$ so that the sequence of observations is a sequence of scalar observations. This model is known as a multiple linear regression model. In this case, the covariance matrix **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 $\mathbf{b}$ and $\mathbf{C}$, where

$$b^{LS} = \left(\sum_{k=1}^{m} H_k{}^T H_k \right)^{-1} \left(\sum_{k=1}^{m} H_k{}^T z_k \right)$$

and

$$C^{LS} = \frac{1}{m} \sum_{k=1}^{m} \left(z_k - H_k b^{LS} \right) \left(z_k - H_k b^{LS} \right)^T .$$

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 $\mathbf{z}_1, \ldots, \mathbf{z}_m$. 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 $i = 1, \ldots, m$ so that $\mathbf{b}$ is an estimate for the mean and $\mathbf{C}$ is an estimate of the covariance of the collection of observations $\mathbf{z}_1, \ldots, \mathbf{z}_m$.

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 $\mathbf{C}^{ML}$.

If $n = 1$ (which implies a single data series), convergence occurs after only one iterative step, which, in turn, implies that the least-squares and maximum likelihood estimates are identical. If, however, $n > 1$, the least-squares and maximum likelihood estimates are usually distinct.

In Financial Toolbox software, both the changes in the log-likelihood function and the norm of the change in parameter estimates are monitored. Whenever both changes fall below specified tolerances (which should be something between machine precision and its square root), the toolbox functions terminate under an assumption that convergence has been achieved.

Fisher Information

Since maximum likelihood estimates are formed from samples of random variables, their estimators are random variables; an estimate derived from such samples has an uncertainty associated with it. To characterize these uncertainties, which are called standard errors, two quantities are derived from the total log-likelihood function.

The Hessian of the total log-likelihood function is

$$\nabla^2 L(z_1,\ldots,z_m;\theta)$$

and the Fisher information matrix is

$$I(\theta) = -E\left[\nabla^2 L(z_1,\ldots,z_m;\theta)\right],$$

where the partial derivatives of the ∇^2 operator are taken with respect to the combined parameter vector Θ that contains the distinct components of **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

$$\left(\theta^{(t)} - \theta\right) \sim N\left(0, I^{-1}, \left(\theta^{(t)}\right)\right) \text{ as } t \to \infty,$$

where Θ is the combined parameter vector and $\Theta^{(t)}$ is the estimate for the combined parameter vector at iteration $t = 0, 1, \ldots$.

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

$$s^2\left(\hat{\theta}_i\right) = \left(I^{-1}\left(\hat{\theta}_i\right)\right)_{ii}$$

for $i = 1, \ldots, q$.

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

$$\hat{\theta}_i \pm 1.96 s\left(\hat{\theta}_i\right)$$

for $i = 1, \ldots, q$.

Error ellipses at a level-of-significance $\alpha \in [0, 1]$ for the parameter estimates satisfy the inequality

$$(\theta - \hat{\theta})^T I(\hat{\theta})(\theta - \hat{\theta}) \le \chi^2_{1-\alpha,q}$$

and follow a χ^2 distribution with q degrees-of-freedom. Similar inequalities can be formed for any subcollection of the parameters.

In general, given parameter estimates, the computed Fisher information matrix, and the log-likelihood function, you can perform numerous statistical tests on the parameters, the model, and the regression.

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- “Maximum Likelihood Estimation with Missing Data” on page 9-8
- “Multivariate Normal Regression Types” on page 9-16
- “Valuation with Missing Data” on page 9-33

Maximum Likelihood Estimation with Missing Data

In this section...

Introduction

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 use a version of the Expectation Maximization, or EM, algorithm of Dempster, Laird, and Rubin [3] to estimate the parameters of the multivariate normal regression model. The algorithm used in Financial Toolbox software is the ECM (Expectation Conditional Maximization) algorithm of Meng and Rubin [8] with enhancements by Sexton and Swensen [9].

Each sample $\mathbf{z}_k$ for $k = 1, ..., m$, 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 ignorabable conditions, consider an example of stock price data before an IPO. For a counterexample, censored data, in which all values greater than some cutoff are replaced with `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. Define a permutation matrix $\mathbf{P}_k$ so that

$$z_k = P_k \begin{bmatrix} x_k \\ y_k \end{bmatrix}$$

for $k = 1, ..., m$.

ECM Algorithm

The ECM algorithm has two steps – an E, or expectation step, and a CM, or conditional maximization, step. As with maximum likelihood estimation, the parameter estimates

evolve according to an iterative process, where estimates for the parameters after t iterations are denoted as $\mathbf{b}^{(t)}$ and $\mathbf{C}^{(t)}$.

The E step forms conditional expectations for the elements of missing data with

$$E\left[X_k | Y_k = y_k; b^{(t)}, C^{(t)} \right]$$
$$cov\left[X_k | Y_k = y_k; b^{(t)}, C^{(t)} \right]$$

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. So, the Fisher information matrix provides standard errors that are a Cramér-Rao lower bound whereas the Hessian matrix provides standard errors that may be greater if there is missing data.

Data Augmentation

The ECM functions do not “fill in” missing values as they estimate model parameters. In some cases, you may want to fill in the missing values. Although you can fill in the missing values in your data with conditional expectations, you would get optimistic and unrealistic estimates because conditional estimates are not random realizations.

Several approaches are possible, including resampling methods and multiple imputation (see Little and Rubin [7] and Shafer [10] for details). A somewhat informal sampling method for data augmentation is to form random samples for missing values based on

the conditional distribution for the missing values. Given parameter estimates for $X \subset R^n$ and $\hat{C}$, each observation has moments

$$E[Z_k] = H_k\hat{b}$$

and

$$cov(Z_k) = H_k\hat{C}H_k{}^T$$

for $k = 1, ..., m$, where you have dropped the parameter dependence on the left sides for notational convenience.

For observations with missing values partitioned into missing values $\mathbf{X}_k$ and observed values $\mathbf{Y}_k = \mathbf{y}_k$, you can form conditional estimates for any subcollection of random variables within a given observation. Thus, given estimates $\boldsymbol{E}[\ \mathbf{Z}_k\ ]$ and $cov(\mathbf{Z}_k)$ based on the parameter estimates, you can create conditional estimates

$$E\left[X_k \,|\, y_k\right]$$

and

$$cov\left(X_k \,|\, y_k\right)$$

using standard multivariate normal distribution theory. Given these conditional estimates, you can simulate random samples for the missing values from the conditional distribution

$$X_k \sim N\left(E\left[X_k \,|\, y_k\right], cov\left(X_k \,|\, y_k\right)\right).$$

The samples from this distribution reflect the pattern of missing and nonmissing values for observations $k = 1, ..., m$. 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.

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- "Multivariate Normal Regression" on page 9-2
- "Multivariate Normal Regression Types" on page 9-16
- "Valuation with Missing Data" on page 9-33

Multivariate Normal Regression Functions

In this section...

Financial Toolbox software has a number of functions for multivariate normal regression with or without missing data. The toolbox functions solve four classes of regression problems with functions to estimate parameters, standard errors, log-likelihood functions, and Fisher information matrices. The four classes of regression problems are:

- “Multivariate Normal Regression Without Missing Data” on page 9-13
- “Multivariate Normal Regression With Missing Data” on page 9-14
- “Least-Squares Regression With Missing Data” on page 9-14
- “Multivariate Normal Parameter Estimation With Missing Data” on page 9-14

Additional support functions are also provided, see “Support Functions” on page 9-15.

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. The moment estimation functions have `NumSeries = NumParams`.

The collection of observations (or samples) is stored in a MATLAB matrix `Data` such that

$$\mathrm{Data}(\mathrm{k},\ :) = z_k^T$$

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

$$\text{Design}\{k\} = H_k$$

for `k = 1, ..., NumSamples`. If `Design` has a single cell, it is assumed to be the same `Design` matrix for each sample such that

$$\text{Design}\{1\} = H_1 = \ldots = H_m.$$

Otherwise, `Design` must contain individual design matrices for each sample.

The main distinction among the four classes of regression problems depends upon how missing values are handled and where missing values are represented as the MATLAB value `NaN`. If a sample is to be ignored given any missing values in the sample, the problem is said to be a problem "without missing data." If a sample is to be ignored if and only if every element of the sample is missing, the problem is said to be a problem "with missing data" since the estimation must account for possible `NaN` values in the data.

In general, `Data` may or may not have missing values and `Design` should have no missing values. In some cases, however, if an observation in `Data` is to be ignored, the corresponding elements in `Design` are also ignored. Consult the function reference pages for details.

Multivariate Normal Regression Without Missing Data

You can use the following functions for multivariate normal regression without missing data.

`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

You can use the following functions 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

You can use the following functions for least-squares regression with missing data or for covariance-weighted least-squares regression with a fixed covariance matrix.

`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

You can use the following functions 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 specialized problem. Consult the function reference pages for details.

Support Functions

Two 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.)

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Multivariate Normal Regression Types” on page 9-16
- “Valuation with Missing Data” on page 9-33

Multivariate Normal Regression Types

In this section...

Regressions

Each regression function has a specific operation. This section shows how to use these functions to perform specific types of regressions. To illustrate use of the functions for various regressions, "typical" usage is shown with optional arguments kept to a minimum. For a typical regression, you estimate model parameters and residual covariance matrices with the `mle` functions and estimate the standard errors of model parameters with the `std` functions. The regressions "without missing data" essentially ignore samples with any missing values, and the regressions "with missing data" ignore samples with every value missing.

Multivariate Normal Regression

Multivariate normal regression, or MVNR, is the "standard" implementation of the regression functions in Financial Toolbox software.

Multivariate Normal Regression Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Multivariate Normal Regression With Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmmvnrmle(Data, Design);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Least-Squares Regression

Least-squares regression, or LSR, sometimes called ordinary least-squares or multiple linear regression, is the simplest linear regression model. It also enjoys the property that, independent of the underlying distribution, it is a best linear unbiased estimator (BLUE).

Given m = `NumSamples` observations, the typical least-squares regression model seeks to minimize the objective function

$$\sum_{k=1}^{m} (Z_k - H_k b)^T (Z_k - H_k b),$$

which, within the maximum likelihood framework of the multivariate normal regression routine `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.

Least-Squares Regression Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design, 1);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Least-Squares Regression With Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmlsrmle(Data, Design);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Covariance-Weighted Least Squares

Given m = `NUMSAMPLES` observations, the typical covariance-weighted least squares, or CWLS, regression model seeks to minimize the objective function

$$\sum_{k=1}^{m} (Z_k - H_k b)^T C_0 (Z_k - H_k b)$$

with fixed covariance C_0.

In most cases, C_0 is a diagonal matrix. The inverse matrix $W = C_0^{-1}$ has diagonal elements that can be considered relative “weights” for each series. Thus, CWLS is a form of weighted least squares with the weights applied across series.

Covariance-Weighted Least Squares Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, Design, 1, [], [], [],
                                   Covar0);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, Design, Covariance);
```

Covariance-Weighted Least Squares With Missing Data

Estimate Parameters

```
[Parameters, Covariance] = ecmlsrmle(Data, Design, [], [], [], [],
                                     Covar0);
```

Estimate Standard Errors

```
StdParameters = ecmmvnrstd(Data, Design, Covariance);
```

Feasible Generalized Least Squares

An ad hoc form of least squares that has surprisingly good properties for misspecified or nonnormal models is known as feasible generalized least squares, or FGLS. The basic procedure is to do least-squares regression and then to do covariance-weighted least-squares regression with the resultant residual covariance from the first regression.

Feasible Generalized Least Squares Without Missing Data

Estimate Parameters

```
[Parameters, Covariance] = 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);
```

Feasible Generalized Least Squares 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

Given a multivariate normal regression model in standard form with a `Data` matrix and a `Design` array, it is possible to convert the problem into a seemingly unrelated regression (SUR) problem by a simple transformation of the `Design` array. The main idea of SUR is that instead of having a common parameter vector over all data series, you have a separate parameter vector associated with each separate series or with distinct groups of series that, nevertheless, share a common residual covariance. It is this ability to aggregate and disaggregate series and to perform comparative tests on each design that is the power of SUR.

To make the transformation, use the function `convert2sur`, which converts a standard-form design array into an equivalent design array to do SUR with a specified mapping of the series into `NUMGROUPS` groups. The regression functions are used in the usual manner, but with the SUR design array instead of the original design array. Instead of having `NUMPARAMS` elements, the SUR output parameter vector has `NUMGROUPS` of stacked parameter estimates, where the first `NUMPARAMS` elements of `Parameters` contain parameter estimates associated with the first group of series, the next `NUMPARAMS` elements of `Parameters` contain parameter estimates associated with the second group of series, and so on. If the model has only one series, for example, `NUMSERIES` = 1, then the SUR design array is the same as the original design array since SUR requires two or more series to generate distinct parameter estimates.

Given `NUMPARAMS` parameters and `NUMGROUPS` groups with a parameter vector (`Parameters`) with `NUMGROUPS * NUMPARAMS` elements from any of the regression routines, the following MATLAB code fragment shows how to print a table of SUR

parameter estimates with rows that correspond to each parameter and columns that correspond to each group or series:

```
fprintf(1,'Seemingly Unrelated Regression Parameter
   Estimates\n');
fprintf(1,'    %7s ',' ');
fprintf(1,'  Group(%3d) ',1:NumGroups);
fprintf(1,'\n');
for i = 1:NumParams
    fprintf(1,'    %7d ',i);
    ii = i;
        for j = 1:NumGroups
            fprintf(1,'%12g ',Param(ii));
            ii = ii + NumParams;
            end
            fprintf(1,'\n');
end
fprintf(1,'\n');
```

Seemingly Unrelated Regression Without Missing Data

Form an SUR Design

```
DesignSUR = convert2sur(Design, Group);
```

Estimate Parameters

```
[Parameters, Covariance] = mvnrmle(Data, DesignSUR);
```

Estimate Standard Errors

```
StdParameters = mvnrstd(Data, DesignSUR, Covariance);
```

Seemingly Unrelated Regression With Missing Data

Form a 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 estimate the mean of your `Data` with the function `mean` and the covariance with the function `cov`. Nevertheless, the function `ecmnmle` does this for you if it detects an absence of missing values. Otherwise, it uses the ECM algorithm to handle missing values.

Estimate Parameters

```
[Mean, Covariance] = ecmnmle(Data);
```

Estimate Standard Errors

```
StdMean = ecmnstd(Data, Mean, Covariance);
```

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-8
- “Valuation with Missing Data” on page 9-33

Troubleshooting Multivariate Normal Regression

This section provides a few pointers to handle various technical and operational difficulties that might occur.

Biased Estimates

If samples are ignored, the number of samples used in the estimation is less than `NumSamples`. Clearly the actual number of samples used must be sufficient to obtain estimates. In addition, although the model parameters `Parameters` (or mean estimates `Mean`) are unbiased maximum likelihood estimates, the residual covariance estimate `Covariance` is biased. To convert to an unbiased covariance estimate, multiply `Covariance` by

$$\mathrm{Count}/(\mathrm{Count}-1),$$

where `Count` is the actual number of samples used in the estimation with $\mathrm{Count} \le \mathrm{NumSamples}$. 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 rule, the multivariate normal regression functions require

$$\mathrm{Count}\times\mathrm{NumSeries} \le \max\left\{\mathrm{NumParams},\ \mathrm{NumSeries}\times(\mathrm{NumSeries}+1)/2\right\}$$

and the least-squares regression functions require

$$\mathrm{Count}\times\mathrm{NumSeries} \le \mathrm{NumParams},$$

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 in greater detail in “Nonconvergence” on page 9-24.

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 regularly with regular updates, you can use prior estimates as initial guesses for the next period's estimation. This approach often speeds up things 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.

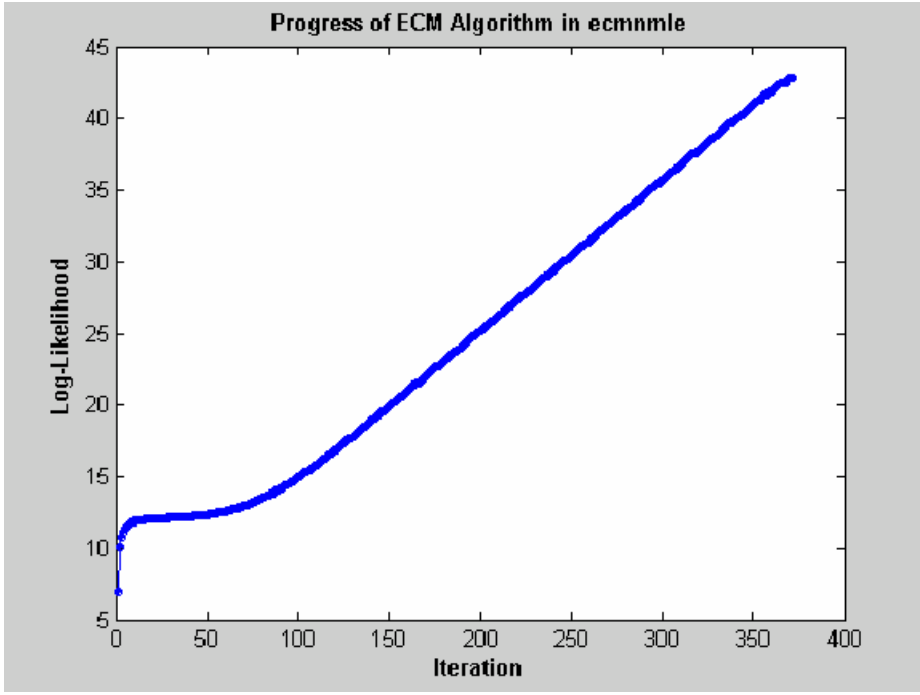
Other improvised approaches are possible although most approaches are problem-dependent. In particular, for mean and covariance estimation, the estimation function `ecmnmle` uses a function `ecmninit` to obtain an initial estimate.

Nonrandom Residuals

Simultaneous estimates for parameters **b** and covariances **C** require **C** to be positive-definite. So, the general multivariate normal regression routines require nondegenerate residual errors. If you are faced with a model that has exact results, the least-squares routine `ecmlsrmle` still works, although it provides a least-squares estimate with a singular residual covariance matrix. The other regression functions fail.

Nonconvergence

Although the regression functions are robust and work for most “typical” cases, they can fail to converge. The main failure mode is an ill-conditioned covariance matrix, where failures are either soft or hard. A soft failure wanders endlessly toward a nearly singular covariance matrix and can be spotted if the algorithm fails to converge after about 100 iterations. If `MaxIterations` is increased to 500 and display mode is initiated (with no output arguments), a typical soft failure looks like this.



This case, which is based on 20 observations of five assets with 30% of data missing, shows that the log-likelihood goes linearly to infinity as the likelihood function goes to 0. In this case, the function converges but the covariance matrix is effectively singular with a smallest eigenvalue on the order of machine precision (`eps`).

For the function `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. This is important because a soft error has a matrix that appears to be positive-definite but actually has a near-zero-valued eigenvalue to within machine precision. To do this with a covariance estimate `Covariance`, use `cond(Covariance)`, where any value greater than `1/eps` should be considered suspect.

If either type of failure occurs, however, note that the regression routine is indicating that something is probably wrong with the data. (Even with no missing data, two time series that are proportional to one another produce a singular covariance matrix.)

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-8
- “Valuation with Missing Data” on page 9-33

Portfolios with Missing Data

This example illustrates how to use the missing data algorithms for portfolio optimization and for valuation. This example works with five years of daily total return data for 12 computer technology stocks, with 6 hardware and 6 software companies. The example estimates the mean and covariance matrix for these stocks, forms efficient frontiers with both a naïve approach and the ECM approach, and compares results.

You can run the example directly with `ecmtechdemo.m`.

1 Load the following data file:

```
load ecmtechdemo
```

This file contains these three quantities:

- `Assets` is a cell array of the tickers for the 12 stocks in the example.
- `Data` is a `1254`-by-`12` matrix of 1254 daily total returns for each of the 12 stocks.
- `Dates` is a `1254`-by-`1` column vector of the dates associated with the data.

The time period for the data extends from April 19, 2000 to April 18, 2005.

The sixth stock in `Assets` is Google (GOOG), which started trading on August 19, 2004. So, all returns before August 20, 2004 are missing and represented as `NaNs`. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

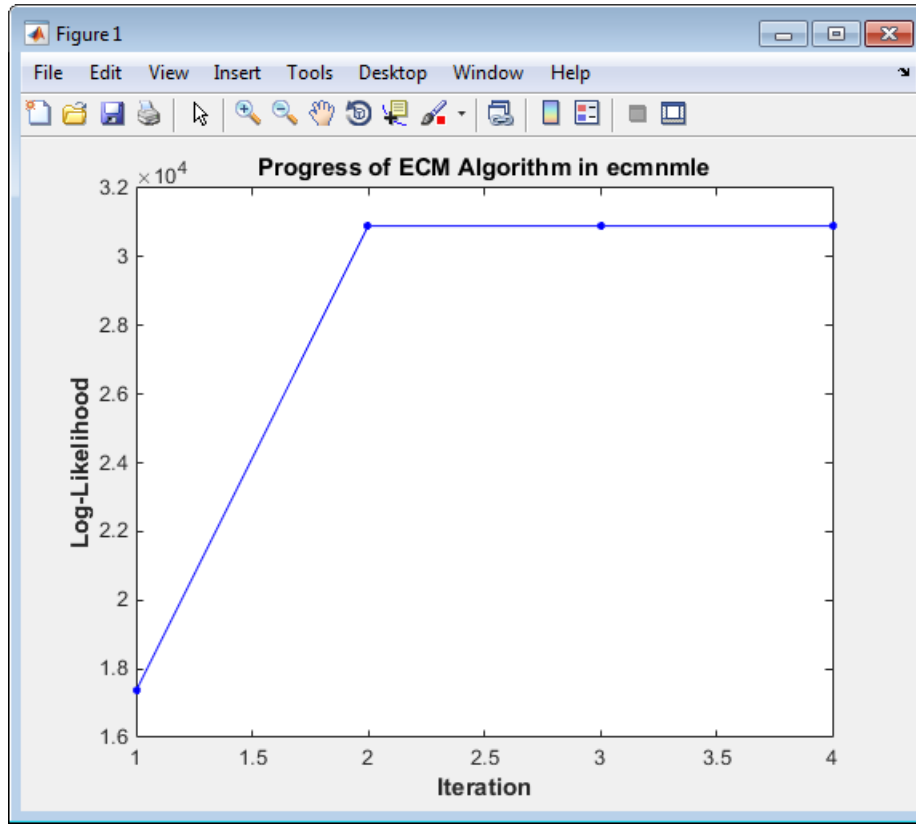
2 A naïve approach to the estimation of the mean and covariance for these 12 assets is to eliminate all days that have missing values for any of the 12 assets. Use the function `ecmninit` with the `nanskip` option to do this.

```
[NaNMean, NaNCovar] = ecmninit(Data,'nanskip');
```

3 Contrast the result of this approach with using all available data and the function `ecmnmle` to compute the mean and covariance. First, call `ecmnmle` with no output arguments to establish that enough data is available to obtain meaningful estimates.

```
ecmnmle(Data);
```

The following figure shows that, even with almost 87% of the Google data being `NaN` values, the algorithm converges after only four iterations.



4 Estimate the mean and covariance as computed by `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 =
```

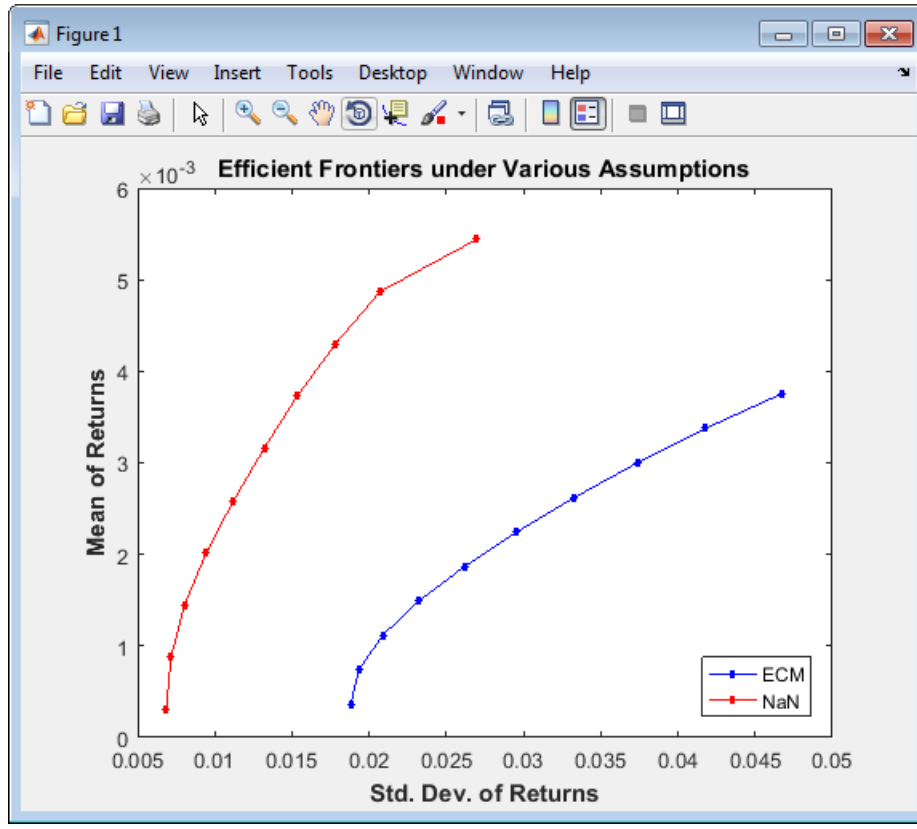
```
    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002
    0.0005    0.0010    0.0007    0.0006    0.0016    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
```

5 Given estimates for the mean and covariance of asset returns derived from the naïve and ECM approaches, estimate portfolios, and associated expected returns and risks on the efficient frontier for both approaches.

```
[ECMRisk, ECMReturn, ECMWts] = portopt(ECMMean',ECMCovar,10);
[NaNRisk, NaNReturn, NaNWts] = portopt(NaNMean',NaNCovar,10);
```

6 Plot the results on the same graph to illustrate the differences.

```
figure(gcf)
plot(ECMRisk,ECMReturn,'-bo','MarkerFaceColor','b','MarkerSize', 3);
hold on
plot(NaNRisk,NaNReturn,'-ro','MarkerFaceColor','r','MarkerSize', 3);
title('\bfEfficient Frontiers under Various Assumptions');
legend('ECM','NaN','Location','SouthEast');
xlabel('\bfStd. Dev. of Returns');
ylabel('\bfMean of Returns');
hold off
```



7 Clearly, the naïve approach is optimistic about the risk-return trade-offs for this universe of 12 technology stocks. The proof, however, lies in the portfolio weights. To view the weights, enter

```
Assets
ECMWts
NaNWts
```

which generates

```
>> Assets

ans =

    'AAPL'    'AMZN'    'CSCO'    'DELL'    'EBAY'    'GOOG'

>> ECMWts

ans =
```

```
    0.0358    0.0011   -0.0000    0.0000    0.0000    0.0989
    0.0654    0.0110    0.0000    0.0000    0.0000    0.1877
    0.0923    0.0194    0.0000    0.0000    0.0000    0.2784
    0.1165    0.0264    0.0000   -0.0000    0.0000    0.3712
    0.1407    0.0334   -0.0000         0    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 Computer and to recommend increased weights in Google relative to the naïve weights.

8 To evaluate the impact of estimation error and, in particular, the effect of missing data, use `ecmnstd` to calculate standard errors. Although it is possible to estimate the standard errors for both the mean and covariance, the standard errors for the mean estimates alone are usually the main quantities of interest.

```
StdMeanF = ecmnstd(Data,ECMMean,ECMCovar,'fisher');
```

9 Calculate standard errors that use the data-generated Hessian matrix (which accounts for the possible loss of information due to missing data) with the option `HESSIAN`.

```
StdMeanH = ecmnstd(Data,ECMMean,ECMCovar,'hessian');
```

The difference in the standard errors shows the increase in uncertainty of estimation of asset expected returns due to missing data. This can be viewed by entering:

```
Assets
StdMeanH'
StdMeanF'
StdMeanH' - StdMeanF'
```

The two assets with `missing data`, AMZN and GOOG, are the only assets to have differences due to missing information.

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-8
- “Valuation with Missing Data” on page 9-33

Valuation with Missing Data

In this section...

Introduction

The Capital Asset Pricing Model (CAPM) is a venerable but often maligned tool to characterize comovements between asset and market prices. Although many issues arise in CAPM implementation and interpretation, one problem that practitioners face is to estimate the coefficients of the CAPM with incomplete stock price data.

This example shows how to use the missing data regression functions to estimate the coefficients of the CAPM. You can run the example directly using `CAPMdemo.m` located at *matlabroot*`/toolbox/finance/findemos`.

Capital Asset Pricing Model

Given a host of assumptions that can be found in the references (see Sharpe [11], Lintner [6], Jarrow [5], and Sharpe, et. al. [12]), the CAPM concludes that asset returns have a linear relationship with market returns. Specifically, given the return of all stocks that constitute a market denoted as M and the return of a riskless asset denoted as C, the CAPM states that the return of each asset R_i in the market has the expectational form

$$E[R_i] = \alpha_i + C + \beta_i(E[M] - C)$$

for assets $i = 1, ..., n$, where β_i is a parameter that specifies the degree of comovement between a given asset and the underlying market. In other words, the expected return of each asset is equal to the return on a riskless asset plus a risk-adjusted expected market return net of riskless asset returns. The collection of parameters $\beta_1, ..., \beta_n$ is called asset betas.

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. Conversely, if an asset has a beta < 1, the asset is said to be less volatile than the market.

Estimation of the CAPM

The standard CAPM model is a linear model with additional parameters for each asset to characterize residual errors. For each of n assets with m samples of observed asset returns $R_{k,i}$, market returns M_k, and riskless asset returns C_k, the estimation model has the form

$$R_{k,i} = \alpha_i + C_k + \beta_i(M_k - C_k) + V_{k,i}$$

for samples $k = 1, ..., m$ and assets $i = 1, ..., n$, where a_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 $a_1, ..., a_n$ 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 $i,j = 1, ..., n$, where the parameters $S_{11}, ..., S_{nn}$ are called residual or nonsystematic variances/covariances.

The square root of the residual variance of each asset, for example, sqrt(S_{ii}) for $i = 1, ..., n$, is said to be the residual or nonsystematic risk of the asset since it characterizes the residual variation in asset prices that are not explained by variations in market prices.

Estimation with Missing Data

Although betas can be estimated for companies with sufficiently long histories of asset returns, it is difficult to estimate betas for recent IPOs. However, if a collection of sufficiently observable companies exists that can be expected to have some degree of correlation with the new company's stock price movements, that is, companies within the same industry as the new company, it is possible to obtain imputed estimates for new company betas with the missing-data regression routines.

Estimation of Some Technology Stock Betas

To illustrate how to use the missing-data regression routines, estimate betas for 12 technology stocks, where a single stock (GOOG) is an IPO.

1 Load dates, total returns, and ticker symbols for the 12 stocks from the MAT-file `CAPMuniverse`.

```
load CAPMuniverse
whos Assets Data Dates
```

```
  Name         Size                    Bytes  Class

  Assets       1x14                      952  cell array
  Data      1471x14                   164752  double array
  Dates     1471x1                     11768  double array

Grand total is 22135 elements using 177472 bytes
```

The assets in the model have the following symbols, where the last two series are proxies for the market and the riskless asset:

```
Assets(1:7)
Assets(8:14)
```

```
ans =

  'AAPL'    'AMZN'    'CSCO'    'DELL'    'EBAY'    'GOOG'    'HPQ'
```

```
ans =

  'IBM'    'INTC'    'MSFT'    'ORCL'    'YHOO'    'MARKET'    'CASH'
```

The data covers the period from January 1, 2000 to November 7, 2005 with daily total returns. Two stocks in this universe have missing values that are represented by `NaNs`. One of the two stocks had an IPO during this period and, so, has significantly less data than the other stocks.

2 Compute separate regressions for each stock, where the stocks with missing data 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, that is, estimates for nonsystematic risks. Instead of t-statistics, the associated standard errors for the residual standard deviations are enclosed in parentheses.

Grouped Estimation of Some Technology Stock Betas

To estimate stock betas for all 12 stocks, set up a joint regression model that groups all 12 stocks within a single design. (Since each stock has the same design matrix, this model is actually an example of seemingly unrelated regression.) The routine to estimate model parameters is `ecmmvnrmle`, and the routine to estimate standard errors is `ecmmvnrstd`.

Because GOOG has a significant number of missing values, a direct use of the missing data routine `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 are used as initial estimates to compute stock betas.

```
load CAPMgroupparam
whos Param0 Covar0
```

```
Name         Size                    Bytes  Class

Covar0      12x12                    1152  double array
Param0      24x1                      192  double array

Grand total is 168 elements using 1344 bytes
```

Now estimate the parameters for the collection of 12 stocks.

```
fprintf(1,'\n');
fprintf(1,'Grouped regression with ');
fprintf(1,'daily total return data from %s to %s ...\n', ...
    datestr(StartDate,1),datestr(EndDate,1));
fprintf(1,'  %4s %-20s %-20s %-20s\n','','Alpha','Beta','Sigma');
fprintf(1,'  ---- -------------------- ');
fprintf(1,'-------------------- --------------------\n');

NumParams = 2 * NumAssets;

% Set up grouped asset data and design matrices
TestData = zeros(NumSamples, NumAssets);
TestDesign = cell(NumSamples, 1);
Design = zeros(NumAssets, NumParams);

for k = 1:NumSamples
    for i = 1:NumAssets
        TestData(k,i) = Data(k,i) - Data(k,14);
        Design(i,2*i - 1) = 1.0;
        Design(i,2*i) = Data(k,13) - Data(k,14);
    end
    TestDesign{k} = Design;
end

% Estimate CAPM for all assets together with initial parameter
% estimates
[Param, Covar] = 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));

StdAlpha = StdParam(1:2:end-1);
StdBeta = StdParam(2:2:end);
StdSigma = sqrt(diag(StdCovar));

% Display estimates
for i = 1:NumAssets
  fprintf('  %4s %9.4f (%8.4f) %9.4f (%8.4f) %9.4f (%8.4f)\n', ...
  Assets{i},Alpha(i),abs(Alpha(i)/StdAlpha(i)), ...
  Beta(i),abs(Beta(i)/StdBeta(i)),Sigma(i),StdSigma(i));
end
```

This code fragment generates the following table.

```
Grouped regression with daily total return data from 03-Jan-2000
to 07-Nov-2005 ...
       Alpha                 Beta                Sigma
---------------------- ------------------------------------------
AAPL    0.0012 (  1.3882)    1.2294 ( 17.1839)    0.0322 (  0.0062)
AMZN    0.0007 (  0.6086)    1.3673 ( 13.6427)    0.0450 (  0.0086)
CSCO   -0.0002 (  0.2878)    1.5653 ( 23.6085)    0.0298 (  0.0057)
DELL   -0.0000 (  0.0368)    1.2594 ( 22.2164)    0.0255 (  0.0049)
EBAY    0.0014 (  1.4326)    1.3441 ( 16.0732)    0.0376 (  0.0072)
GOOG    0.0041 (  2.8907)    0.6173 (  3.1100)    0.0337 (  0.0065)
HPQ     0.0001 (  0.1747)    1.3745 ( 24.2390)    0.0255 (  0.0049)
IBM    -0.0000 (  0.0312)    1.0807 ( 28.7576)    0.0169 (  0.0032)
INTC    0.0001 (  0.1608)    1.6002 ( 27.3684)    0.0263 (  0.0050)
MSFT   -0.0002 (  0.4871)    1.1765 ( 27.4554)    0.0193 (  0.0037)
ORCL    0.0000 (  0.0389)    1.5010 ( 21.1855)    0.0319 (  0.0061)
YHOO    0.0001 (  0.1282)    1.6543 ( 19.3838)    0.0384 (  0.0074)
```

Although the results for complete-data stocks are the same, 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. However, 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. So, it is less tightly correlated with the market, implying that it is less volatile than the market (beta < 1).

References

[1] Caines, Peter E. *Linear Stochastic Systems*. John Wiley & Sons, Inc., 1988.

[2] Cramér, Harald. *Mathematical Methods of Statistics*. Princeton University Press, 1946.

[3] Dempster, A.P, N.M. Laird, and D.B Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm,"*Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, 1977, pp. 1-37.

[4] Greene, William H. *Econometric Analysis*, 5th ed., Pearson Education, Inc., 2003.

[5] Jarrow, R.A. *Finance Theory*, Prentice-Hall, Inc., 1988.

[6] Lintner, J. "The Valuation of Risk Assets and the Selection of Risky Investments in Stocks," *Review of Economics and Statistics*, Vol. 14, 1965, pp. 13-37.

[7] Little, Roderick J. A and Donald B. Rubin. *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[8] Meng, Xiao-Li and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm," *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278.

[9] Sexton, Joe and Anders Rygh Swensen. "ECM Algorithms that Converge at the Rate of EM," *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

[10] Shafer, J. L. *Analysis of Incomplete Multivariate Data*, Chapman & Hall/CRC, 1997.

[11] Sharpe, W. F. "Capital Asset Prices: A Theory of Market Equilibrium Under Conditions of Risk," *Journal of Finance*, Vol. 19, 1964, pp. 425-442.

[12] Sharpe, W. F., G. J. Alexander, and J. V. Bailey. *Investments*, 6th ed., Prentice-Hall, Inc., 1999.

See Also

`convert2sur` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrfish` | `ecmmvnrfish` | `ecmmvnrmle` | `ecmmvnrobj` | `ecmmvnrstd` | `ecmmvnrstd` | `ecmnfish` | `ecmnhess` |

`ecmninit` | `ecmnmle` | `ecmnobj` | `ecmnstd` | `mvnrfish` | `mvnrmle` | `mvnrobj` | `mvnrstd`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-8
- “Multivariate Normal Regression Types” on page 9-16

Solving Sample Problems

Introduction

This section shows how Financial Toolbox functions solve real-world problems. The examples ship with the toolbox as MATLAB files. Try them by entering the commands directly or by executing the code.

This section contains two major topics:

- A demonstration of how Financial Toolbox solves real-world financial problems, specifically:
  - "Sensitivity of Bond Prices to Interest Rates" on page 10-3
  - "Bond Portfolio for Hedging Duration and Convexity" on page 10-7
  - "Bond Prices and Yield Curve Parallel Shifts" on page 10-11
  - "Greek-Neutral Portfolios of European Stock Options" on page 10-19
  - "Term Structure Analysis and Interest-Rate Swaps" on page 10-23
- An illustration of how the toolbox produces presentation-quality graphics by solving these problems:
  - "Plotting an Efficient Frontier Using portopt" on page 10-27
  - "Plotting Sensitivities of an Option" on page 10-31
  - "Plotting Sensitivities of a Portfolio of Options" on page 10-34

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Sensitivity of Bond Prices to Interest Rates

Macaulay and *modified duration* measure the sensitivity of a bond's price to changes in the level of interest rates. *Convexity* measures the change in duration for small shifts in the yield curve, and thus measures the second-order price sensitivity of a bond. Both measures can gauge the vulnerability of a bond portfolio's value to changes in the level of interest rates.

Alternatively, analysts can use duration and convexity to construct a bond portfolio that is partly hedged against small shifts in the term structure. If you combine bonds in a portfolio whose duration is zero, the portfolio is insulated, to some extent, against interest rate changes. If the portfolio convexity is also zero, this insulation is even better. However, since hedging costs money or reduces expected return, you must know how much protection results from hedging duration alone compared to hedging both duration and convexity.

This example demonstrates a way to analyze the relative importance of duration and convexity for a bond portfolio using some of the SIA-compliant bond functions in Financial Toolbox software. Using duration, it constructs a first-order approximation of the change in portfolio price to a level shift in interest rates. Then, using convexity, it calculates a second-order approximation. Finally, it compares the two approximations with the true price change resulting from a change in the yield curve.

Step 1

Define three bonds using values for the settlement date, maturity date, face value, and coupon rate. For simplicity, accept default values for the coupon payment periodicity (semiannual), end-of-month payment rule (rule in effect), and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (no odd first or last coupon dates). Any inputs for which defaults are accepted are set to empty matrices (`[]`) as placeholders where appropriate.

```
Settle     = '19-Aug-1999';
Maturity   = ['17-Jun-2010'; '09-Jun-2015'; '14-May-2025'];
Face       = [100; 100; 1000];
CouponRate = [0.07; 0.06; 0.045];
```

Also, specify the yield curve information.

```
Yields = [0.05; 0.06; 0.065];
```

Step 2

Use Financial Toolbox functions to calculate the price, modified duration in years, and convexity in years of each bond.

The true price is quoted (clean) price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields, CouponRate,...
Settle, Maturity, 2, 0, [], [], [], [], [], Face);

Durations = bnddury(Yields, CouponRate, Settle, Maturity, 2, 0,...
[], [], [], [], [], Face);

Convexities = bndconvy(Yields, CouponRate, Settle, Maturity, 2, 0,...
[], [], [], [], [], Face);

Prices  =  CleanPrice + AccruedInterest;
```

Step 3

Choose a hypothetical amount by which to shift the yield curve (here, 0.2 percentage point or 20 basis points).

```
dY = 0.002;
```

Weight the three bonds equally, and calculate the actual quantity of each bond in the portfolio, which has a total value of $100,000.

```
PortfolioPrice   = 100000;
PortfolioWeights = ones(3,1)/3;
PortfolioAmounts = PortfolioPrice * PortfolioWeights ./ Prices;
```

Step 4

Calculate the modified duration and convexity of the portfolio. 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:

- 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 are needed for “Bond Portfolio for Hedging Duration and Convexity” on page 10-7.
- The first-order approximation, based on modified duration, predicts the new portfolio price (`PriceApprox1`), which is $97,936.37.
- The second-order approximation, based on duration and convexity, predicts the new portfolio price (`PriceApprox2`), which is $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 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`.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-23
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Bond Portfolio for Hedging Duration and Convexity

This example constructs a bond portfolio to hedge the portfolio of "Sensitivity of Bond Prices to Interest Rates" on page 10-3. It assumes a long position in (holding) the portfolio, and that three other bonds are available for hedging. It chooses weights for these three other bonds in a new portfolio so that the duration and convexity of the new portfolio match those of the original portfolio. Taking a short position in the new portfolio, in an amount equal to the value of the first portfolio, partially hedges against parallel shifts in the yield curve.

Recall that portfolio duration or convexity is a weighted average of the durations or convexities of the individual bonds in a portfolio. As in the previous example, this example uses modified duration in years and convexity in years. The hedging problem therefore becomes one of solving a system of linear equations, which is an easy to do in MATLAB software.

Step 1

Define three bonds available for hedging the original portfolio. Specify values for the settlement date, maturity date, face value, and coupon rate. For simplicity, accept default values for the coupon payment periodicity (semiannual), end-of-month payment rule (rule in effect), and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (that is, no odd first or last coupon dates). Set any inputs for which defaults are accepted to empty matrices (`[]`) as placeholders where appropriate. The intent is to hedge against duration and convexity and constrain total portfolio price.

```
Settle     = '19-Aug-1999';
Maturity   = ['15-Jun-2005'; '02-Oct-2010'; '01-Mar-2025'];
Face       = [500; 1000; 250];
CouponRate = [0.07; 0.066; 0.08];
```

Also, specify the yield curve for each bond.

```
Yields = [0.06; 0.07; 0.075];
```

Step 2

Use Financial Toolbox functions to calculate the price, modified duration in years, and convexity in years of each bond.

The true price is quoted (clean price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields,CouponRate,...
Settle, Maturity, 2, 0, [], [], [], [], [], Face);

Durations = bnddury(Yields, CouponRate, Settle, Maturity,...
2, 0, [], [], [], [], [], Face);

Convexities = bndconvy(Yields, CouponRate, Settle,...
Maturity, 2, 0, [], [], [], [], [], Face);

Prices  =  CleanPrice + AccruedInterest;
```

Step 3

Set up and solve the system of linear equations whose solution is the weights of the new bonds in a new portfolio with the same duration and convexity as the original portfolio. In addition, scale the weights to sum to `1`; that is, force them to be portfolio weights. You can then scale this unit portfolio to have the same price as the original portfolio. Recall that the original portfolio duration and convexity are `10.3181` and `157.6346`, respectively. Also, note that the last row of the linear system ensures that the sum of the weights is unity.

```
A = [Durations'
     Convexities'
     1 1 1];

b = [ 10.3181
     157.6346
       1];

Weights = A\b;
```

Step 4

Compute the duration and convexity of the hedge portfolio, which should now match the original portfolio.

```
PortfolioDuration  = Weights' * Durations;
PortfolioConvexity = Weights' * Convexities;
```

Step 5

Finally, scale the unit portfolio to match the value of the original portfolio and find the number of bonds required to insulate against small parallel shifts in the yield curve.

```
PortfolioValue = 100000;
HedgeAmounts   = Weights ./ Prices * PortfolioValue;
```

Step 6

Compare the results.

- As required, the duration and convexity of the new portfolio are `10.3181` and `157.6346`, respectively.
- The hedge amounts for bonds 1, 2, and 3 are `-57.37`, `71.70`, and `216.27`, respectively.

Notice that the hedge matches the duration, convexity, and value ($100,000) of the original portfolio. If you are holding that first portfolio, you can hedge by taking a short position in the new portfolio.

Just as the approximations of the first example are appropriate only for small parallel shifts in the yield curve, the hedge portfolio is appropriate only for reducing the impact of small level changes in the term structure.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16

Bond Prices and Yield Curve Parallel Shifts

This example uses Financial Toolbox™ bond pricing functions to evaluate the impact of time-to-maturity and yield variation on the price of a bond portfolio. Also, this example shows how to visualize the price behavior of a portfolio of bonds over a wide range of yield curve scenarios, and as time progresses toward maturity.

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. Also, specify the points on the yield curve for each bond.

```
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];

Yields = [0.078; 0.09; 0.075; 0.085];
```

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
```

```
Prices =

  145.2452
  594.7757
  165.8949
  715.7584
```

Assume that 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;
```

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
```

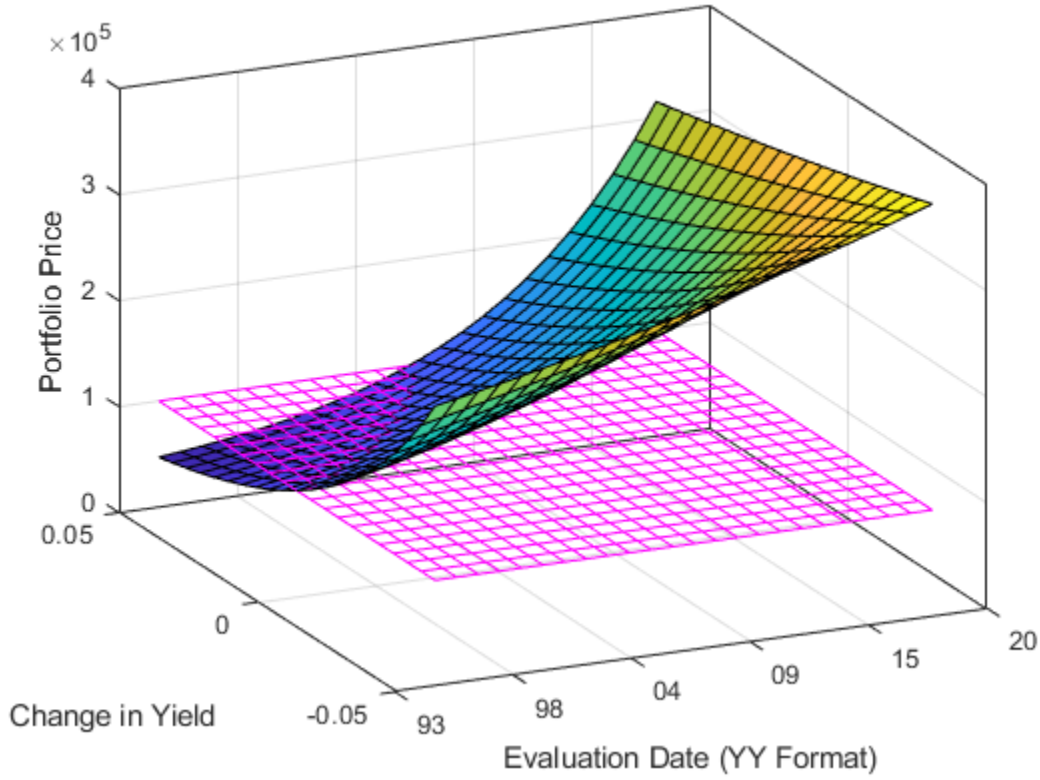
Scale the bond prices by the quantity of bonds and reshape the bond values to conform to the underlying evaluation grid.

```
Prices = Prices * BondAmounts;
Prices = reshape(Prices, NumYields, NumTimes);
```

Plot the price of the portfolio as a function of settlement date and a range of yields, and as a function of the change in yield (`dY`). This plot illustrates the interest-rate sensitivity of the portfolio as time progresses (`dT`), under a range of interest-rate scenarios. With the

following graphics commands, you can visualize the three-dimensional surface relative to the current portfolio value (that is, $100,000).

```
figure                    % Open a new figure window
surf(dt, dy, Prices)      % Draw the surface

hold on                   % Add the current value for reference
basemesh = mesh(dt, dy, 100000*ones(NumYields, NumTimes));
set(basemesh, 'facecolor', 'none');
set(basemesh, 'edgecolor', 'm');
set(gca, 'box', 'on');

dateaxis('x', 11);
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.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-23
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Bond Prices and Yield Curve Nonparallel Shifts

This example shows how to construct a bond portfolio to hedge the interest-rate risk of a Treasury bond maturing in 20 years. Key rate duration enables you to determine the sensitivity of the price of a bond to nonparallel shifts in the yield curve. This example uses `bndkrdur` to construct a portfolio to hedge the interest-rate risk of a U.S. Treasury bond maturing in 20 years.

Specify the bond.

```
Settle = datenum('2-Dec-2008');
CouponRate = 5.500/100;
Maturity = datenum('15-Aug-2028');
Price = 128.68;
```

The interest-rate risk of this bond is hedged with the following four on-the-run Treasury bonds:

```
Maturity_30 = datenum('15-May-2038');  % 30-year bond
Coupon_30 = .045;
Price_30 = 124.69;

Maturity_10 = datenum('15-Nov-2018');  %10-year note
Coupon_10 = .0375;
Price_10 = 109.35;

Maturity_05 = datenum('30-Nov-2013');  % 5-year note
Coupon_05 = .02;
Price_05 = 101.67;

Maturity_02 = datenum('30-Nov-2010');  % 2-year note
Coupon_02 = .01250;
Price_02 =  100.72;
```

You can get the Treasury spot or zero curve from http://www.treas.gov/offices/domestic-finance/debt-management/interest-rate/yield.shtml:

```
ZeroDates = daysadd(Settle,[30 90 180 360 360*2 360*3 360*5 ...
360*7 360*10 360*20 360*30]);
ZeroRates = ([0.09 0.07 0.44 0.81 0.90 1.16 1.71 2.13 2.72 3.51 3.22]/100)';
```

Compute the key rate durations for both the bond and the hedging portfolio.

```
BondKRD = bndkrdur([ZeroDates ZeroRates], CouponRate, Settle,...
Maturity,'keyrates',[2 5 10 20]);
```

```
HedgeMaturity = [Maturity_02;Maturity_05;Maturity_10;Maturity_30];
HedgeCoupon = [Coupon_02;Coupon_05;Coupon_10;Coupon_30];
HedgeKRD = bndkrdur([ZeroDates ZeroRates], HedgeCoupon,...
Settle, HedgeMaturity, 'keyrates',[2 5 10 20])

HedgeKRD =

    1.9675         0         0         0
    0.1269    4.6152         0         0
    0.2129    0.7324    7.4010         0
    0.2229    0.7081    2.1487   14.5172
```

Compute the dollar durations for each of the instruments and each of the key rates (assuming holding 100 bonds).

```
PortfolioDD = 100*Price* BondKRD;
HedgeDD = bsxfun(@times, HedgeKRD,[Price_30;Price_10;Price_05;Price_02])

HedgeDD =

   1.0e+03 *

    0.2453         0         0         0
    0.0139    0.5047         0         0
    0.0216    0.0745    0.7525         0
    0.0224    0.0713    0.2164    1.4622
```

Compute the number of bonds to sell short to obtain a key rate duration that is `0` for the entire portfolio.

```
NumBonds = PortfolioDD/HedgeDD

NumBonds =

    3.8973    6.1596   23.0282   80.0522
```

These results indicate selling 4, 6, 23 and 80 bonds respectively of the 2-, 5-, 10-, and 30-year bonds achieves a portfolio that is neutral with respect to the 2-, 5-, 10-, and 30-year spot rates.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-23
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

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 2-48 or the Glossary for other definitions.

The greeks of a particular option are a function of the model used to price the option. However, given enough different options to work with, a trader can construct a portfolio with any desired values for its greeks. For example, to insulate the value of an option portfolio from small changes in the price of the underlying asset, one trader might construct an option portfolio whose delta is zero. Such a portfolio is then said to be “delta neutral.” Another trader may want to protect an option portfolio from larger changes in the price of the underlying asset, and so might construct a portfolio whose delta and gamma are both zero. Such a portfolio is both delta and gamma neutral. A third trader may want to construct a portfolio insulated from small changes in the volatility of the underlying asset in addition to delta and gamma neutrality. Such a portfolio is then delta, gamma, and vega neutral.

Using the Black-Scholes model for European options, this example creates an equity option portfolio that is simultaneously delta, gamma, and vega neutral. The value of a particular greek of an option portfolio is a weighted average of the corresponding greek of each individual option. The weights are the quantity of each option in the portfolio. Hedging an option portfolio thus involves solving a system of linear equations, an easy process in MATLAB.

Step 1

Create an input data matrix to summarize the relevant information. Each row of the matrix contains the standard inputs to Financial Toolbox Black-Scholes suite of functions: column 1 contains the current price of the underlying stock; column 2 the strike price of each option; column 3 the time to-expiry of each option in years; column 4 the annualized stock price volatility; and column 5 the annualized dividend rate of the underlying asset. 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 that the annualized risk-free rate is 10% and is constant for all maturities of interest.

```
RiskFreeRate = 0.10;
```

For clarity, assign each column of `DataMatrix` to a column vector whose name reflects the type of financial data in the column.

```
StockPrice   = DataMatrix(:,1);
StrikePrice  = DataMatrix(:,2);
ExpiryTime   = DataMatrix(:,3);
Volatility   = DataMatrix(:,4);
DividendRate = DataMatrix(:,5);
```

Step 2

Based on the Black-Scholes model, compute the prices, and the delta, gamma, and vega sensitivity greeks of each of the four options. The functions `blsprice` and `blsdelta` have two outputs, while `blsgamma` and `blsvega` have only one. The price and delta of a call option differ from the price and delta of an otherwise equivalent put option, in contrast to the gamma and vega sensitivities, which are valid for both calls and puts.

```
[CallPrices, PutPrices] = blsprice(StockPrice, StrikePrice,...
RiskFreeRate, ExpiryTime, Volatility, DividendRate);

[CallDeltas, PutDeltas] = blsdelta(StockPrice,...
StrikePrice, RiskFreeRate, ExpiryTime, Volatility,...
DividendRate);

Gammas = blsgamma(StockPrice, StrikePrice, RiskFreeRate,...
                  ExpiryTime, Volatility , DividendRate)';

Vegas  = blsvega(StockPrice, StrikePrice, RiskFreeRate,...
                 ExpiryTime, Volatility , DividendRate)';
```

Extract the prices and deltas of interest to account for the distinction between call and puts.

```
Prices = [CallPrices(1) PutPrices(2) CallPrices(3)...
PutPrices(4)];
```

```
Deltas = [CallDeltas(1) PutDeltas(2) CallDeltas(3)...
PutDeltas(4)];
```

Step 3

Now, assuming an arbitrary portfolio value of $17,000, set up and solve the linear system of equations such that the overall option portfolio is simultaneously delta, gamma, and vega-neutral. The solution computes the value of a particular greek of a portfolio of options as a weighted average of the corresponding greek of each individual option in the portfolio. The system of equations is solved using the back slash (\) operator discussed in "Solving Simultaneous Linear Equations" on page 1-13.

```
A = [Deltas; Gammas; Vegas; Prices];
b = [0; 0; 0; 17000];
OptionQuantities = A\b; % Quantity (number) of each option.
```

Step 4

Finally, compute the market value, delta, gamma, and vega of the overall portfolio as a weighted average of the corresponding parameters of the component options. The weighted average is computed as an inner product of two vectors.

```
PortfolioValue =  Prices * OptionQuantities;
PortfolioDelta =  Deltas * OptionQuantities;
PortfolioGamma =  Gammas * OptionQuantities;
PortfolioVega  =  Vegas  * OptionQuantities;
```

The output for these computations is:

```
Option  Price    Delta    Gamma    Vega     Quantity
   1   6.3441   0.5856   0.0290  17.4293   22332.6131
   2   6.6035  -0.6255   0.0353  20.0347    6864.0731
   3   4.2993   0.7003   0.0548   9.5837  -15654.8657
   4  22.7694  -0.4830   0.0074  83.5225   -4510.5153

Portfolio Value: $17000.00
Portfolio Delta:      0.00
Portfolio Gamma:     -0.00
Portfolio Vega :      0.00
```

You can verify that the portfolio value is $17,000 and that the option portfolio is indeed delta, gamma, and vega neutral, as desired. Hedges based on these measures are effective only for small changes in the underlying variables.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Sensitivity of Bond Prices to Interest Rates" on page 10-3
- "Bond Portfolio for Hedging Duration and Convexity" on page 10-7
- "Bond Prices and Yield Curve Parallel Shifts" on page 10-11
- "Bond Prices and Yield Curve Nonparallel Shifts" on page 10-16
- "Term Structure Analysis and Interest-Rate Swaps" on page 10-23
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Term Structure Analysis and Interest-Rate Swaps

This example illustrates some of the term-structure analysis functions found in Financial Toolbox software. Specifically, it illustrates how to derive implied zero (*spot*) and forward curves from the observed market prices of coupon-bearing bonds. The zero and forward curves implied from the market data are then used to price an interest rate swap agreement.

In an interest rate swap, two parties agree to a periodic exchange of cash flows. One of the cash flows is based on a fixed interest rate held constant throughout the life of the swap. The other cash flow stream is tied to some variable index rate. Pricing a swap at inception amounts to finding the fixed rate of the swap agreement. This fixed rate, appropriately scaled by the notional principal of the swap agreement, determines the periodic sequence of fixed cash flows.

In general, interest rate swaps are priced from the forward curve such that the variable cash flows implied from the series of forward rates and the periodic sequence of fixed-rate cash flows have the same current value. Thus, interest rate swap pricing and term structure analysis are intimately related.

Step 1

Specify values for the settlement date, maturity dates, coupon rates, and market prices for 10 U.S. Treasury Bonds. This data allows you to price a five-year swap with net cash flow payments exchanged every six months. For simplicity, accept default values for the end-of-month payment rule (rule in effect) and day-count basis (actual/actual). To avoid issues of accrued interest, assume that all Treasury Bonds pay semiannual coupons and that settlement occurs on a coupon payment date.

```
Settle   = datenum('15-Jan-1999');

BondData = {'15-Jul-1999'  0.06000   99.93
            '15-Jan-2000'  0.06125   99.72
            '15-Jul-2000'  0.06375   99.70
            '15-Jan-2001'  0.06500   99.40
            '15-Jul-2001'  0.06875   99.73
            '15-Jan-2002'  0.07000   99.42
            '15-Jul-2002'  0.07250   99.32
            '15-Jan-2003'  0.07375   98.45
            '15-Jul-2003'  0.07500   97.71
            '15-Jan-2004'  0.08000   98.15};
```

`BondData` is an instance of a MATLAB *cell array*, indicated by the curly braces (`{}`).

Next assign the date stored in the cell array to `Maturity`, `CouponRate`, and `Prices` vectors for further processing.

```
Maturity   = datenum(char(BondData{:,1}));
CouponRate = [BondData{:,2}]';
Prices     = [BondData{:,3}]';
Period     = 2; % semiannual coupons
```

Step 2

Now that the data has been specified, use the term structure function `zbtprice` to bootstrap the zero curve implied from the prices of the coupon-bearing bonds. This implied zero curve represents the series of zero-coupon Treasury rates consistent with the prices of the coupon-bearing bonds such that arbitrage opportunities will not exist.

```
ZeroRates = zbtprice([Maturity CouponRate], Prices, Settle);
```

The zero curve, stored in `ZeroRates`, is quoted on a semiannual bond basis (the periodic, six-month, interest rate is doubled to annualize). The first element of `ZeroRates` is the annualized rate over the next six months, the second element is the annualized rate over the next 12 months, and so on.

Step 3

From the implied zero curve, find the corresponding series of implied forward rates using the term structure function `zero2fwd`.

```
ForwardRates = zero2fwd(ZeroRates, Maturity, Settle);
```

The forward curve, stored in `ForwardRates`, is also quoted on a semiannual bond basis. The first element of `ForwardRates` is the annualized rate applied to the interval between settlement and six months after settlement, the second element is the annualized rate applied to the interval from six months to 12 months after settlement, and so on. This implied forward curve is also consistent with the observed market prices such that arbitrage activities will be unprofitable. Since the first forward rate is also a zero rate, the first element of `ZeroRates` and `ForwardRates` are the same.

Step 4

Now that you have derived the zero curve, convert it to a sequence of discount factors with the term structure function `zero2disc`.

```
DiscountFactors = zero2disc(ZeroRates, Maturity, Settle);
```

Step 5

From the discount factors, compute the present value of the variable cash flows derived from the implied forward rates. For plain interest rate swaps, the notional principal remains constant for each payment date and cancels out of each side of the present value equation. The next line assumes unit notional principal.

```
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 principal cancels out of each side of the equation, it is assumed to be 1.

```
SwapFixedRate = Period * PresentValue / sum(DiscountFactors);
```

The output for these computations is:

```
  Zero Rates  Forward Rates
    0.0614        0.0614
    0.0642        0.0670
    0.0660        0.0695
    0.0684        0.0758
    0.0702        0.0774
    0.0726        0.0846
    0.0754        0.0925
    0.0795        0.1077
    0.0827        0.1089
    0.0868        0.1239

  Swap Price (Fixed Rate) = 0.0845
```

All rates are in decimal format. The swap price, 8.45%, would likely be the mid-point between a market-maker's bid/ask quotes.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Plotting an Efficient Frontier Using portopt

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.

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 ];
```

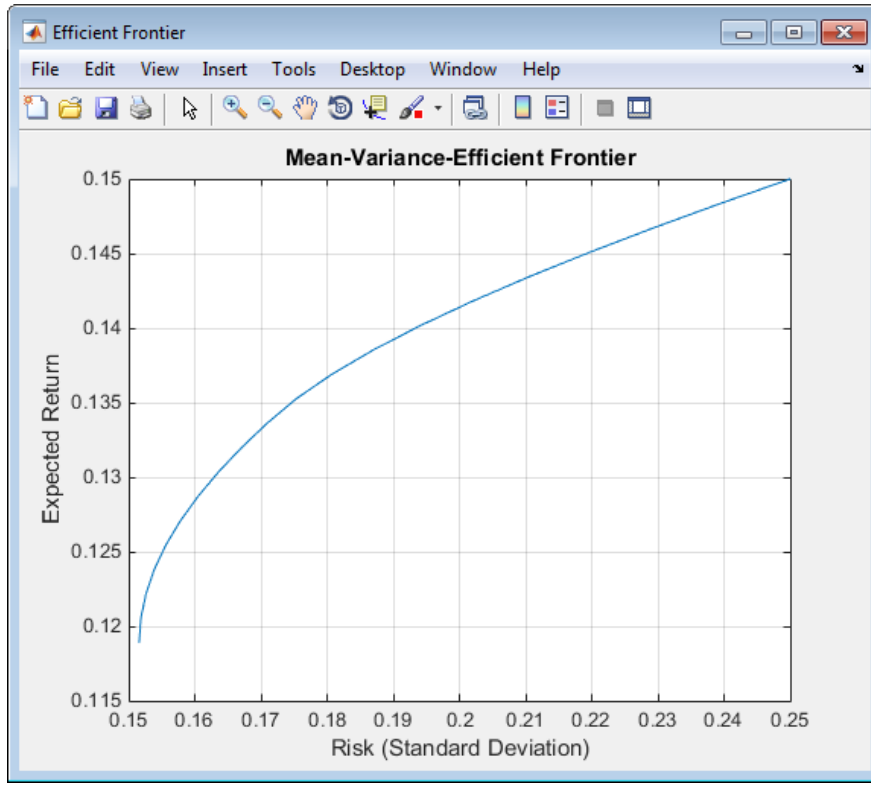
Convert the standard deviations and correlation matrix into a variance-covariance matrix with the Financial Toolbox function `corr2cov`.

```
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).

Note `portopt` has been partially removed and will no longer accept `ConSet` or `varargin` arguments. Use `Portfolio` instead to solve portfolio problems that are more than a long-only fully-invested portfolio. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21. For more information on migrating `portopt` code to `Portfolio`, see “portopt Migration to Portfolio Object” on page 3-14.

```
portopt(Returns, Covariances, 20)
```



Now that the efficient frontier is displayed, randomly generate the asset weights for 1000 portfolios starting from the MATLAB initial state.

```
rng('default')
Weights = rand(1000, 3);
```

The previous line of code generates three columns of uniformly distributed random weights, but does not guarantee they sum to 1. The following code segment normalizes the weights of each portfolio so that the total of the three weights represent a valid portfolio.

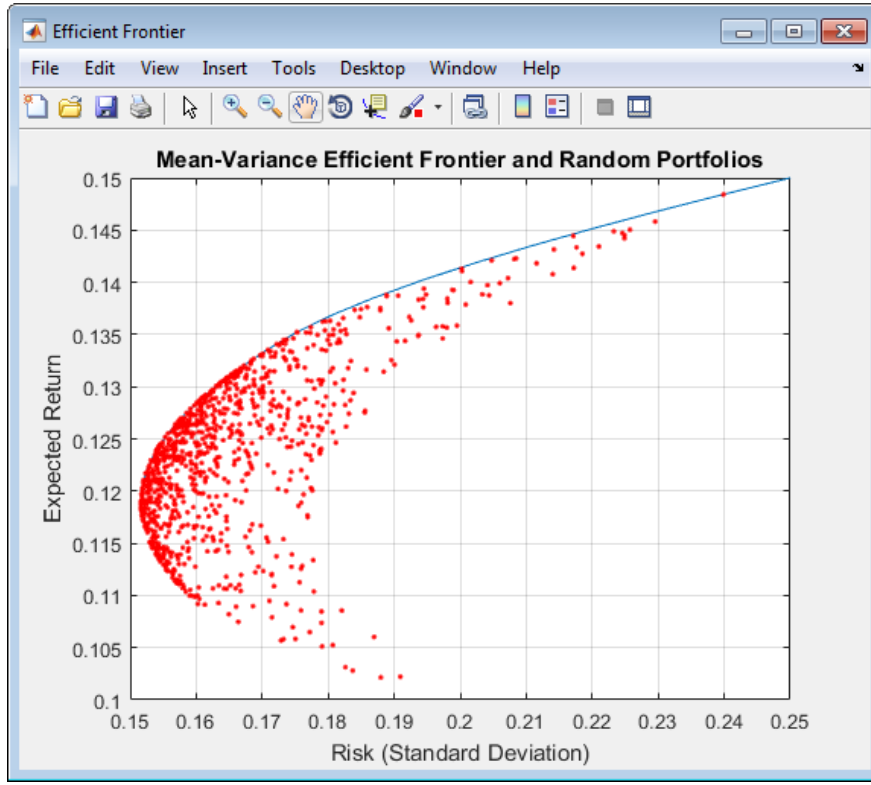
```
Total = sum(Weights, 2);      % Add the weights
Total = Total(:,ones(3,1));   % Make size-compatible matrix
Weights = Weights./Total;     % Normalize so sum = 1
```

Given the 1000 random portfolios 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
```



See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `corr2cov` | `portopt` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34
- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-23

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 in this example shows a three-dimensional surface whose *z*-value is the gamma of an option as price (*x*-axis) and time (*y*-axis) vary. The plot adds yet a fourth dimension by showing option delta (the first derivative of option price to security price) as the color of the surface. 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.

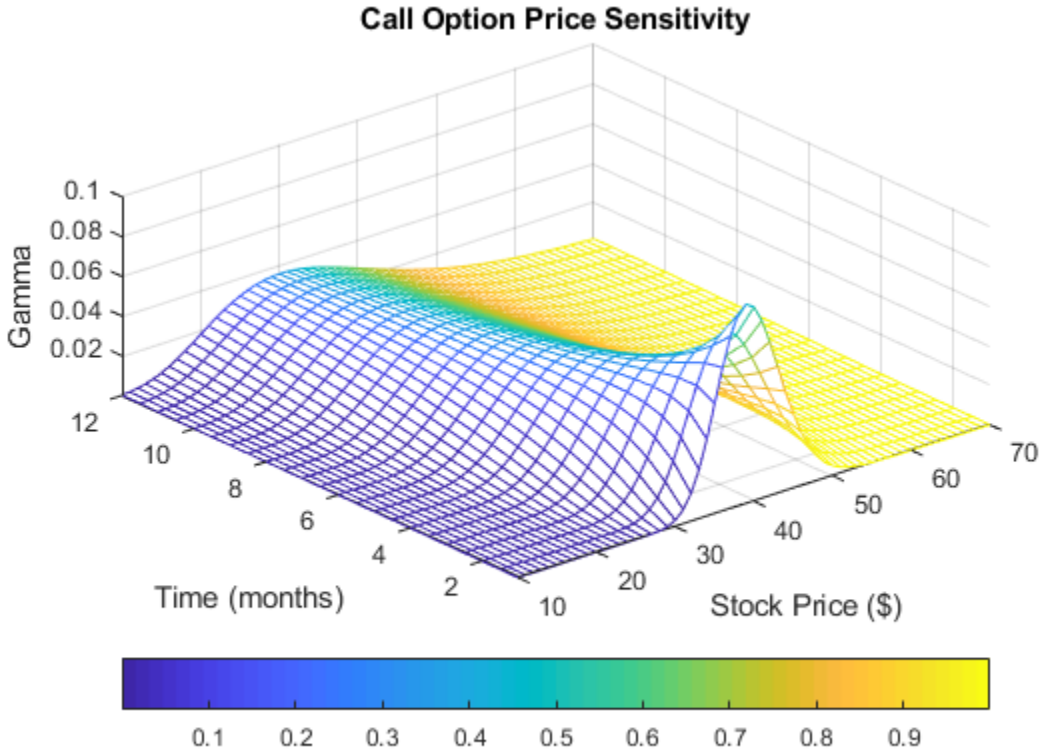
```
JSpan = ones(length(j),1);
NewRange = Range(JSpan,:);
Pad = ones(size(Newj));
```

Calculate the gamma and delta sensitivities (greeks) using the `blsgamma` and `blsdelta` functions. 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');
```



See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `corr2cov` | `portopt` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Plotting Sensitivities of a Portfolio of Options” on page 10-34
- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19

- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-23

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 in this example 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 example has applications in hedging. First set up the portfolio with arbitrary data. Current prices range from $20 to $90 for each option. Then, set the 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.

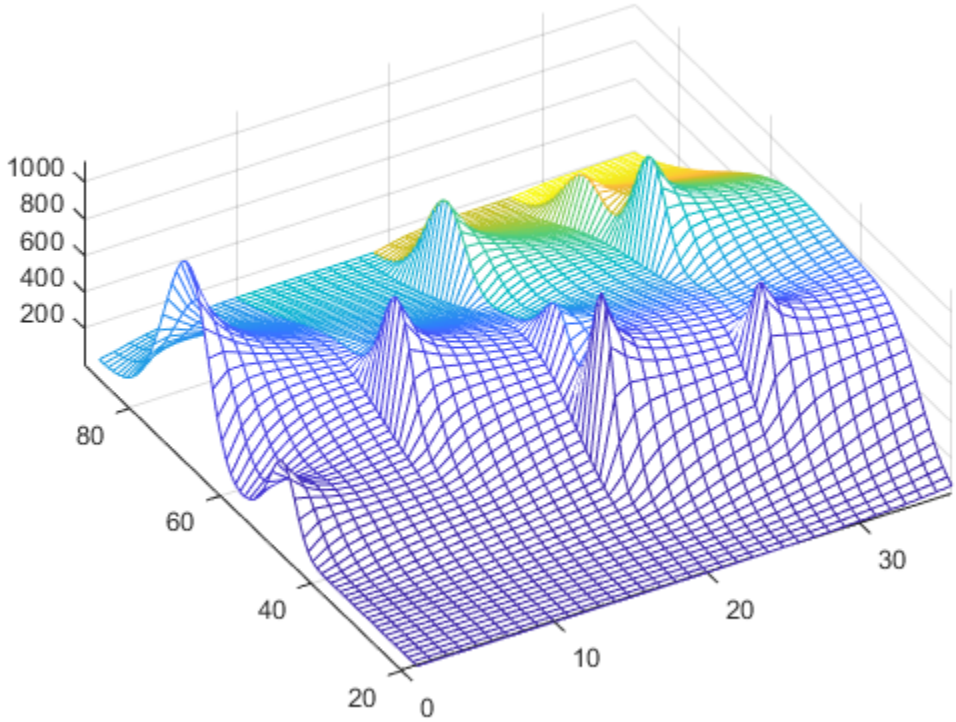
```
  T = (1:Time(i))';
  NewT = T(:,ones(PLen,1));
```

Use the Black-Scholes gamma and delta sensitivity functions `blsgamma` and `blsdelta` 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
```

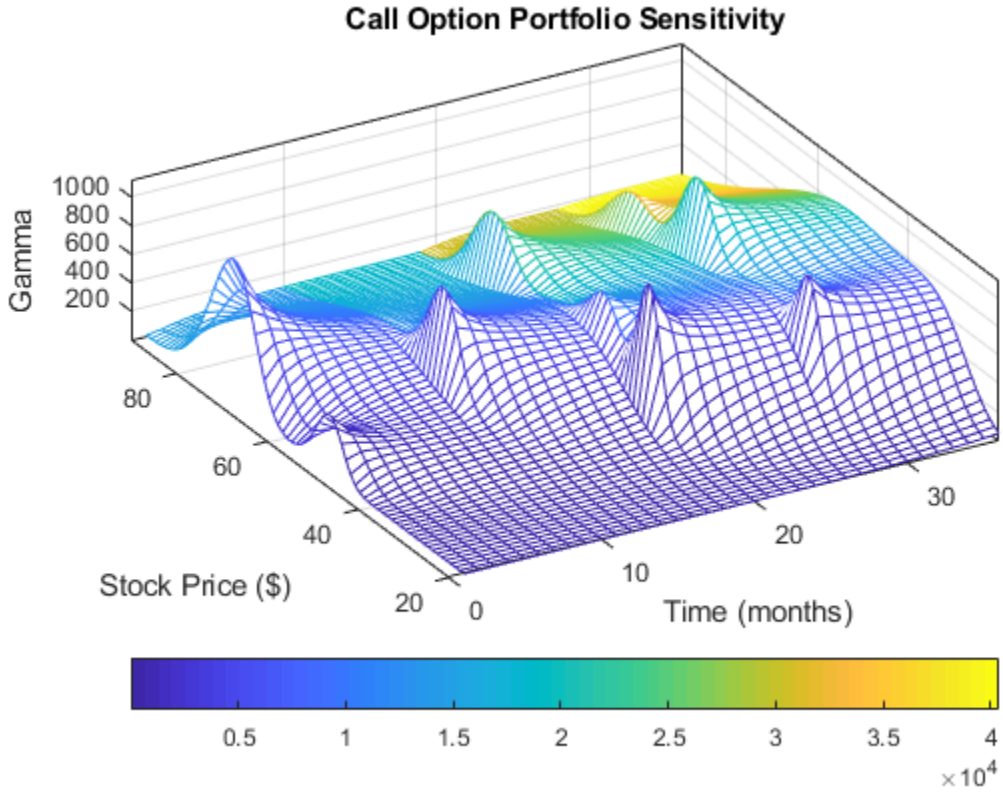
Draw the surface as a mesh, set the viewpoint, and reverse the *x*-axis because of the viewpoint. The axes range from 20 to 90, 0 to 36, and -∞ to ∞.

```
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 color bar.

```
title('Call Option Portfolio Sensitivity');
xlabel('Stock Price ($)');
ylabel('Time (months)');
zlabel('Gamma');
set(gca, 'box', 'on');
colorbar('horiz');
```



See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsvega` | `bndconvy` | `bnddury` | `bndkrdur` | `bndprice` | `corr2cov` | `portopt` | `zbtprice` | `zero2disc` | `zero2fwd`

Related Examples

- “Plotting Sensitivities of an Option” on page 10-31
- “Pricing and Analyzing Equity Derivatives” on page 2-48
- on page 10-19
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3

Financial Time Series Analysis

Analyzing Financial Time Series

Financial Toolbox software provides a collection of tools for the analysis of time series data in the financial markets. The toolbox contains a financial time series object constructor and several methods that operate on and analyze the object. Financial engineers working with time series data, such as equity prices or daily interest fluctuations, can use these tools for more intuitive data management than by using regular vectors or matrices.

This section discusses how to create a financial time series object in one of two ways:

- "Using the Constructor" on page 11-3
- "Transforming a Text File" on page 11-13

`chartfts` is a graphical tool for visualizing financial time series objects. You can find this discussion in "Visualizing Financial Time Series Objects" on page 11-16.

Creating Financial Time Series Objects

In this section...

Introduction

Financial Toolbox software provides three 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`
- Use the Financial Time Series app, you can create a financial time series (`fints`) object from one or more selected variables. For more information, see “Creating a Financial Time Series Object” on page 13-11.

The structure of the object minimally consists of a description field, a frequency indicator field, the date vector field, and at least one data series vector. The names for the fields are fixed for the first three fields: `desc`, `freq`, and `dates`. You can specify names of your choice for any data series vectors. If you do not specify names, the object uses the default names `series1`, `series2`, `series3`, and so on.

If time-of-day information is incorporated in the date vector, the object contains an additional field named `times`.

Using the Constructor

The object constructor function `fints` has five different syntaxes. These forms exist to simplify object construction. The syntaxes vary according to the types of input arguments presented to the constructor. The syntaxes are

- Single Matrix Input
  - See “Time-of-Day Information Excluded” on page 11-4.
  - See “Time-of-Day Information Included” on page 11-6.

- Separate Vector Input
  - See “Time-of-Day Information Excluded” on page 11-7.
  - See “Time-of-Day Information Included” on page 11-8.
- See “Data Name Input” on page 11-9.
- See “Frequency Indicator Input” on page 11-11.
- See “Description Field Input” on page 11-12.

Single Matrix Input

The date information provided with this syntax must be in serial date number format. The date number may on page 11-6 or may not on page 11-4 include time-of-day information.

Note If you are unfamiliar with the concepts of date character vectors and serial date numbers, consult “Handle and Convert Dates” on page 2-2.

Time-of-Day Information Excluded

```
fts = fints(dates_and_data)
```

In this simplest form of syntax, the input must be at least a two-column matrix. The first column contains the dates in serial date format; the second column is the data series. The input matrix can have more than two columns, each additional column representing a different data series or set of observations.

If the input is a two-column matrix, the output object contains four fields: `desc`, `freq`, `dates`, and `series1`. The description field, `desc`, defaults to blanks `''`, and the frequency indicator field, `freq`, defaults to `0`. The dates field, `dates`, contains the serial dates from the first column of the input matrix, while the data series field, `series1`, has the data from the second column of the input matrix.

The first example makes two financial time series objects. The first one has only one data series, while the other has more than one. A random vector provides the values for the data series. The range of dates is arbitrarily chosen using the `today` function:

```
date_series = (today:today+100)';
data_series = exp(randn(1, 101))';
dates_and_data = [date_series data_series];
fts1 = fints(dates_and_data);
```

Examine the contents of the object `fts1` create. The actual date series you observe will vary according to the day when you run the example (the value of `today`). Also, your values in `series1` will differ from those shown, depending upon the sequence of random numbers generated:

```
fts1 =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (101)'    'series1:  (101)'
    '12-Jul-1999'      [         0.3124]
    '13-Jul-1999'      [         3.2665]
    '14-Jul-1999'      [         0.9847]
    '15-Jul-1999'      [         1.7095]
    '16-Jul-1999'      [         0.4885]
    '17-Jul-1999'      [         0.5192]
    '18-Jul-1999'      [         1.3694]
    '19-Jul-1999'      [         1.1127]
    '20-Jul-1999'      [         6.3485]
    '21-Jul-1999'      [         0.7595]
    '22-Jul-1999'      [         9.1390]
    '23-Jul-1999'      [         4.5201]
    '24-Jul-1999'      [         0.1430]
    '25-Jul-1999'      [         0.1863]
    '26-Jul-1999'      [         0.5635]
    '27-Jul-1999'      [         0.8304]
    '28-Jul-1999'      [         1.0090]...
```

The output is truncated for brevity. There are actually 101 data points in the object.

The `desc` field displays as `(none)` instead of `''`, and that the contents of the object display as cell array elements. Although the object displays as such, it should be thought of as a MATLAB structure containing the default field names for a single data series object: `desc`, `freq`, `dates`, and `series1`.

Now create an object with more than one data series in it:

```
date_series = (today:today+100)';
data_series1 = exp(randn(1, 101))';
data_series2 = exp(randn(1, 101))';
dates_and_data = [date_series data_series1 data_series2];
fts2 = fints(dates_and_data);
```

Now look at the object created (again in abbreviated form):

```
fts2 =

    desc:  (none)
    freq:  Unknown (0)
```

```
'dates:  (101)'    'series1:  (101)'    'series2:  (101)'
'12-Jul-1999'      [         0.5816]    [         1.2816]
'13-Jul-1999'      [         5.1253]    [         0.9262]
'14-Jul-1999'      [         2.2824]    [         5.6869]
'15-Jul-1999'      [         1.2596]    [         5.0631]
'16-Jul-1999'      [         1.9574]    [         1.8709]
'17-Jul-1999'      [         0.6017]    [         1.0962]
'18-Jul-1999'      [         2.3546]    [         0.4459]
'19-Jul-1999'      [         1.3080]    [         0.6304]
'20-Jul-1999'      [         1.8682]    [         0.2451]
'21-Jul-1999'      [         0.3509]    [         0.6876]
'22-Jul-1999'      [         4.6444]    [         0.6244]
'23-Jul-1999'      [         1.5441]    [         5.7621]
'24-Jul-1999'      [         0.1470]    [         2.1238]
'25-Jul-1999'      [         1.5999]    [         1.0671]
'26-Jul-1999'      [         3.5764]    [         0.7462]
'27-Jul-1999'      [         1.8937]    [         1.0863]
'28-Jul-1999'      [         3.9780]    [         2.1516]...
```

The second data series name defaults to `series2`, as expected.

Before you can perform any operations on the object, you must set the frequency indicator field `freq` to the valid frequency of the data series contained in the object. You can leave the description field `desc` blank.

To set the frequency indicator field to a daily frequency, enter

`fts2.freq = 1`, or

`fts2.freq = 'daily'`.

For more information, see `fints`.

Time-of-Day Information Included

The serial date number used with this form of the `fints` function can incorporate time-of-day information. When time-of-day information is present, the output of the function contains a field `times` that indicates the time of day.

If you recode the previous example on page 11-4 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 character vector format. The date information may on page 11-8 or may not on page 11-7 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 character vector. 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 character vector date and time are present, the entry must be a column-oriented cell array of character vectors for dates and times.

With date character vector 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 character vector 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 character vector 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 character vector 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 character vector for a single data series. For multiple data series names, it must be a cell array of character vectors.

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 character vector 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 character vector `'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 if the data file conforms to a general format. The general format of the text data file is as follows:

- Can contain header text lines.
- Can contain column header information. The column header information must immediately precede the data series columns unless the `skiprows` argument (see below) is specified.
- Leftmost column must be the date column.
- Dates must be in a valid date character vector 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` subfolder within the folder *`matlabroot`*`/toolbox/finance`.

The syntax of the function

```
fts = ascii2fts(filename, descrow, colheadrow, skiprows);
```

takes in the data file name (`filename`), the row number where the text for the description field is (`descrow`), the row number of the column header information (`colheadrow`), and the row numbers of rows to be skipped (`skiprows`). For example, rows need to be skipped when there are intervening rows between the column head row and the start of the time series data.

Look at the beginning of the ASCII file `disney.dat` in the `ftsdata` subfolder:

```
Walt Disney Company (DIS)
Daily prices (3/29/96 to 3/29/99)
```

```
DATE     OPEN     HIGH     LOW      CLOSE     VOLUME
3/29/99  33.0625  33.188   32.75    33.063    6320500
3/26/99  33.3125  33.375   32.75    32.938    5552800
3/25/99  33.5     33.625   32.875   33.375    7936000
3/24/99  33.0625  33.25    32.625   33.188    6025400...
```

The command-line

```
disfts = ascii2fts('disney.dat', 1, 3, 2)
```

uses `disney.dat` to create time series object `disfts`. This example

- Reads the text data file `disney.dat`
- Uses the first line in the file as the content of the description field
- Skips the second line
- Parses the third line in the file for column header (or data series names)
- Parses the rest of the file for the date vector and the data series values

The resulting financial time series object looks like this.

```
disfts =

 desc:  Walt Disney Company (DIS)
 freq:  Unknown (0)

 'dates:  (782)'    'OPEN:  (782)'    'HIGH:  (782)'    'LOW:  (782)'
'29-Mar-1996'     [    21.1938]    [    21.6250]   [   21.2920]
'01-Apr-1996'     [    21.1120]    [    21.6250]   [   21.4170]
'02-Apr-1996'     [    21.3165]    [    21.8750]   [   21.6670]
'03-Apr-1996'     [    21.4802]    [    21.8750]   [   21.7500]
'04-Apr-1996'     [    21.4393]    [    21.8750]   [   21.5000]
'05-Apr-1996'     [         NaN]   [         NaN]   [        NaN]
'09-Apr-1996'     [    21.1529]    [    21.5420]   [   21.2080]
'10-Apr-1996'     [    20.7387]    [    21.1670]   [   20.2500]
'11-Apr-1996'     [    20.0829]    [    20.5000]   [   20.0420]
'12-Apr-1996'     [    19.9189]    [    20.5830]   [   20.0830]
'15-Apr-1996'     [    20.2878]    [    20.7920]   [   20.3750]
'16-Apr-1996'     [    20.3698]    [    20.9170]   [   20.1670]
'17-Apr-1996'     [    20.4927]    [    20.9170]   [   20.7080]
'18-Apr-1996'     [    20.4927]    [    21.0420]   [   20.7920]
```

There are 782 data points in this object. Only the first few lines are shown here. Also, this object has two other data series, the `CLOSE` and `VOLUME` data series, that are not shown here. 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`.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `today` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Working with Financial Time Series Objects” on page 12-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Using the Financial Time Series App” on page 13-11
- “Using Time Series to Predict Equity Return” on page 12-25

Visualizing Financial Time Series Objects

In this section...

Introduction

Financial Toolbox software contains the function `chartfts`, which provides a visual representation of a financial time series object. `chartfts` is an interactive charting and graphing utility for financial time series objects. With this function, you can observe time series values on the entire range of dates covered by the time series.

Note Interactive charting is also available from the **Graphs** menu of the user interface. See “Interactive Chart” on page 14-16 for additional information.

Using chartfts

`chartfts` requires a single input argument, `tsobj`, where `tsobj` is the name of the financial time series object you want to explore. Most equity financial time series objects contain four price series, such as opening, closing, highest, and lowest prices, plus an additional series containing the volume traded. However, `chartfts` is not limited to a time series of equity prices and volume traded. It can be used to display any time series data you may have.

To illustrate the use of `chartfts`, use the equity price and volume traded data for the Walt Disney Corporation (NYSE: DIS) provided in the file `disney.mat`:

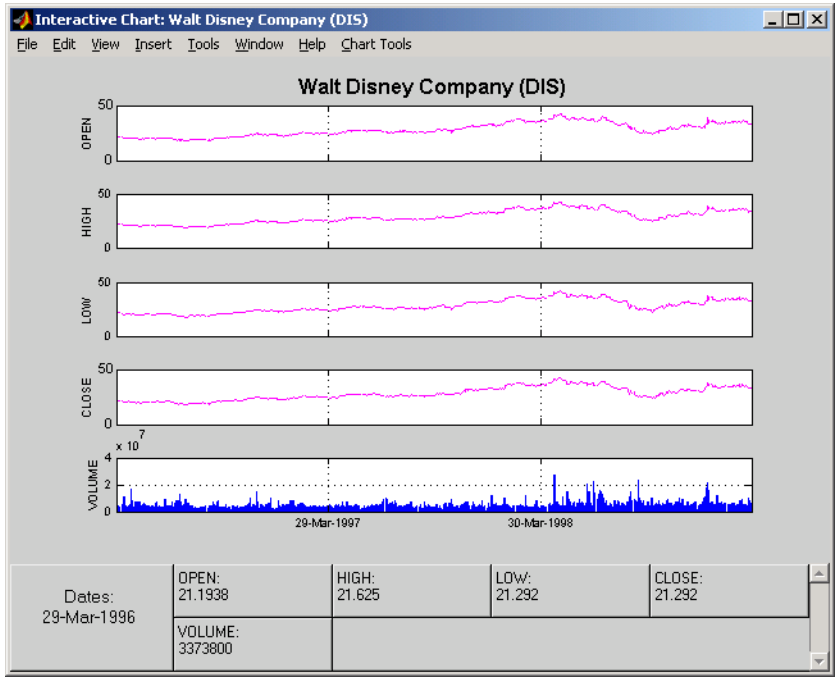
```
load disney.mat

whos
```

```
  Name             Size         Bytes  Class

  dis            782x5          39290  fints object
  dis_CLOSE      782x1           6256  double array
```

```
dis_HIGH        782x1           6256  double array
dis_LOW         782x1           6256  double array
dis_OPEN        782x1           6256  double array
dis_VOLUME      782x1           6256  double array
dis_nv          782x4          32930  fints object
q_dis            13x4           2196  fints object
```

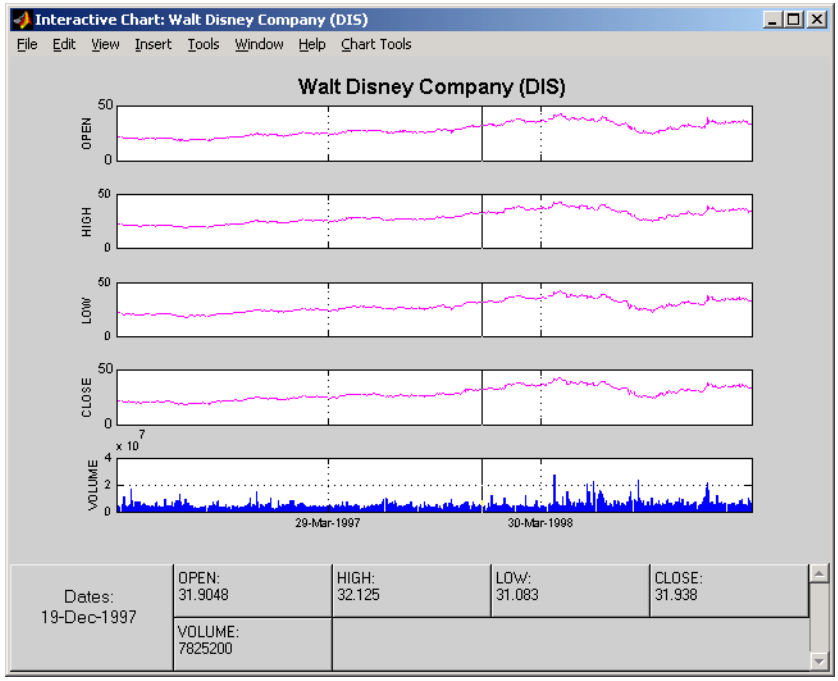
For charting purposes look only at the objects `dis` (daily equity data including volume traded) and `dis_nv` (daily data without volume traded). Both objects contain the series `OPEN`, `HIGH`, `LOW`, and `CLOSE`, but only `dis` contains the additional `VOLUME` series.

Use `chartfts(dis)` to observe the values.

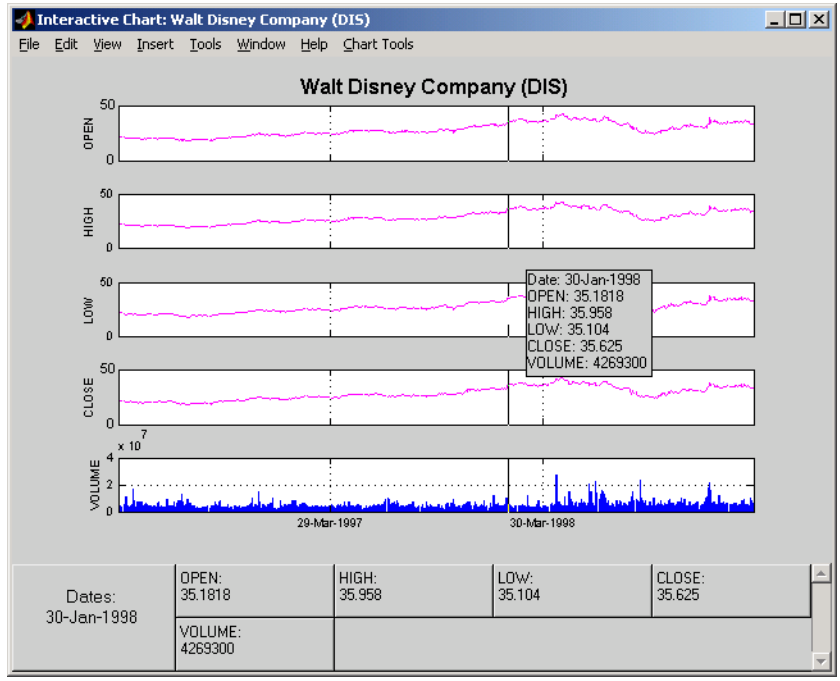


The chart contains five plots, each representing one of the series in the time series object. Boxes indicate the value of each individual plot. The date box is always on the left. The number of data boxes on the right depends upon the number of data series in the time series object, five in this case. The order in which these boxes are arranged (left to right) matches the plots from top to bottom. With more than eight data series in the object, the scroll bar on the right is activated so that additional data from the other series can be brought into view.

Slide the mouse cursor over the chart. A vertical bar appears across all plots. This bar selects the set of data shown in the boxes below. Move this bar horizontally and the data changes accordingly.

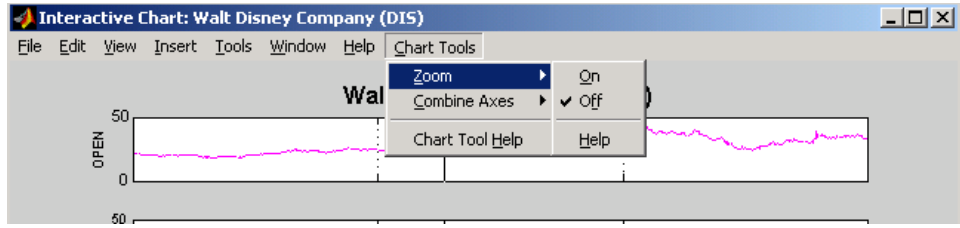


Click the plot. A small information box displays the data at the point where you click the mouse button.



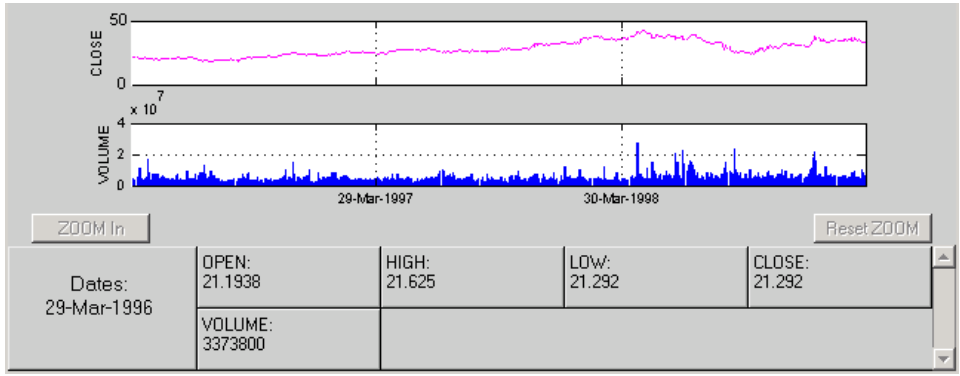
Zoom Tool

The zoom feature of `chartfts` enables a more detailed look at the data during a selected time frame. The Zoom tool is found under the **Chart Tools** menu.

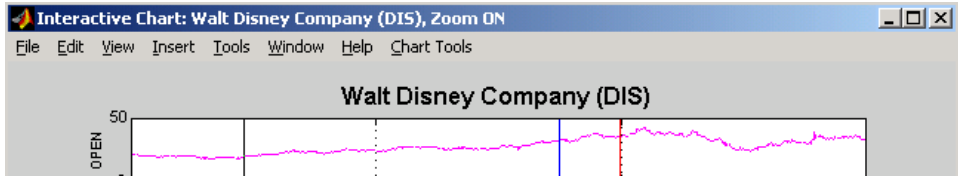


Note Due to the specialized nature of this feature, do not use the MATLAB `zoom` command or **Zoom In** and **Zoom Out** from the **Tools** menu.

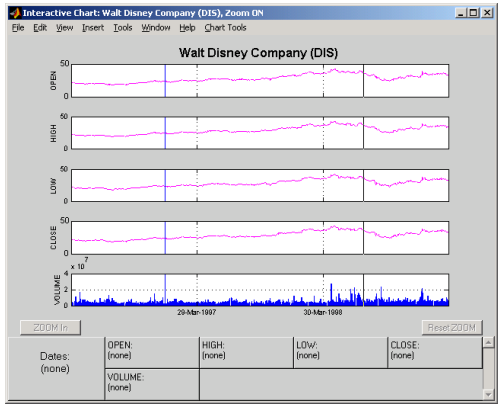
When the feature is turned on, you will see two inactive buttons (**ZOOM In** and **Reset ZOOM**) above the boxes. The buttons become active later after certain actions have been performed.

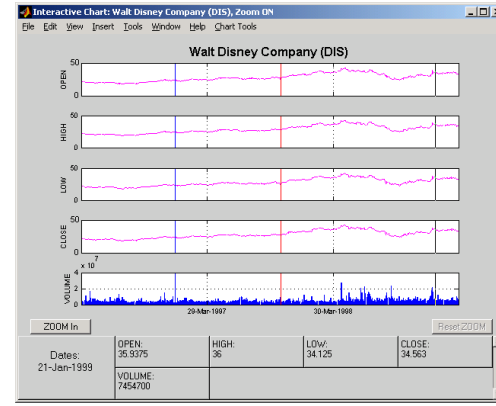


The window title bar displays the status of the chart tool that you are using. With the Zoom tool turned on, you see **Zoom ON** in the title bar in addition to the name of the time series you are working with. When the tool is off, no status is displayed.

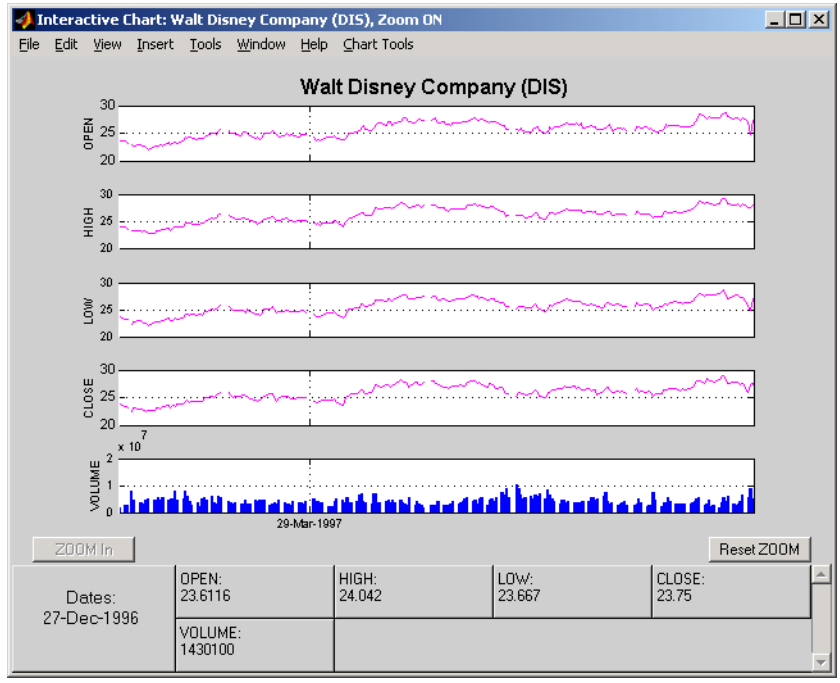


To zoom into the chart, you need to define the starting and ending dates. Define the starting date by moving the cursor over the chart until the desired date appears at the bottom-left box and click the mouse button. A blue vertical line indicates the starting date that you have selected. Next, again 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.





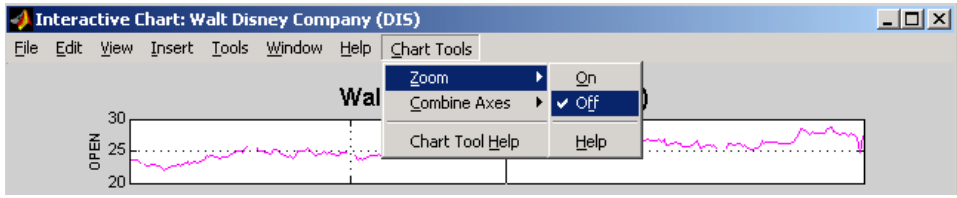
To zoom into the chart, click the **ZOOM In** button.



The chart is zoomed in. 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 off the Zoom tool by going back to the **Chart Tools** menu and choosing **Zoom Off**.



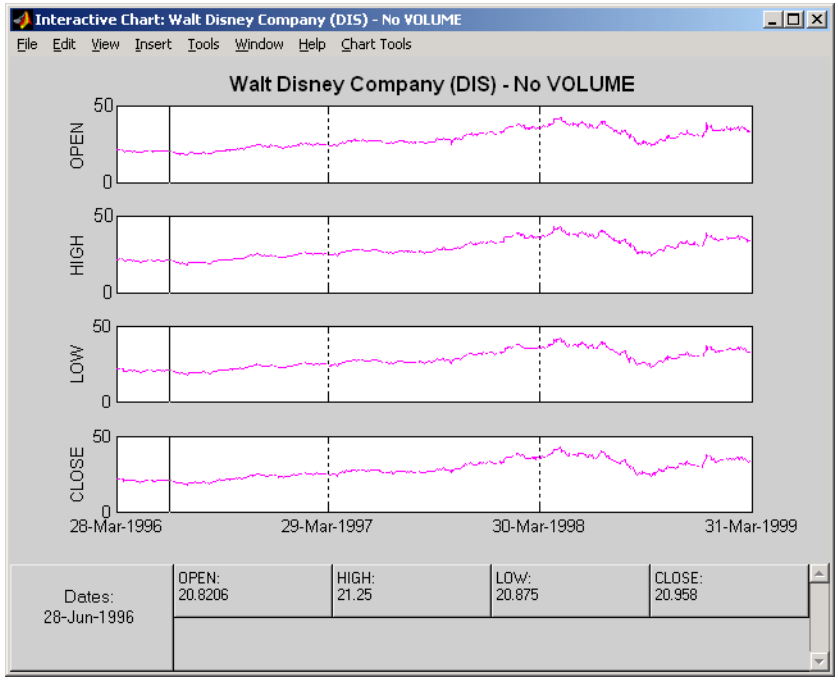
With the tool turned off, the chart stays at the last state that it was in. If you turn it off when the chart is zoomed in, the chart stays zoomed in. If you reset the zoom before turning it off, the chart becomes the original (not zoomed).

Combine Axes Tool

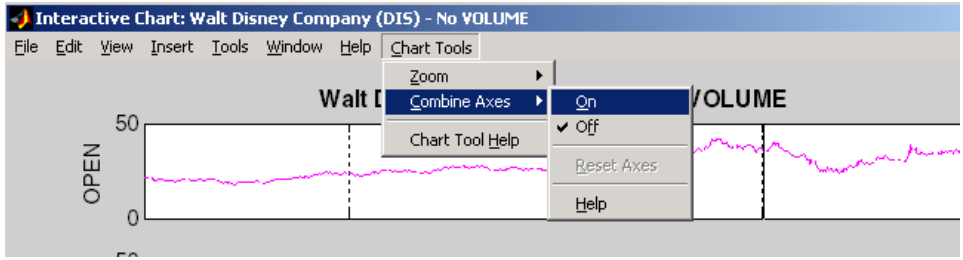
The Combine Axes tool allows you to combine all axes or specific axes into one. With axes combined, you can visually spot any trends that can occur among the data series in a financial time series object.

To illustrate this tool, use `dis_nv`, the financial time series object that does not contain volume traded data:

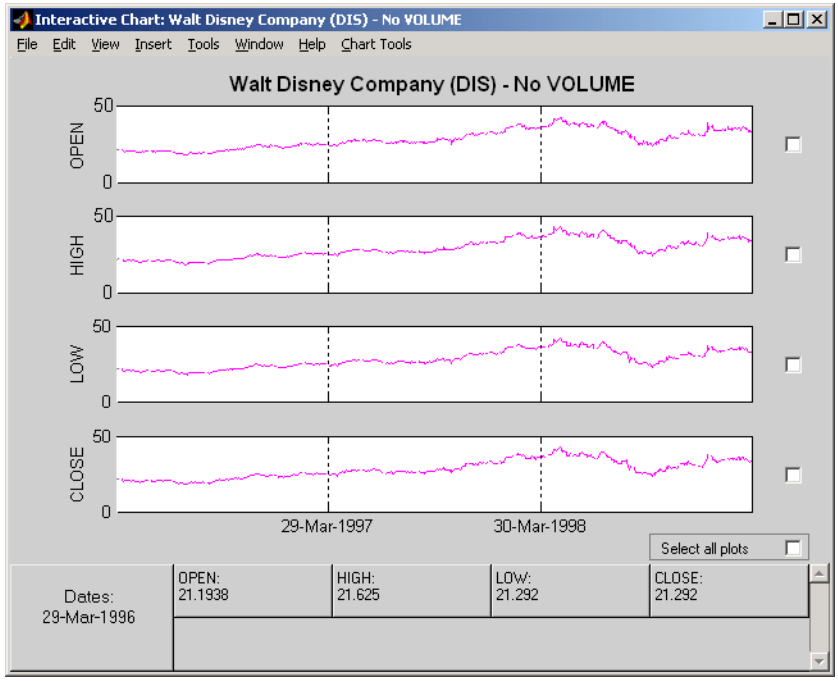
```
chartfts(dis_nv)
```



To combine axes, choose the **Chart Tools** menu, followed by **Combine Axes** and **On**.

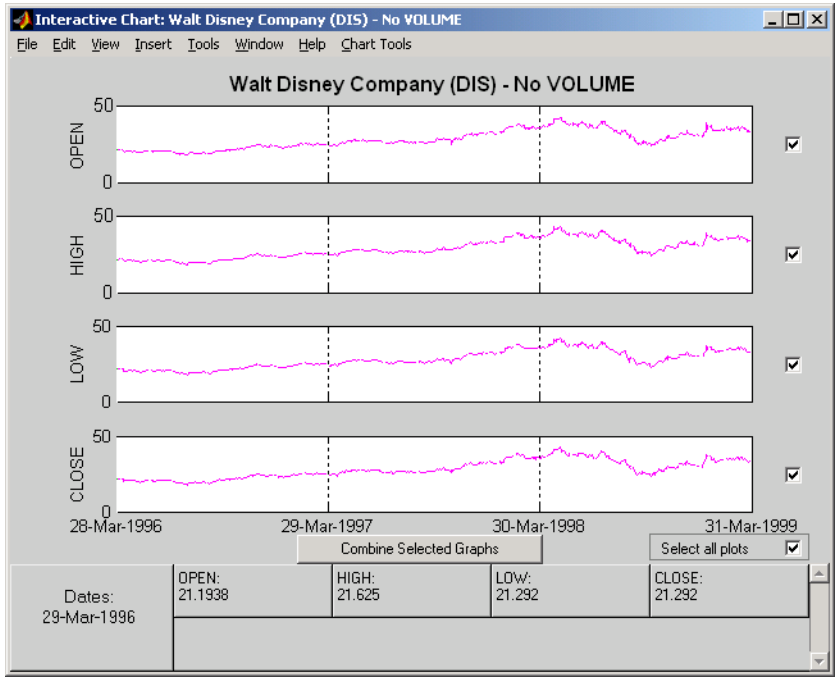


When the Combine Axes tool is on, check boxes appear beside each individual plot. An additional check box enables the combination of all plots.

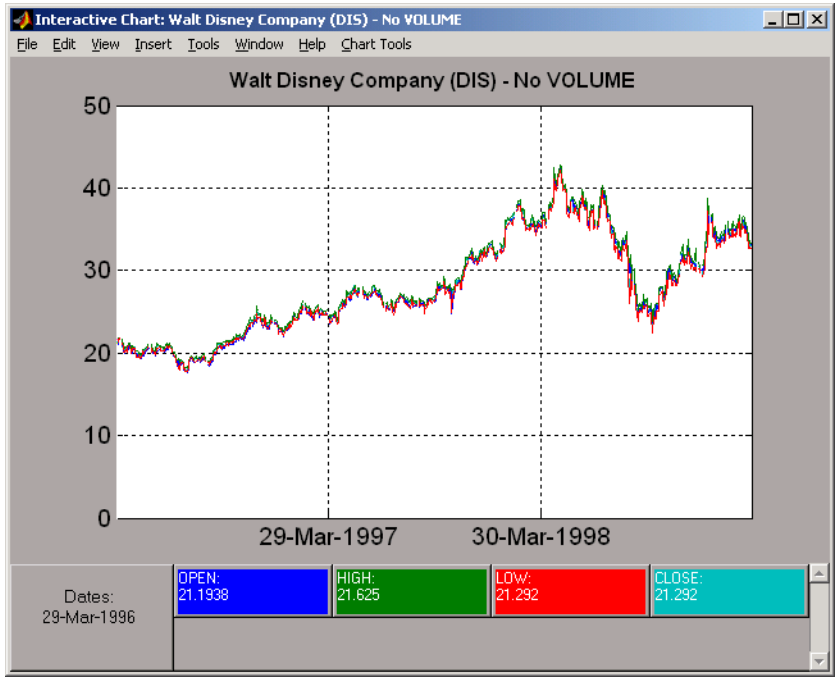


Combining All Axes

To combine all plots, select the **Select all plots** check box.



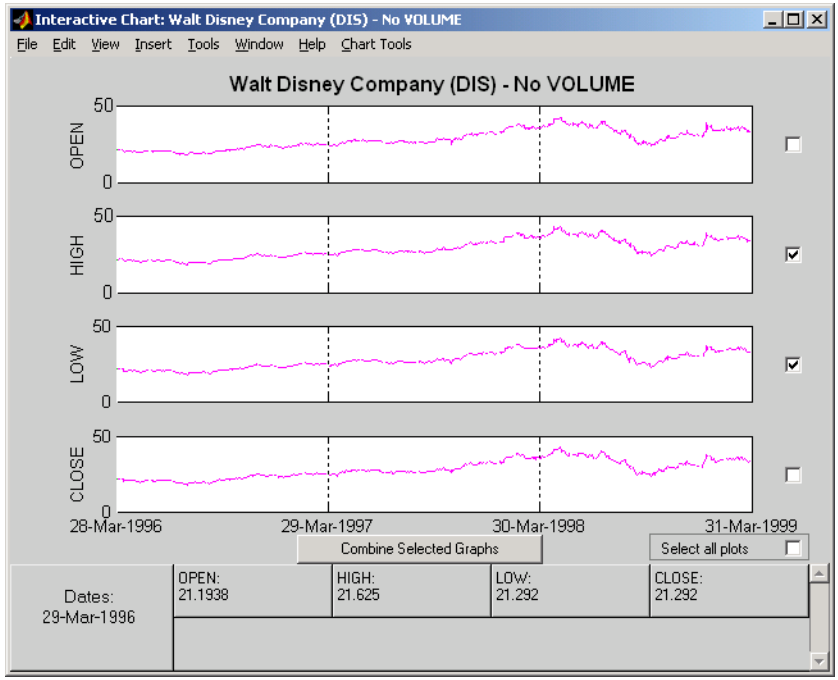
Now click the **Combine Selected Graphs** button to combine the chosen plots. In this case, all plots are combined.



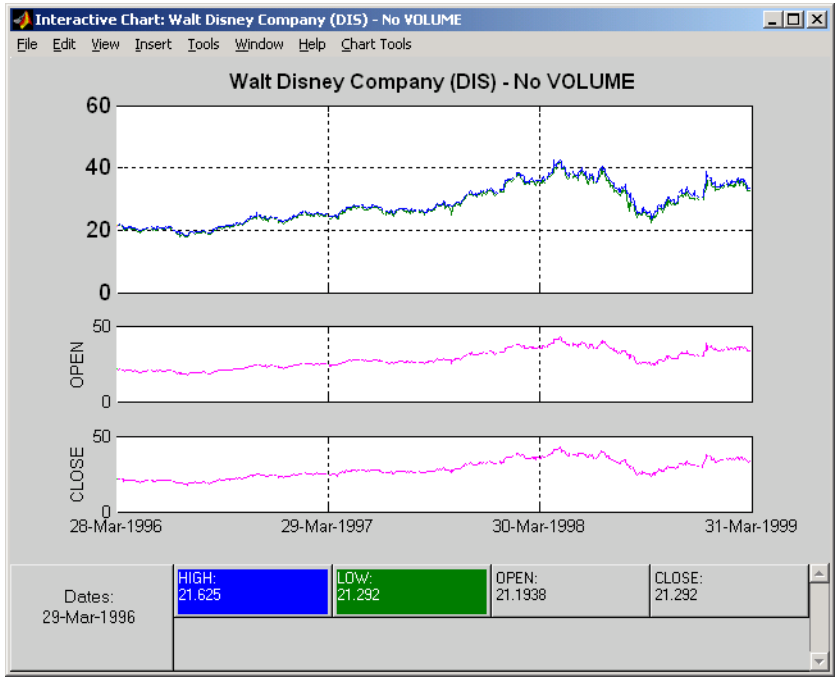
The combined plots have a single plot axis with all data series traced. The background of each data box has changed to the color corresponding to the color of the trace that represents the data series. After the axes are combined, the tool is turned off.

Combining Selected Axes

You can choose any combination of the available axes to combine. For example, combine the `HIGH` and `LOW` price series of the Disney time series. Click the check boxes next to the corresponding plots. The **Combine Selected Graphs** button appears and is active.



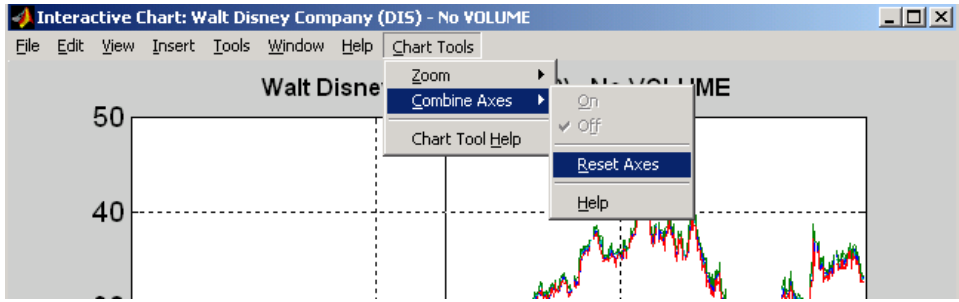
Click the **Combine Selected Graphs** button. The chart with the combined plots looks like the next figure.



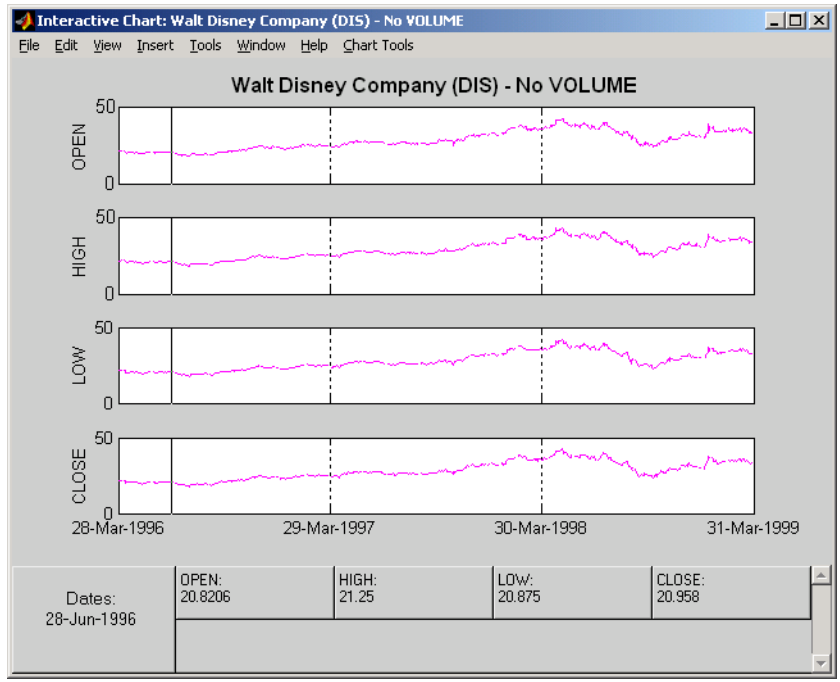
The plot with the combined axes is located at the top of the chart while the remaining plots follow it. The data boxes have also been changed. The boxes that correspond to the combined axes are relocated to the beginning, and the background colors are set to the color of the respective traces. The data boxes for the remaining axes retain their original formats.

Resetting Axes

If you have altered the chart by combining axes, you must reset the axes before you can visualize additional combinations. Reset the axes with the **Reset Axes** menu item under **Chart Tools** > **Combine Axes**. 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.



See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` |

`toannual` | `todaily` | `today` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Creating Financial Time Series Objects” on page 11-3
- “Working with Financial Time Series Objects” on page 12-3
- “Using the Financial Time Series App” on page 13-11
- “Using Time Series to Predict Equity Return” on page 12-25

Using Financial Time Series

Introduction

This section discusses how to manipulate and analyze financial time series data. The major topics discussed include

Much of this information is summarized in the "Using Time Series to Predict Equity Return" on page 12-25.

Working with Financial Time Series Objects

In this section...

Introduction

A financial time series object is 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 character vector (`''`). If you build the object from a text data file using `ascii2fts`, the default is the name of the text data file. The default for the frequency indicator field is `0` (`Unknown` frequency). Objects created from operations can default the setting to `0`. For example, if you decide to pick out values selectively from an object, the frequency of the new object might not be the same as that of the object from which it came.

The date vector `dates` does not have a default set of values. When you create an object, you have to supply the date vector. You can change the date vector afterward but, at object creation time, you must provide a set of dates.

The final component of a financial time series object is one or more data series vectors. If you do not supply a name for the data series, the default name is `series1`. If you have multiple data series in an object and do not supply the names, the default is the name series followed by a number, for example, `series1`, `series2`, and `series3`.

Data Extraction

Here is an exercise on how to extract data from a financial time series object. As mentioned before, you can think of the object as a MATLAB structure. Highlight each line in the exercise in the MATLAB Help browser, press the right mouse button, and select **Evaluate Selection** to execute it.

To begin, create a financial time series object called `myfts`:

```
dates = (datenum('05/11/99'):datenum('05/11/99')+100)';
data_series1 = exp(randn(1, 101))';
data_series2 = exp(randn(1, 101))';
data = [data_series1 data_series2];
myfts = fints(dates, data);
```

The `myfts` object looks like this:

```
myfts =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (101)'    'series1:  (101)'    'series2:  (101)'
    '11-May-1999'      [         2.8108]    [         0.9323]
    '12-May-1999'      [         0.2454]    [         0.5608]
    '13-May-1999'      [         0.3568]    [         1.5989]
    '14-May-1999'      [         0.5255]    [         3.6682]
    '15-May-1999'      [         1.1862]    [         5.1284]
    '16-May-1999'      [         3.8376]    [         0.4952]
    '17-May-1999'      [         6.9329]    [         2.2417]
    '18-May-1999'      [         2.0987]    [         0.3579]
    '19-May-1999'      [         2.2524]    [         3.6492]
    '20-May-1999'      [         0.8669]    [         1.0150]
    '21-May-1999'      [         0.9050]    [         1.2445]
    '22-May-1999'      [         0.4493]    [         5.5466]
    '23-May-1999'      [         1.6376]    [         0.1251]
    '24-May-1999'      [         3.4472]    [         1.1195]
    '25-May-1999'      [         3.6545]    [         0.3374]...
```

There are more dates in the object; only the first few lines are shown here.

Note The actual data in your `series1` and `series2` differs 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 `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 character vector 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` 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.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Using the Financial Time Series App” on page 13-11
- “Using Time Series to Predict Equity Return” on page 12-25

Financial Time Series Operations

Several MATLAB functions have been overloaded to work with financial time series objects. The overloaded functions include basic arithmetic functions such as addition, subtraction, multiplication, and division and other functions such as arithmetic average, filter, and difference. Also, specific methods have been designed to work with the financial time series object. For a list of functions grouped by type, enter

```
help ftseries
```

at the MATLAB command prompt.

Basic Arithmetic

Financial time series objects permit you to do addition, subtraction, multiplication, and division, either on the entire object or on specific object fields. This is a feature that MATLAB structures do not allow. You cannot do arithmetic operations on entire MATLAB structures, only on specific fields of a structure.

You can perform arithmetic operations on two financial time series objects as long as they are compatible. (All contents are the same except for the description and the values associated with the data series.)

Note *Compatible* time series are not the same as *equal* time series. Two time series objects are equal when everything but the description fields are 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 4 data series:

```
submtx = myfts1 - randn(20, 4)

submtx =

desc:  DJI30MAR94.dat
freq:  Daily (1)

'dates: (20)'  'Open: (20)' 'High: (20)' 'Low: (20)' 'Close: (20)'
'04-Mar-1994'  [ 3831.33]   [ 3867.75]   [ 3802.10]  [ 3832.63]
'07-Mar-1994'  [ 3853.39]   [ 3883.74]   [ 3824.45]  [ 3857.06]
'08-Mar-1994'  [ 3858.35]   [ 3880.84]   [ 3823.51]  [ 3851.22]
'09-Mar-1994'  [ 3853.68]   [ 3872.90]   [ 3816.53]  [ 3851.92]
'10-Mar-1994'  [ 3853.72]   [ 3866.20]   [ 3802.44]  [ 3831.17]...
```

Arithmetic Operations with Differing Data Series Names

Arithmetic operations on two objects that have the same size but contain different data series names require the function `fts2mat`. This function extracts the values in an object and puts them into a matrix or vector, whichever is appropriate.

To see an example, create another financial time series object the same size as `myfts1` but with different values and data series names:

```
newfts2 = fints(myfts1.dates, fts2mat(myfts1/10000),...
{'Rat1','Rat2', 'Rat3','Rat4'}, 1, 'New FTS')
```

If you attempt to add (or subtract, and so on) this new object to `myfts1`, an error indicates that the objects are not identical. Although they contain the same dates, number of dates, number of data series, and frequency, the two time series objects do not have the same data series names. Use `fts2mat` to bypass this problem:

```
addother = myfts1 + fts2mat(newfts2);
```

This operation adds the matrix that contains the contents of the data series in the object `newfts2` to `myfts1`. You should carefully consider the effects on your data before deciding to combine financial time series objects in this manner.

Other Arithmetic Operations

In addition to the basic arithmetic operations, several other mathematical functions operate directly on financial time series objects. These functions include exponential (`exp`), natural logarithm (`log`), common logarithm (`log10`), and many more.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Using the Financial Time Series App” on page 13-11
- “Using Time Series to Predict Equity Return” on page 12-25

Data Transformation and Frequency Conversion

The data transformation and the frequency conversion functions convert a data series into a different format.

Data Transformation Functions

Function	Purpose
`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
`smoothts`	Smooth data
`tsmovavg`	Moving average

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:

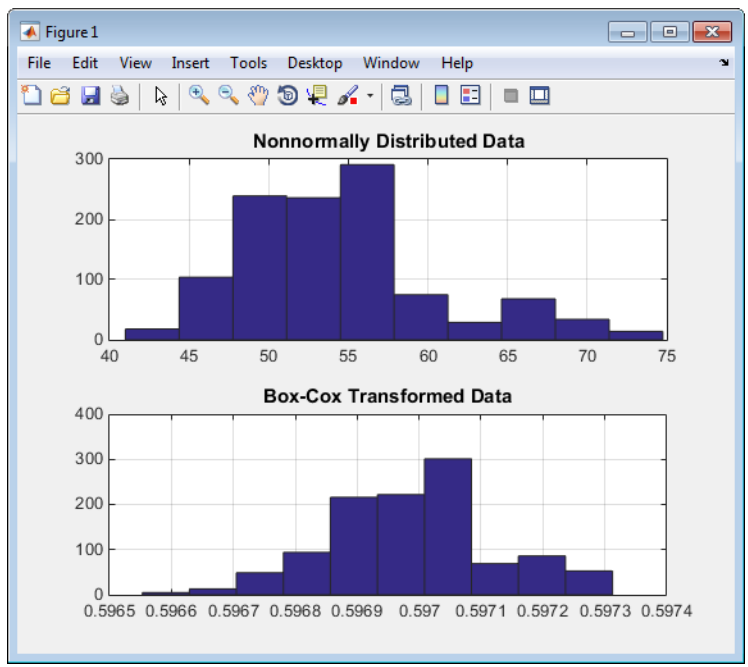
```
f_whrl = fillts(whrl);
```

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 and the nonnormally distributed `f_whrl`:

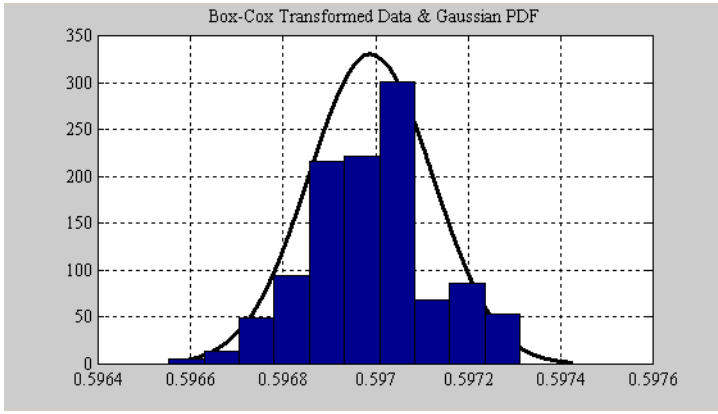
```
subplot(2, 1, 1);
hist(f_whrl.Close);
grid; title('Nonnormally Distributed Data');
subplot(2, 1, 2);
hist(bc_whrl.Close);
grid; title('Box-Cox Transformed Data');
```



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 toward the left (not normally distributed). The bar chart on the bottom is less skewed to the left. If you plot a Gaussian probability distribution function (PDF) with similar mean and standard deviation, the distribution of the transformed data is very close to normal (Gaussian).

When you examine the contents of the resulting object `bc_whrl`, you find an identical object to the original object `whrl` but the contents are the transformed data series. If you have the Statistics and Machine Learning Toolbox software, you can generate a Gaussian PDF with mean and standard deviation equal to those of the transformed data series and plot it as an overlay to the second bar chart. In the next figure, you can see that it is an approximately normal distribution.



Overlay of Gaussian PDF

The next example uses the `smoothts` function to smooth a time series.

To begin, transform `ibm9599.dat`, a supplied data file, into a financial time series object:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
```

Fill the missing data for holidays with data interpolated using the `fillts` function and the `Spline` fill method:

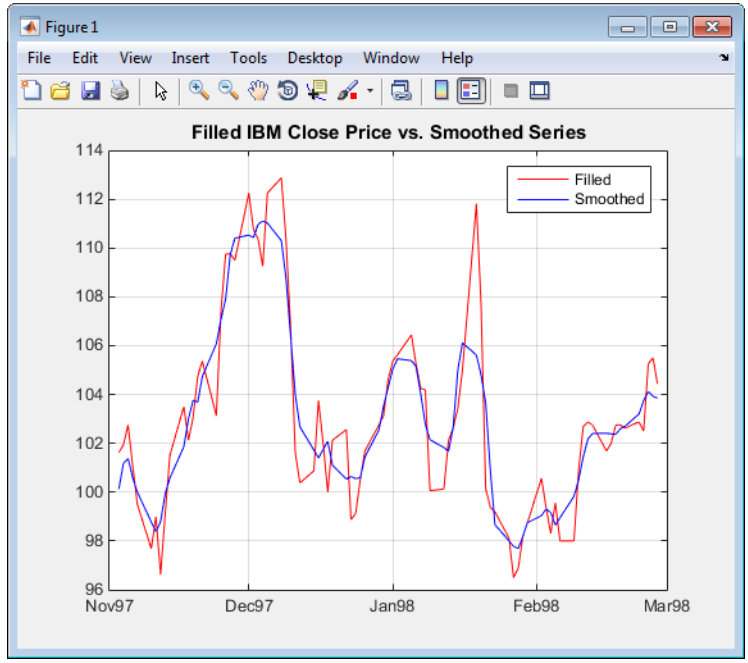
```
f_ibm = fillts(ibm, 'Spline');
```

Smooth the filled data series using the default Box (rectangular window) method:

```
sm_ibm = smoothts(f_ibm);
```

Now, plot the original and smoothed closing price series for IBM stock:

```
plot(f_ibm.CLOSE('11/01/97::02/28/98'), 'r')
datetick('x', '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')
```



Smoothed Data Series

These examples give you an idea of what you can do with a financial time series object. This toolbox provides some MATLAB functions that have been overloaded to work directly with these objects. The overloaded functions are those most commonly needed to work with time series data.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Using the Financial Time Series App” on page 13-11
- “Using Time Series to Predict Equity Return” on page 12-25

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 character vector on page 12-17, a cell array on page 12-0 of date character vectors, a date character vector range on page 12-18, or normal integer on page 12-20 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 character vectors.

When indexing by date character vector, note that

- Each date character vector 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, and so on, whenever the data has gone through a frequency conversion.

Indexing with Date Character Vectors

With date character vector indexing, you get the values in a financial time series object for a specific date using a date character vector as the index into the object. Similarly, if you want values for multiple dates in the object, you can put those date character vectors into a cell array of character vectors 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 Character Vector Range

A financial time series is unique because it allows you to index into the object using a date character vector range. A date character vector range consists of two date character vector separated by two colons (`::`). In MATLAB this separator is called the double-colon operator. An example of a MATLAB date character vector 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 character vector 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 character vector 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 character vector indexing, enclose the date and time character vectors 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
```

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Using the Financial Time Series App” on page 13-11
- “Using Time Series to Predict Equity Return” on page 12-25

Using Time Series to Predict Equity Return

This example shows 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.

Step 1. Load the data.

The data for this demonstration is found in the MAT-file `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`

```
load predict_ret_data.mat
```

Step 2. Create Financial Time Series objects.

It is useful to work with financial time series objects rather than with the vectors now in the workspace. By using objects, you can easily keep track of the dates. Also, you can manipulate the data series based on dates because a time series object keeps track of the administration of a 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.

Step 3. 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 (`dadj1`) that reflects this adjustment factor. Then create the series (`dadj2`) that adjusts 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. Combining the two objects (`dadj1` and `dadj2`) 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 (`dadj3`) from the adjustment data. Use the dates from the stock price data to make the dates of the adjustment the same.

```
dadj1       = d0;
dadj1.dates = dadj1.dates-1;

dadj2              = d0;
dadj2.Dividends    = 0;
dadj2              = fillts(dadj2,'linear','12/31/98');
dadj2('12/31/98') = 0;

dadj3 = [dadj1; dadj2];
dadj3 = fillts(dadj3, 'linear', t0.dates);
```

Step 4. 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)));
```

Step 5. Create return series.

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.

Step 6. 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. Get all the dates common to the return series calculated and the explanatory (metric) data and then combine the contents of the two series that have dates in common into a new time series (`regts0`). Remove the contents of the new time series that are not finite to create time series (`regts1`).

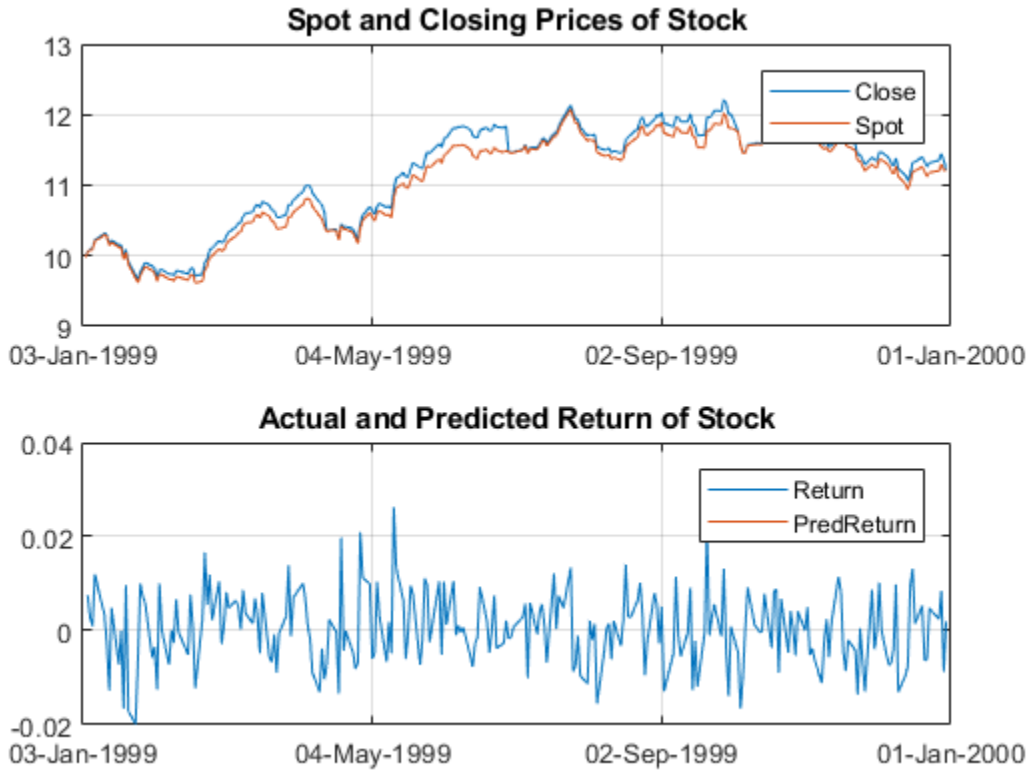
Place the data to be regressed into a matrix using the function `fts2mat`. The first column of the matrix corresponds to the values of the first data series in the object, the second column to the second data series, and so on. In this case, the first column is regressed against the second and third column. 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`.

```
x1 = todaily(x0);
x1.Const = 1;

dcommon = intersect(tret.dates, x1.dates);
regts0  = [tret(datestr(dcommon)), x1(datestr(dcommon))];

finite_regts0 = find(all(isfinite( fts2mat(regts0)), 2));
regts1        = regts0( finite_regts0 );

DataMatrix = fts2mat(regts1);
XCoeff     = DataMatrix(:, 2:3) \ DataMatrix(:, 1);

RetPred = DataMatrix(:,2:3) * XCoeff;
tret.PredReturn(datestr(regts1.dates)) = RetPred;
```

Step 7. Plot the results.

Plot the results in a single 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');
```



Step 8. Calculate the dividend rate.

Calculate the dividend rate from the stock price data by dividing the dividend payments by the corresponding closing stock prices. 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. Use `fillts` to create a new time series (`t1`)

containing the missing date from the original data series. Then set the frequency indicator to daily.

```
datestr(d0.dates, 2)

ans = 4x8 char array
    '04/15/99'
    '06/30/99'
    '10/02/99'
    '12/30/99'



t1 = fillts(t0,'nearest',d0.dates);
t1.freq = 'd';

tdr = d0./fts2mat(t1.Close(datestr(d0.dates)))   % Calculate the dividend rate


tdr =

    desc:  Inc
    freq:  Unknown (0)

    'dates:  (4)'    'Dividends:  (4)'
    '15-Apr-1999'    [         0.0193]
    '30-Jun-1999'    [         0.0305]
    '02-Oct-1999'    [         0.0166]
    '30-Dec-1999'    [         0.0134]
```

You can find a file for this example program in the folder *matlabroot* `/toolbox/finance/findemos` on your MATLAB® path. The file is `predict_ret.m`.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “Creating Financial Time Series Objects” on page 11-3

- “Visualizing Financial Time Series Objects” on page 11-16
- “Working with Financial Time Series Objects” on page 12-3
- “Using the Financial Time Series App” on page 13-11

Financial Time Series App

What Is the Financial Time Series App?

The Financial Time Series app enables you to create and manage financial time series (`fints`) objects. The Financial Time Series app interoperates with the Financial Time Series Graphical User Interface (`ftsgui`) and Interactive Chart (`chartfts`). In addition, you can use Datafeed Toolbox™ software to connect to external data sources.

A financial time series object minimally consists of:

- `Desc`, which is the description field.
- `Freq`, which is a frequency indicator field.
- `Dates`, which is a date vector field. If the date vector incorporates time-of-day information, the object contains an additional field named `times`.
- In addition, you can have at least one data series vector. You can specify names for any data series vectors. If you do not specify names, the object uses the default names `series1`, `series2`, `series3`, and so on.

In general, the workflow for using the Financial Time Series app is:

1. Acquire data.
2. Create a variable.
3. Convert the variable to `fints`.
4. Convert `fints` to a MATLAB double object.

To obtain the data for the Financial Time Series app, you need to use a MATLAB double object or a financial time series (`fints`) object. You can use previously stored internal data on your computer or you can connect to external data sources using Datafeed Toolbox software.

Note You must obtain a license for these products from MathWorks before you can use Datafeed Toolbox.

After creating a financial time series object, you can use the Financial Time Series app to change the characteristics of the time series object, including merging with other financial time series objects, removing rows or columns, and changing the frequency. You can also use the Financial Time Series app to generate various forms of plotted output and you can reconvert a `fints` object to a MATLAB double-precision matrix.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- "Getting Started with the Financial Time Series App" on page 13-4
- "Loading Data with the Financial Time Series App" on page 13-7
- "Using the Financial Time Series App" on page 13-11
- "Using the Financial Time Series App with GUIs" on page 13-19
- "Creating Financial Time Series Objects" on page 11-3
- "Visualizing Financial Time Series Objects" on page 11-16
- "Working with Financial Time Series Objects" on page 12-3

Getting Started with the Financial Time Series App

To start the Financial Time Series app:

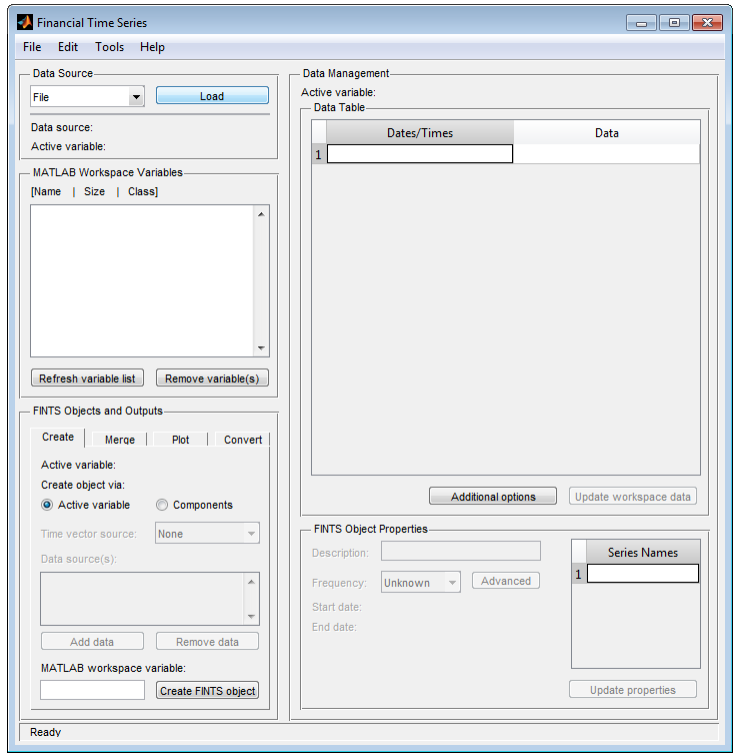
1. On the MATLAB desktop toolstrip, click the **Apps** tab and in the apps gallery, under **Computational Finance**, click **Financial Time Series**.

   The Financial Time Series app opens. For an overview of the Financial Time Series app, see “What Is the Financial Time Series App?” on page 13-2.

2. To load data with the Financial Time Series app, see “Loading Data with the Financial Time Series App” on page 13-7.

   If you plan to load data from Datafeed Toolbox software, ensure that you have a license. For more information on this toolbox, see the Datafeed Toolbox documentation.

3. For more information on the tasks supported by the Financial Time Series app, see “Using the Financial Time Series App” on page 13-11 and “Using the Financial Time Series App with GUIs” on page 13-19.



See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “What Is the Financial Time Series App?” on page 13-2
- “Loading Data with the Financial Time Series App” on page 13-7
- “Using the Financial Time Series App” on page 13-11
- “Using the Financial Time Series App with GUIs” on page 13-19
- “Creating Financial Time Series Objects” on page 11-3

- “Visualizing Financial Time Series Objects” on page 11-16
- “Working with Financial Time Series Objects” on page 12-3

Loading Data with the Financial Time Series App

In this section...

Overview

The **Data source** pane in the Financial Time Series app lets you do the following:

- Load data you previously obtained and stored in a file.
- View data contained within the MATLAB workspace.

Obtaining Internal Data

You can use the Financial Time Series app to load data from files previously stored on your computer. The types of data files you can load are as follows:

- MATLAB `.mat` files, with or without `fints` objects
- ASCII text files (`.dat` or `.txt` suffixes)
- Excel `.xls` files

To obtain internal data:

1. From the Financial Time Series app, select **File** > **Load** > **File** to open the Load a MAT, ASCII, .XLS File dialog box.
2. Select the data you want to load into the Financial Time Series app.

    - If you load a MATLAB MAT-file, the variables in the file are placed into the MATLAB workspace. The **MATLAB Workspace Variables** list box shows the variables that have been added to the workspace. For example, if you load the file `disney.mat`, which is distributed with the toolbox, the **MATLAB Workspace Variables** list box displays the variables in that MAT-file.

Note The Financial Time Series app automatically generates a line plot for each workspace variable unless you disable this feature by resetting the default action under **File > Preferences > Generate line plot on load**.

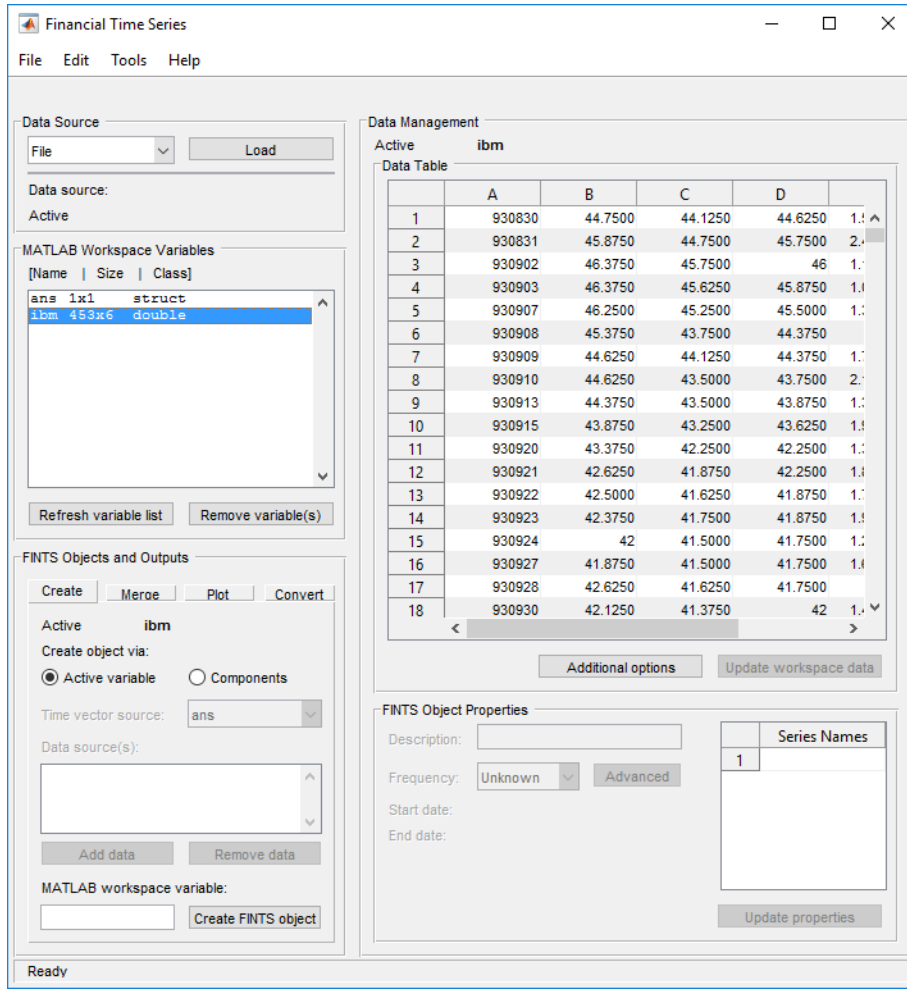
- If you load a `.dat` or an ASCII `.txt` file, the ASCII File Parameters dialog box opens. Use this dialog box to transform a text data file into a MATLAB financial time series `fints` object. The format for the ascii data must be:
  - Dates must be in a valid date character vector format:
    - `'ddmmmyy'` or `'ddmmmyyyy'`
    - `'mm/dd/yy'` or `'mm/dd/yyyy'`
    - `'dd-mmm-yy'` or `'dd-mmm-yyyy'`
    - `'mmm.dd,yy'` or `'mmm.dd,yyyy'`
  - Time information must be in `'hh:mm'` format.
  - Each column must be separated either by spaces or a tab.

  For more information on converting ascii data to a `fints` object, see `ascii2fts`.
- If you load an Excel `.xls` file, the Excel File Parameters dialog box opens. Use this dialog box to transform Excel worksheet data into a MATLAB financial time series (`fints`) object.

3 From the Financial Time Series app, select **File > Save** to save the data you loaded from an internal file.

Viewing the MATLAB Workspace

The **MATLAB Workspace Variables** list box displays all existing MATLAB workspace variables. Double-click any variable to display the data in the **Data Table**. You can only display financial time series (`fints`) objects, MATLAB doubles, and cell arrays of double data in the **Data Table**.



In addition, you can click **Refresh variable list** to refresh the **MATLAB Workspace Variables** list box. You need to refresh this list periodically because it is refreshed automatically only for operations performed with the Financial Time Series app, not for operations performed within MATLAB itself.

Click **Remove variable(s)** to remove variable from the **MATLAB Workspace Variables** list and from the MATLAB workspace.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “What Is the Financial Time Series App?” on page 13-2
- “Getting Started with the Financial Time Series App” on page 13-4
- “Using the Financial Time Series App” on page 13-11
- “Using the Financial Time Series App with GUIs” on page 13-19
- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Working with Financial Time Series Objects” on page 12-3

Using the Financial Time Series App

In this section...

Creating a Financial Time Series Object

Using the **Create** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can create a financial time series (`fints`) object from one or more selected variables.

Note When you first start the Financial Time Series app, the **Create** tab appears on top, unless you reset the default using **File > Preferences > Show Create tab when ftstool starts**.

To create a financial time series (`fints`) object from one or more selected variables:

1. Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox software or an internal data source using **File > Load > File**.
2. Select one or more variables from the **MATLAB Workspace Variables** list.
3. Click the **Create** tab and then click **Active variable**.

   When combining multiple variables, you can type a new variable name for the combined variables in the **MATLAB workspace variable** box. The new variable name is added to the **MATLAB Workspace Variables** list. (If you do not choose a name for the **MATLAB workspace variable**, the Financial Time Series app uses the default name `myFts`.)

4 Click **Create FINTS object** to display the result in the **Data Table**.

Merge Financial Time Series Objects

Using the **Create** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can create a new financial time series object by merging (joining) multiple existing financial time series objects.

Note When you first start the Financial Time Series app, the **Create** tab appears on top, unless you reset the default using **File** > **Preferences**.

To create a financial time series (`fints`) object by merging multiple existing financial time series objects:

1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox software or an internal data source using **File** > **Load** > **File**.

2 To merge multiple existing financial time series objects, click the **Create** tab, click **Components**, and then select a value for the **Time vector source** and one or more items from the **Data sources** list.

   Note You can merge at once multiple financial time series objects. For more information on merging `fints` objects, see `merge`.

3 Click **Create FINTS object** to display the result in the **Data Table**.

Converting a Financial Time Series Object to a MATLAB Double-Precision Matrix

Using the **Convert** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can convert a financial time series (`fints`) object to a MATLAB double-precision matrix.

To create a financial time series object from one or more selected variables:

1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox software or an internal data source using **File** > **Load** > **File**.

2 Select a variable from the **MATLAB Workspace Variables** list box.

3 Click the **Convert** tab and then determine whether to include or exclude dates in the conversion by clicking **Include dates** or **Exclude dates**.

4 Type a variable name in the **Output variable name** box. (If you do not choose a variable name, the Financial Time Series app uses the default name `myDbl`.)

5 Click **Convert FINTS to double matrix**. (This operation is equivalent to performing `fts2mat` on a financial time series object.)

Plotting the Output in Several Formats

Using the **Plot** tab in the **FINTS Objects and Outputs** pane for the Financial Time Series app, you can create several forms of plotted output by using a selection list. You can create four types of bar charts, candle plots, high-low plots, line plots, and interactive charts (the latter is created by using the interoperation of the Financial Time Series app with the function `chartfts`).

The set of plots supported by the Financial Time Series app are identical to the set provided by the **Graphs** menu of the Financial Time Series GUI. (See "Graphs Menu" on page 14-14.) You can find more detailed information for the supported plots by consulting the reference page for each individual type of plot.

To create a plotted output:

1 Load data into the Financial Time Series app from either an external data source using Datafeed Toolbox software or an internal data source using **File** > **Load** > **File**.

2 Select a variable from the **MATLAB Workspace Variables** list box or select data from the **Data Table**.

3 Click the **Plot** tab and indicate whether you are plotting based on a workspace variable or data from the **Data Table**.

4 From the **Type** drop-down list, select the type of plot.

5 Click **Plot**. The plot is displayed.

Note If the selected workspace variable that you are plotting is not a `fints` object, a `fints` object is created when you click **Plot**. The new `fints` object uses the name designated by the **MATLAB workspace variable** box on the **Create** tab.

Viewing Data for a Financial Time Series Object in the Data Table

Once a financial time series (`fints`) object is created, the Financial Time Series app **Data Table** displays user-designated data, including financial time series objects, MATLAB double-precision variables, and cell arrays of doubles.

When displaying double variables (or a cell array of doubles) in the **Data Table**, the column headings for a double variable or cell array of doubles displayed in the **Data Table** are labeled **A**, **B**, **C**, and so on.

Overwriting Data in the Data Table Display

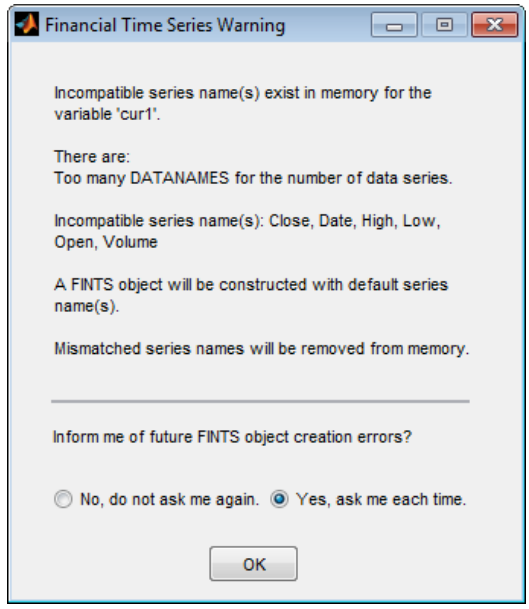
If you use the command line to overwrite data previously retrieved using Datafeed Toolbox software, two events could occur:

- If the new data contains the same number of columns as before, the headers remain unchanged when you attempt to create a financial time series (`fints`) object using the modified data.
- If the data contains a different number of columns, a warning dialog box appears.

For example, assume that you use Datafeed Toolbox software to obtain `Close`, `Date`, `High`, `Low`, `Open`, and `Volume` data for the equity GOOG. You store the data in the MATLAB workspace with the variable name `cur1`. From the command line, if you redefine the variable `cur1`, eliminating the second column (`Close`)

```
cur1(:,2) = [ ]
```

and then return to the Financial Time Series app and attempt to create a financial time series object, a warning dialog box appears.



Modifying Data for a Financial Time Series Object in the Data Table

The Financial Time Series app lets you update your data displayed in the **Data Table** by adding or removing rows or columns.

Note Modifying data in the **Data Table** will not update the MATLAB workspace variable. To update the workspace variable after modifying the **Data Table**, click **Update workspace variable**.

Adding and Removing Rows

To add a row of data displayed in the **Data Table**:

1. Select a row from the **Data Table** display where you want to add a row. Click **Additional options** to open the Data Table Options dialog box.
2. Click **Add row**. The default is to add up the row. To add a row down, select **Insertion option** and then click **Add down**. In addition, you can select the **Insertion option** of **Date** to designate a specific date. (If a date is not specified, the

added row contains a date that is chronologically in order with respect to the initial row.)

When you add rows, the **Data Table** display is immediately updated.

To remove a row of data from the **Data Table**:

1. Select one or more rows in the **Data Table** display that you want to remove. Click **Additional options** to open the Data Table Options dialog box.
2. Click **Remove row(s)**. The default is to remove the selected rows. In addition, to remove selected rows, select **Removal options** and then select other options for row removal from the **Remove rows** list box. You can specify a **Start** and **End** date or you can click the **Non-uniform range setting** option to designate a range.

   When you remove rows, the **Data Table** display is updated immediately.

Adding and Removing Columns

To add a column of data displayed in the **Data Table**:

1. Select a column from the **Data Table** display where you want to add a column. Click **Additional options** to open the Data Table Options dialog box.
2. Click **Add column**. The default is to add the column to the left of the selected column.

   Note For time series objects, you cannot add a column to the left of the `Date/Times` column; there is no restriction for double data.

   To add a column to the right, select **Insertion option** and then click **Add right**. In addition, you can use the **Insertion option** of **New Column Name** to designate a specific column name. (If a **New Column Name** is not specified, an added column contains a column name of `series1`, `series2`, and so on.)

   When you add columns, the **Data Table** display is updated immediately.

To remove a column of data displayed in the **Data Table**:

1. Select one or more columns in the **Data Table** display that you want to remove. Click **Additional options** to open the Data Table Options dialog box.

2 Click **Remove column(s)**. The default is to remove the selected rows. In addition, to remove selected columns, select **Removal options** and then select columns for removal from the **Remove columns** list box.

When you remove columns, the **Data Table** display is updated immediately.

Viewing and Modifying the Properties for a FINTS Object

The **FINTS Object Properties** pane in the Financial Time Series app lets you modify financial time series (`fints`) object properties. This area becomes active whenever the **Data Table** displays a financial time series object.

To modify the properties for a `fints` object:

1 After you create a `fints` object, double-click the object name in the **MATLAB Workspace Variables** list box to open the **Data Table** and display the `fints` object properties.

2 Click to modify the **Description**, **Frequency**, or **Series Names** fields.

The **Frequency** drop-down list supports the following conversion functions:

Function	New Frequency
`toannual`	Annual
`todaily`	Daily
`tomonthy`	Monthly
`toquarterly`	Quarterly
`tosemi`	Semiannually
`toweekly`	Weekly

3 Click **Update properties** to save the changes. This action also updates the associated workspace variable.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “What Is the Financial Time Series App?” on page 13-2
- “Getting Started with the Financial Time Series App” on page 13-4
- “Using the Financial Time Series App with GUIs” on page 13-19
- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Working with Financial Time Series Objects” on page 12-3

Using the Financial Time Series App with GUIs

The Financial Time Series app works with Datafeed Toolbox software to load data. In addition, the Financial Time Series app interoperates with `chartfts` to display an interactive plot and `ftsgui` to perform further time series data analysis.

The workflow for using the Financial Time Series app with `chartfts` is:

1. After loading data from either Datafeed Toolbox software or an internal file, select a variable from the **MATLAB Workspace Variables** list box.
2. Click the **Plot** tab, click **Type**, and then select **Interactive Chart**.
3. Click **Plot**. The interactive plot is displayed in `chartfts`. You can then use `chartfts` menu items for further display options.

   For more information on `chartfts`, select **Help > Graphics Help**.

The workflow for using the Financial Time Series app with the Financial Time Series GUI (`ftsgui`) is:

1. After loading data from either Datafeed Toolbox software or an internal file, select a variable from the **MATLAB Workspace Variables** list box.
2. Select **Tools > FTSGUI** to open the Financial Time Series GUI window.
3. Select a variable from the **MATLAB Workspace Variables** list box. Click the **Plot** tab and then select one of the following from the **Type** drop-down list: `Line Plot`, `High-Low Plot`, or `Candlestick Plot`.
4. Click **Plot**. The plot is displayed in a MATLAB graphic window. In addition, the Financial Time Series GUI window displays an entry for the plotted `fints` object. You can then use the menu items in the Financial Time Series GUI window to perform further analysis.

   For more information on `ftsgui`, select **Help > Help on Financial Time Series GUI**.

Note If the selected workspace variable that you are plotting is not a `fints` object, a `fints` object is created when you click **Plot**. The new `fints` object uses the name designated by the **MATLAB workspace variable** box on the **Create** tab.

See Also

`ascii2fts` | `boxcox` | `convertto` | `datestr` | `diff` | `fillts` | `filter` | `fints` | `fts2mat` | `ftsbound` | `lagts` | `leadts` | `peravg` | `resamplets` | `smoothts` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly` | `tsmovavg`

Related Examples

- “What Is the Financial Time Series App?” on page 13-2
- “Getting Started with the Financial Time Series App” on page 13-4
- “Using the Financial Time Series App” on page 13-11
- “Creating Financial Time Series Objects” on page 11-3
- “Visualizing Financial Time Series Objects” on page 11-16
- “Working with Financial Time Series Objects” on page 12-3

Financial Time Series User Interface

Financial Time Series User Interface

Use the Financial Time Series User Interface to analyze your time series data and display the results graphically without resorting to the command line. The Financial Time Series User Interface lets you visualize the data and the results at the same time.

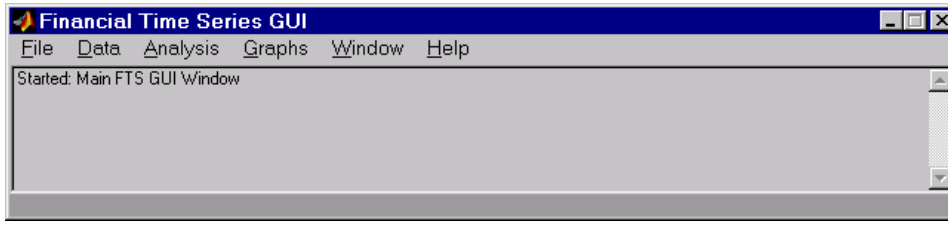
"Using the Financial Time Series GUI" on page 14-7 discusses how to use this user interface.

Main Window

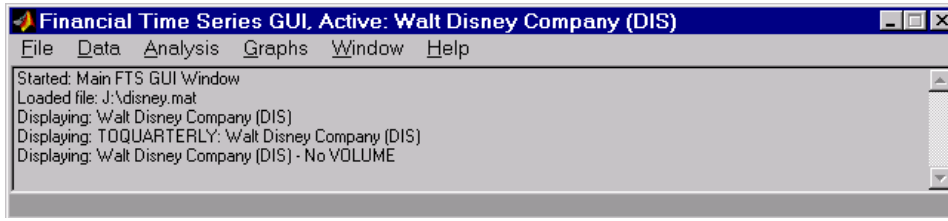
Start the Financial Time Series User Interface with the command

```
ftsgui
```

The Financial Time Series GUI window opens.

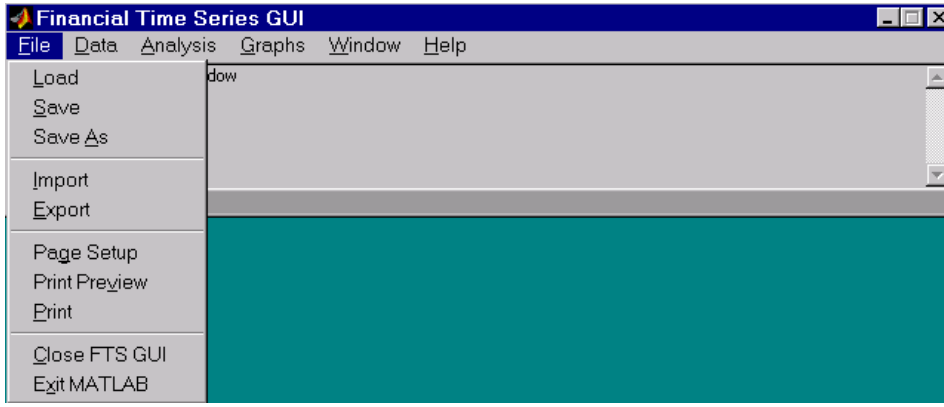


The title bar acts as an active time series object indicator (indicates the currently active financial time series object). For example, if you load the file `disney.mat` and want to use the time series data in the file `dis`, the title bar on the main GUI would read as shown.



The menu bar consists of six menu items: **File** on page 14-3, **Data** on page 14-3, **Analysis** on page 14-4, **Graphs** on page 14-5, **Window** on page 14-5, and **Help** on page 14-5. Under the menu bar is a status box that displays the steps you are doing.

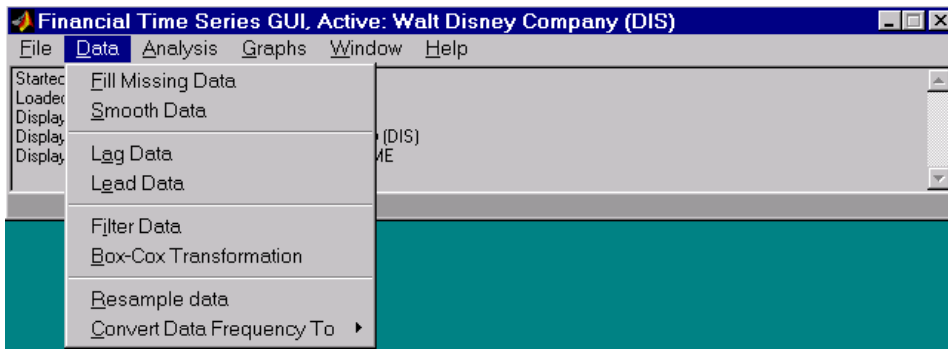
File Menu



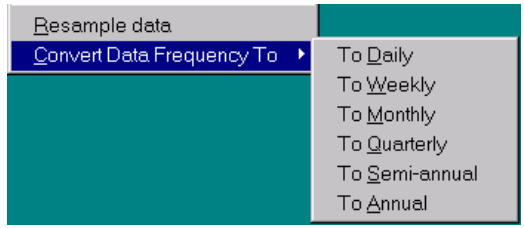
The **File** menu contains the commands for input and output. You can read and save (**Load**, **Save**, and **Save As**) MATLAB MAT-files, ASCII (text) data files. To load MATLAB MAT-files, the MAT-file must contain a `fints` object. You can also import (**Import**) Excel XLS files.

The **File** menu also contains the printing suite (**Page Setup**, **Print Preview**, and **Print**). Lastly, from this menu you can close the GUI itself (**Close FTS GUI**) and quit MATLAB (**Exit MATLAB**).

Data Menu

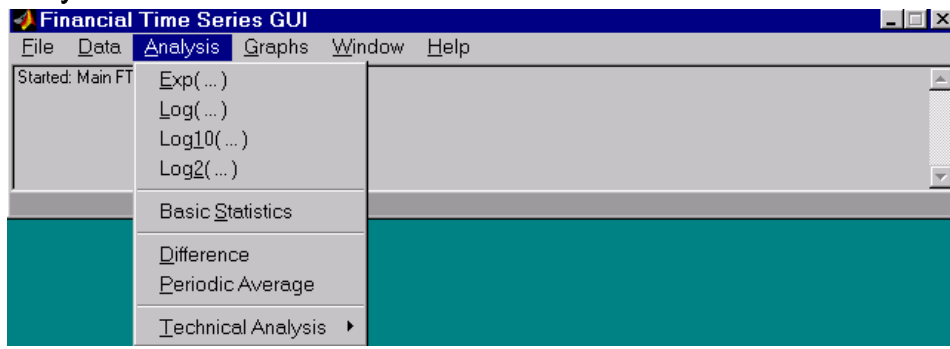


The **Data** menu provides a collection of data manipulation functions and data conversion functions.



To use any of the functions here, make sure that the correct financial time series object is displayed in the title bar of the main GUI window.

Analysis Menu

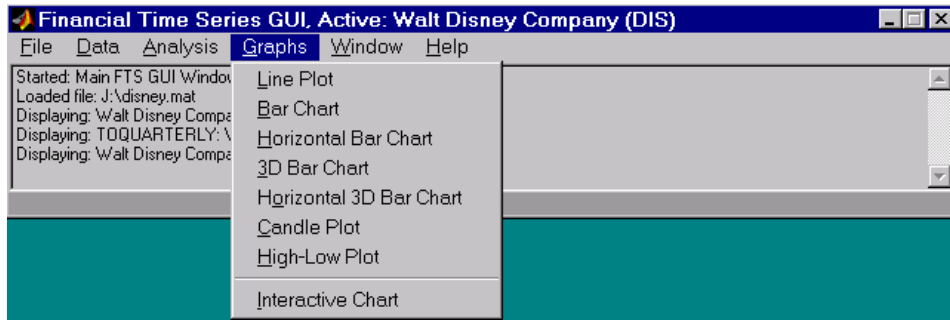


The **Analysis** menu provides

- A set of exponentiation and logarithmic functions.
- Statistical tools (**Basic Statistics**), which calculate and display the minimum, maximum, average (mean), standard deviation, and variance of the current (active) time series object; these basic statistics numbers are displayed in a dialog box.
- Data difference (**Difference**) and periodic average (**Periodic Average**) calculations. Data difference generates a vector of data that is the difference between the first data point and the second, the second, and the third, and so on. The periodic average function calculates the average per defined length period, for example, averages of every five days.
- Technical analysis functions. See “Chart Technical Indicators” for a list of the technical analysis functions.

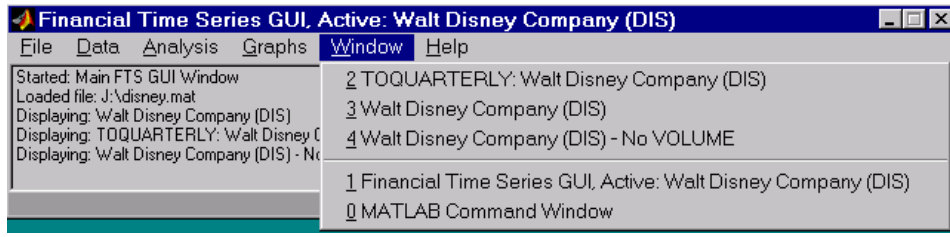
As with the **Data** menu, to use any of the **Analysis** menu functions, make sure that the correct financial time series object is displayed in the title bar of the main GUI window.

Graphs Menu



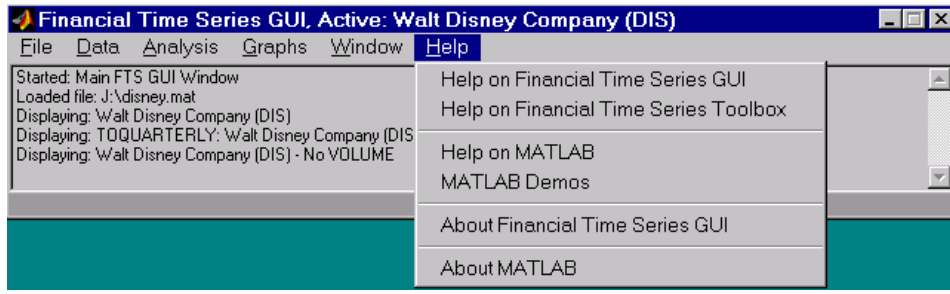
The **Graphs** menu contains functions that graphically display the current (active) financial time series object. You can also start up the interactive charting function (`chartfts`) from this menu.

Window Menu



The **Window** menu lists open windows under the current MATLAB session.

Help Menu



The **Help** menu provides a standard set of Help menu links.

See Also

`ftsgui` | `ftstool`

Related Examples

- “Using the Financial Time Series GUI” on page 14-7
- “Working with Financial Time Series Objects” on page 12-3

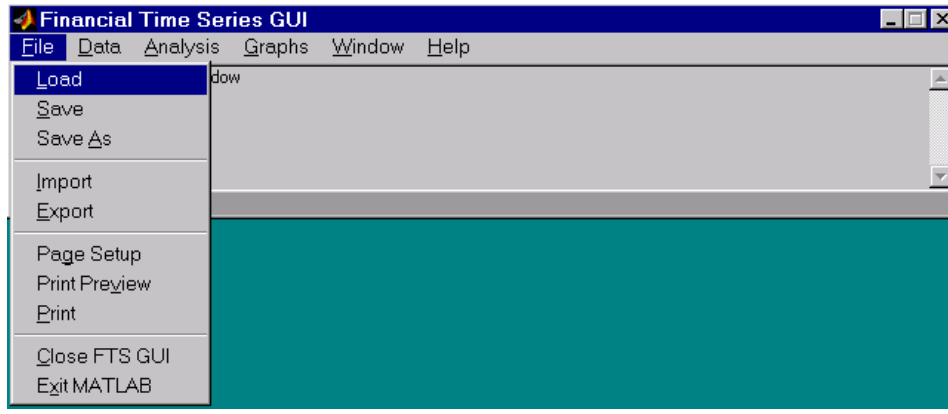
Using the Financial Time Series GUI

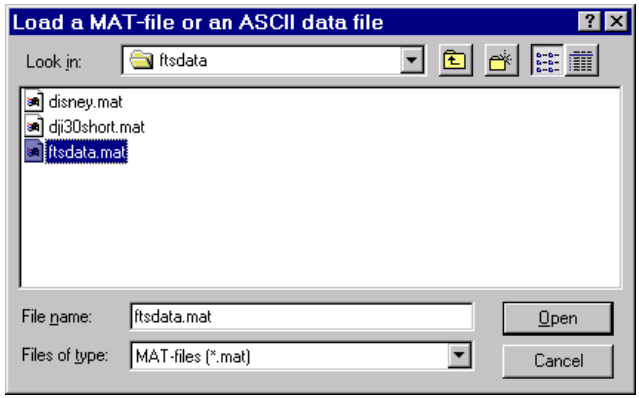
In this section...

Getting Started

To use the Financial Time Series GUI, start the financial time series user interface 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.



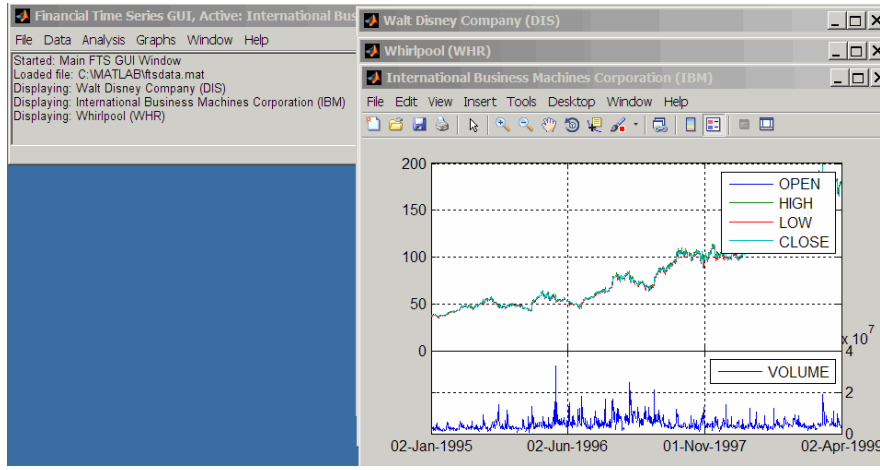


For illustration purposes, choose the file `ftsdata.mat` from the dialog box presented.

If you do not see the MAT-file, look in the folder *`matlabroot`*`\toolbox\finance\findemos`, where *`matlabroot`* is the MATLAB root folder (the folder 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.

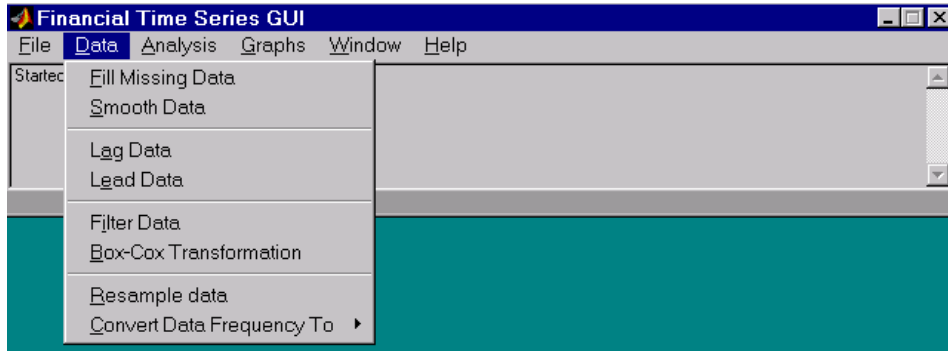
Each financial time series object inside the MAT-file is presented as a line plot in a separate window. The status window is updated accordingly.



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.

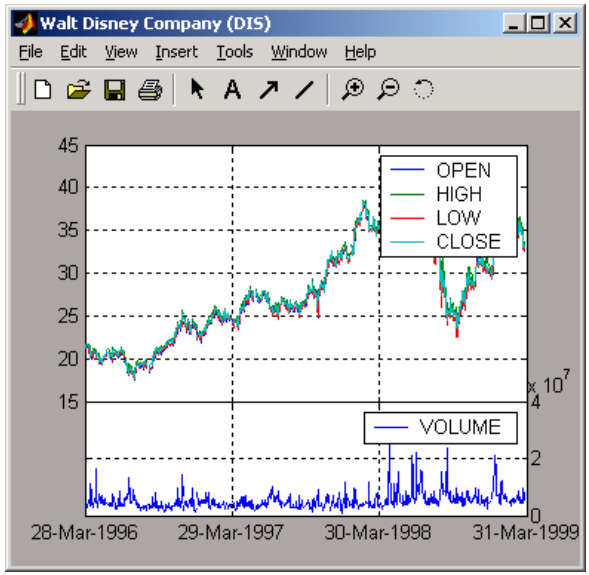


Here are some example tasks that illustrate the use of the functions on this menu.

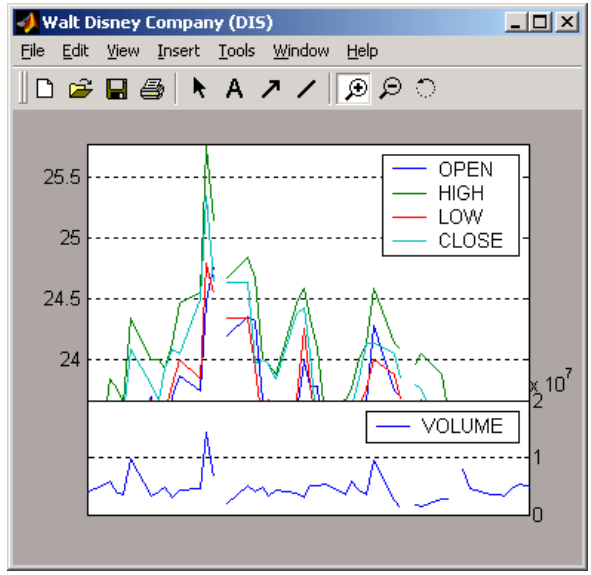
Fill Missing Data

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

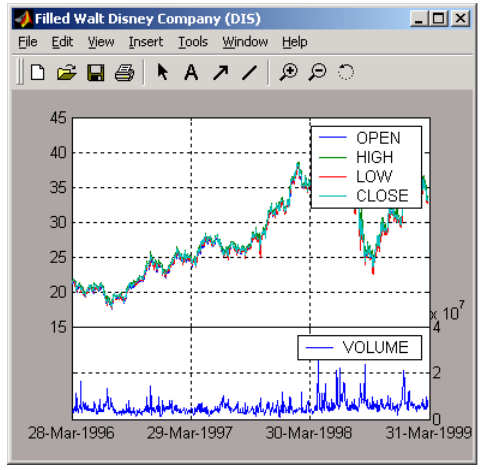
Corp. (IBM), Walt Disney Co. (DIS), and Whirlpool (WHR). Click the window that shows the time series data for Walt Disney Co. (DIS).

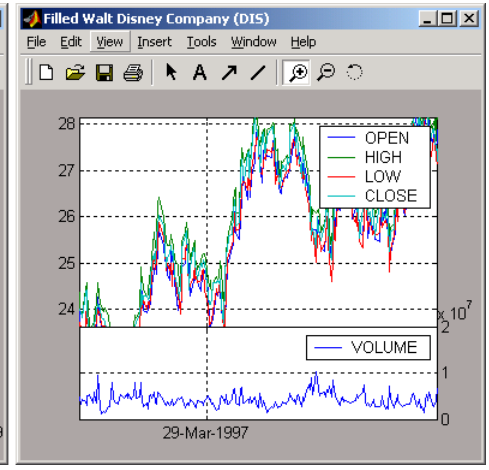


To view any missing data in this time series data set, zoom into the plot using the Zoom tool (the magnifying glass icon with the plus sign) from the toolbar and select a region.



The gaps represent the missing data in the series. To fill these gaps, select **Data > Fill Missing Data**. This selection automatically fills the gaps and generates a new plot that displays the filled time series data.



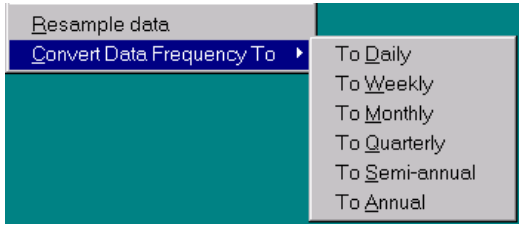


You cannot see the filled gaps when you display the entire data set. However, when you zoom into the plot, you see that the gaps have been eliminated. The title bar has

changed; the title has been prefixed with the word **Filled** to reflect the filled time series data.

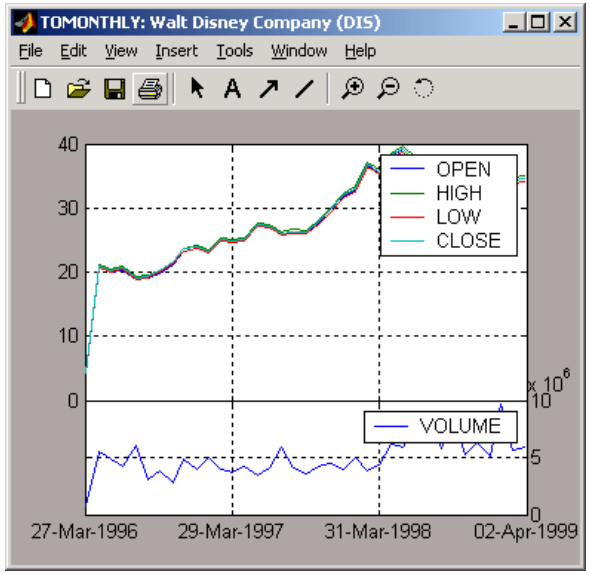
Frequency Conversion

The **Data** menu also provides access to frequency conversion functions.



This example changes the DIS time series data frequency from daily to monthly. Close the Filled Walt Disney Company (DIS) window, and click the Walt Disney Company (DIS) window to make it active (current) again. Then, from the **Data** menu, select **Convert Data Frequency To** and **To Monthly**.

A new figure window displays the result of this conversion.

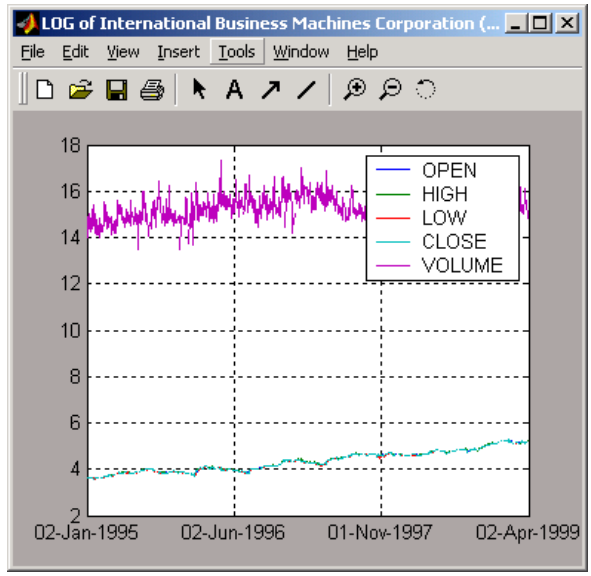


The title reflects that the data displayed had its frequency changed to monthly.

Analysis Menu

The **Analysis** menu provides functions that analyze time series data, including the technical analysis functions. (See “Chart Technical Indicators” for a complete list of the technical analysis functions and several usage examples.)

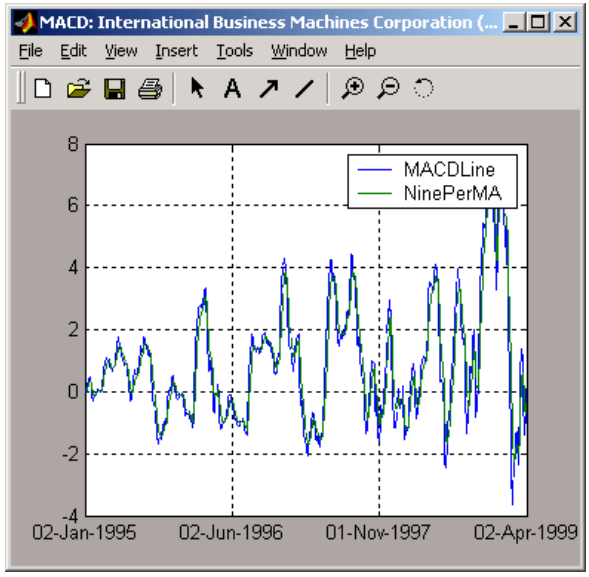
For example, you can use the **Analysis** menu to calculate the natural logarithm (`log`) of the data contained within the data set `ftsdata.mat`. This data file provides time series data for IBM (IBM), Walt Disney (DIS), and Whirlpool (WHR). Click the window displaying the data for IBM Corporation (IBM) to make it active (current). Then select the **Analysis** menu, followed by **Log(...)**. The result appears in its own window.



Close the above window and click again on the IBM data window to make it active (current).

Note Before proceeding with any time series analysis, make certain that the title bar confirms that the active data series is the correct one.

From the **Analysis** menu on the main window on page 14-2, select **Technical Analysis** and **MACD**. The result, again, is displayed in its own window.



Other analysis functions work similarly.

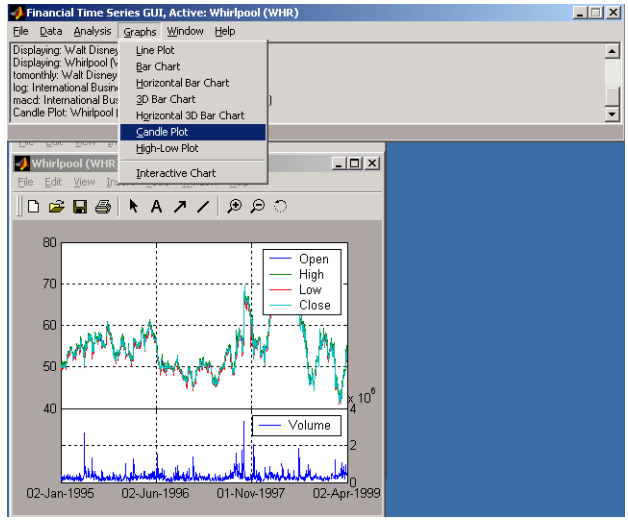
Graphs Menu

The **Graphs** menu displays time series data using the provided graphics functions. Included in the **Graphs** menu are several types of bar charts (`bar, barh` and `bar3, bar3h`), line plot (`plot`), candle plot (`candle`), and High-Low plot (`highlow`). The **Graphs** menu also provides access to the interactive charting function, `chartfts`.

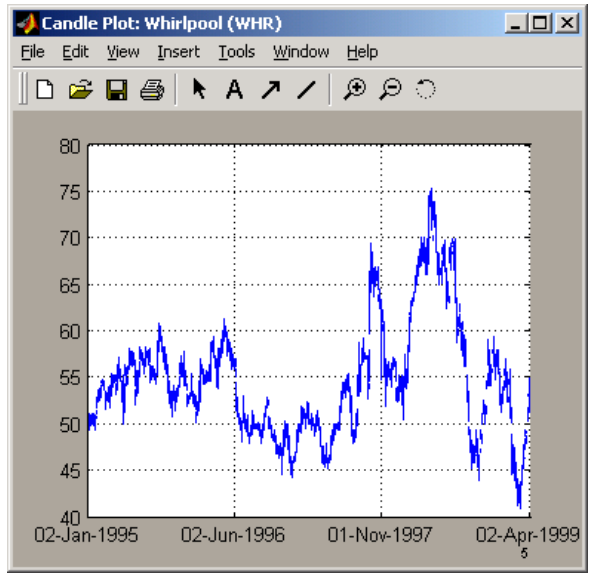
Candle Plot

For example, you can display the candle plot of a set of time series data and start up the interactive chart on the same data set.

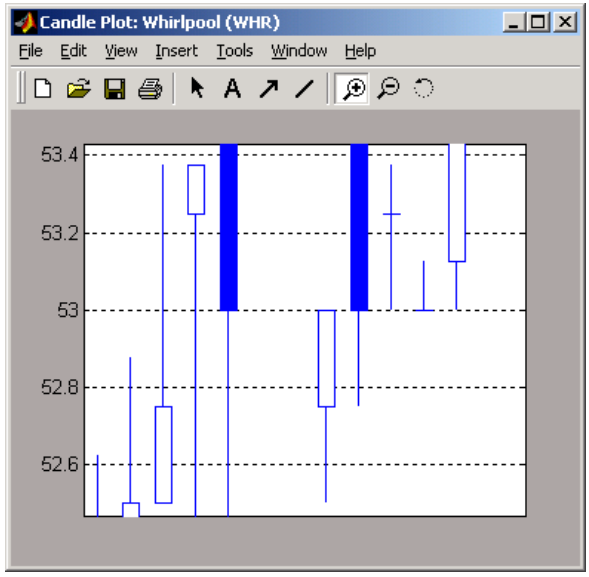
Load the `ftsdata.mat` data set, and click the window that displays the Whirlpool (WHR) time series data to make it active (current). From the main window on page 14-2, select the **Graphs** menu and then **Candle Plot**.



The result is shown below.

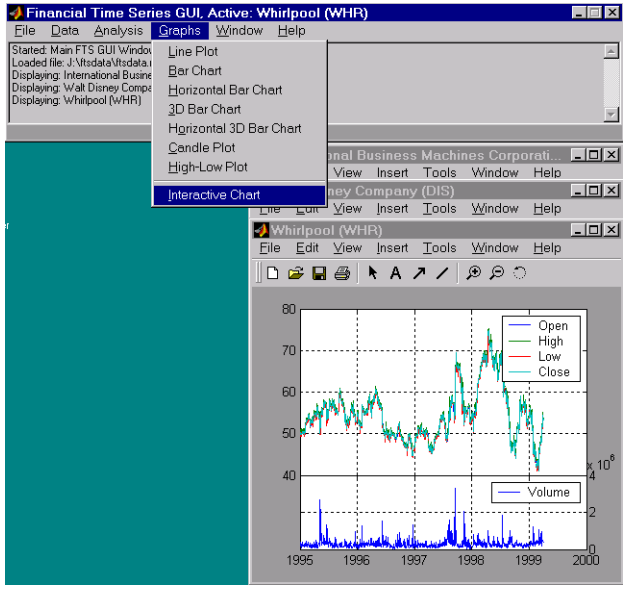


This does not look much like a candle plot because there are too many data points in the data set. All the candles are too compressed for effective viewing. However, when you zoom into a region of this plot, the candles become apparent.

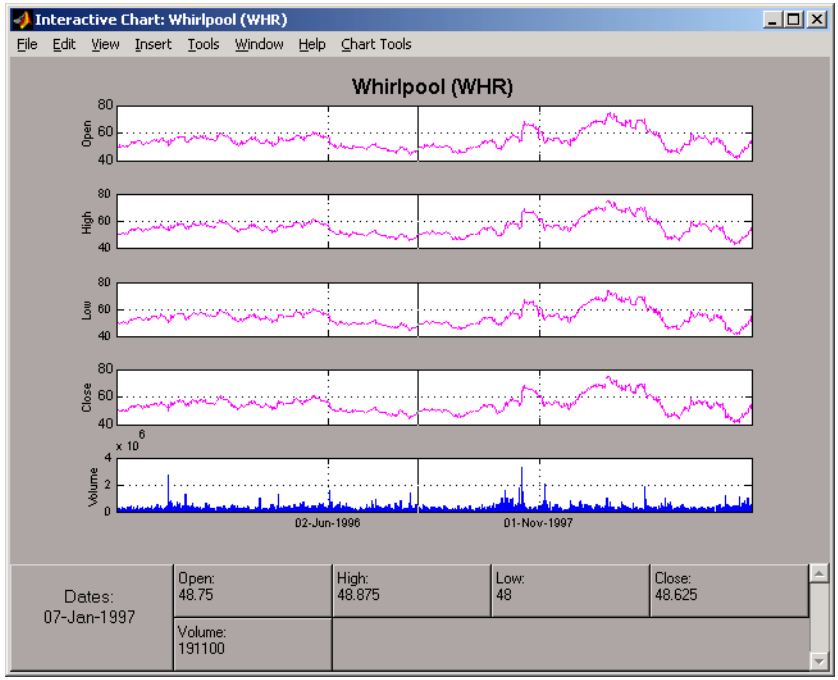


Interactive Chart

To create an interactive chart (`chartfts`) on the Whirlpool data, click the window that displays the Whirlpool (WHR) data to make it active (current). Then, go to the **Graphs** menu and select **Interactive Chart**.



The chart that results is shown below.



You can use this interactive chart as if you had invoked it with the `chartfts` command from the MATLAB command line. For a tutorial on the use of `chartfts`, see "Visualizing Financial Time Series Objects" on page 11-16.

Saving Time Series Data

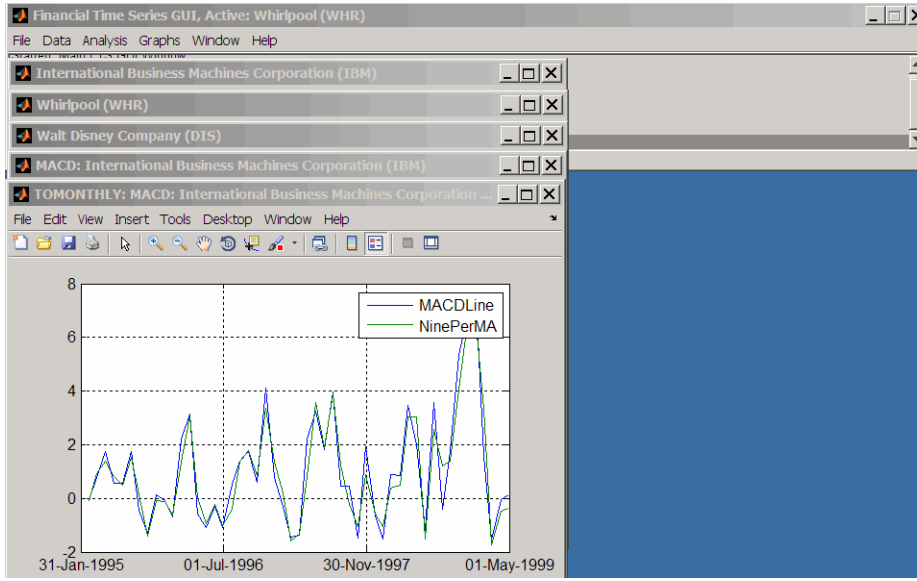
The **Save** and **Save As** items on the main window **File** menu let you save the time series data that results from your analyses and computations. These items save *all* time series data that has been loaded or processed during the current session, even if the window displaying the results of a computation has previously been dismissed.

Note The **Save** and **Save As** items on the **File** menu of the individual plot windows will not save the time series data, but will save the actual plot.

You can save your time series data in two ways:

- Into the latest MAT-file loaded (**Save** on page 14-19)
- Into a MAT-file chosen (or named) from the window (**Save As** on page 14-20)

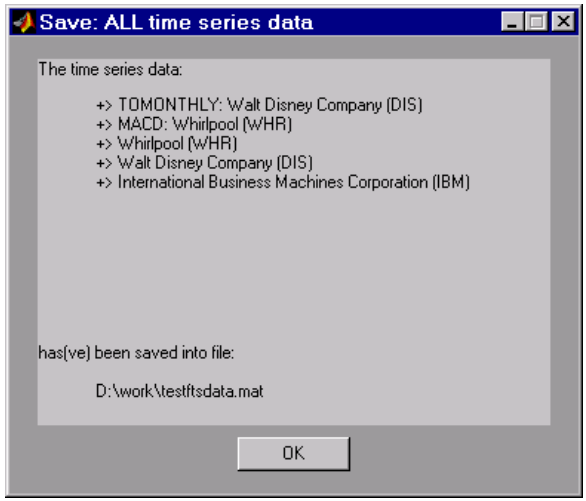
To illustrate this, start by loading the data file `testftsdata.mat` (located in *`matlabroot`*`/toolbox/finance/findemos`). Then, convert the Disney (DIS) data from daily (the original frequency) to monthly data. Next, run the MACD analysis on the Whirlpool (WHR) data. You now have a set of five open figure windows.



Saving into the Original File (Save)

To save the data back into the original file (`testftsdata.mat`), select **Save** from the **File** menu.

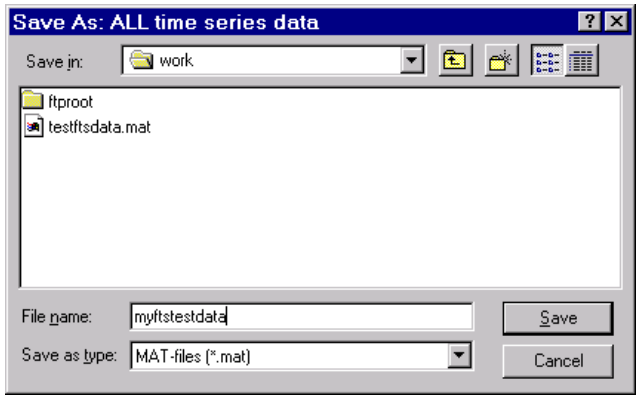
A confirmation window appears. It confirms that the data has been saved in the latest MAT-file loaded (`testftsdata.mat` in this example).



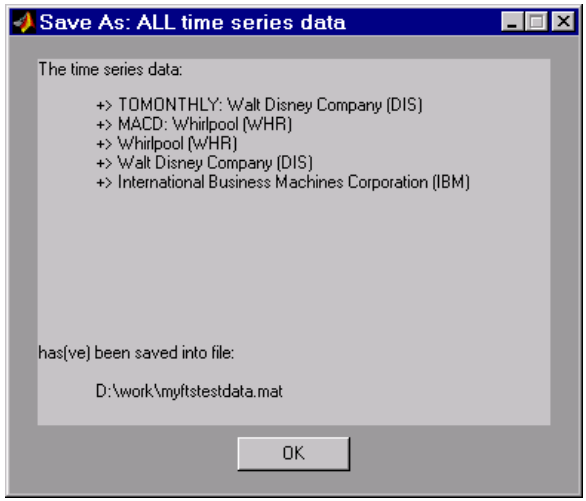
Saving into a New File (Save As)

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`.

See Also

`ftsgui | ftstool`

Related Examples

- “Working with Financial Time Series Objects” on page 12-3

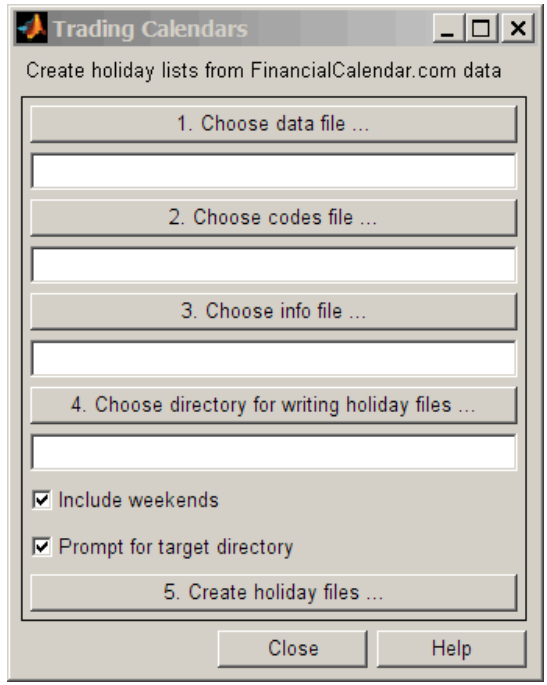
More About

- “Financial Time Series User Interface” on page 14-2

Trading Date Utilities

Trading Calendars User Interface

Use the `createholidays` function to open the Trading Calendars user interface.



The `createholidays` function supports `http://www.FinancialCalendar.com` trading calendars. This function can be used from the command line or from the Trading Calendars user interface. To use `createholidays` or the Trading Calendars user interface, you must obtain data, codes, and info files from `http://www.FinancialCalendar.com` trading calendars. For more information on using the command line to programmatically generate the market-specific `holidays.m` files without displaying the interface, see `createholidays`.

To use the Trading Calendars user interface:

1 From the command line, type the following command to open the Trading Calendars user interface.

```
createholidays
```

2. Click **Choose data file** to select the data file.
3. Click **Choose codes file** to select the codes file.
4. Click **Choose info file** to select the info file.
5. Click **Choose directory for writing holiday files** to select the output folder.
6. Select **Include weekends** to include weekends in the holiday list and click **Prompt for target directory** to be prompted for the file location for each `holidays.m` file that is created.
7. Click **Create holiday files** to convert `FinancialCalendar.com` financial center holiday data into market-specific `holidays.m` files.

   The market-specific `holidays.m` files can be used in place of the standard `holidays.m` that ships with Financial Toolbox software.

See Also

`createholidays` | `holidays` | `nyseclosures`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “UICalendar User Interface” on page 15-4

UICalendar User Interface

In this section...

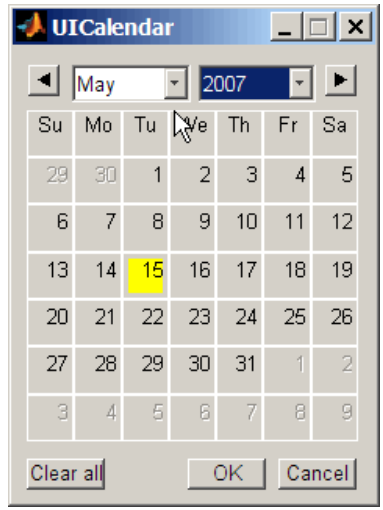
Using UICalendar in Standalone Mode

You can use the UICalendar user interface in standalone mode to look up any date. To use the standalone mode:

1 Type the following command to open the UICalendar GUI:

```
uicalendar
```

The UICalendar interface is displayed:



2 Click the date and year controls to locate any date.

Using UICalendar with an Application

You can use the UICalendar user interface with an application to look up any date. To use the UICalendar graphical interface with an application, use the following command:

```
uicalendar('PARAM1', VALUE1, 'PARAM2', VALUE2', ...)
```

For more information, see `uicalendar`.

Example of Using UICalendar with an Application

The UICalendar example creates a function that displays a user interface that lets you select a date from the UICalendar user interface and fill in a text field with that date.

1. Create a figure.

```
function uicalendarGUIExample
f = figure('Name', 'uicalendarGUIExample');
```

2. Add a text control field.

```
dateTextHandle = uicontrol(f, 'Style', 'Text', ...
'String', 'Date:', ...
'Horizontalalignment', 'left', ...
'Position', [100 200 50 20]);
```

3. Add an uicontrol editable text field to display the selected date.

```
dateEditBoxHandle = uicontrol(f, 'Style', 'Edit', ...
'Position', [140 200 100 20], ...
'BackgroundColor', 'w');
```

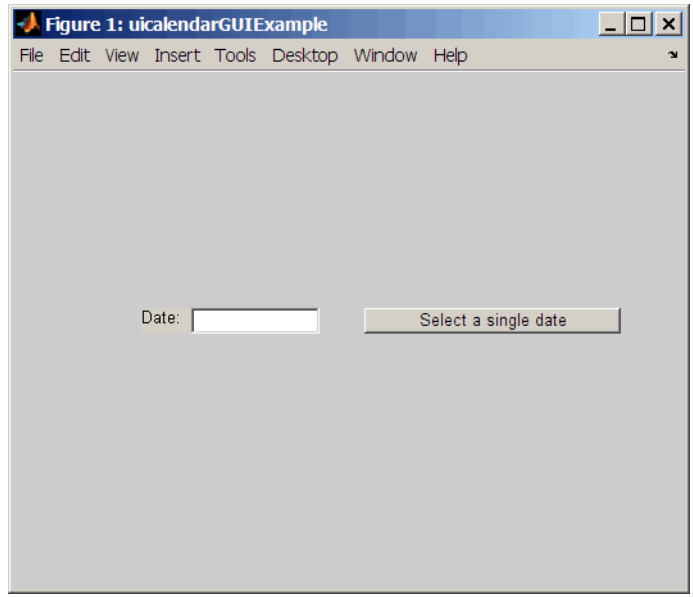
4. Create a push button that startups the UICalendar.

```
calendarButtonHandle = uicontrol(f, 'Style', 'PushButton', ...
'String', 'Select a single date', ...
'Position', [275 200 200 20], ...
'callback', @pushbutton_cb);
```

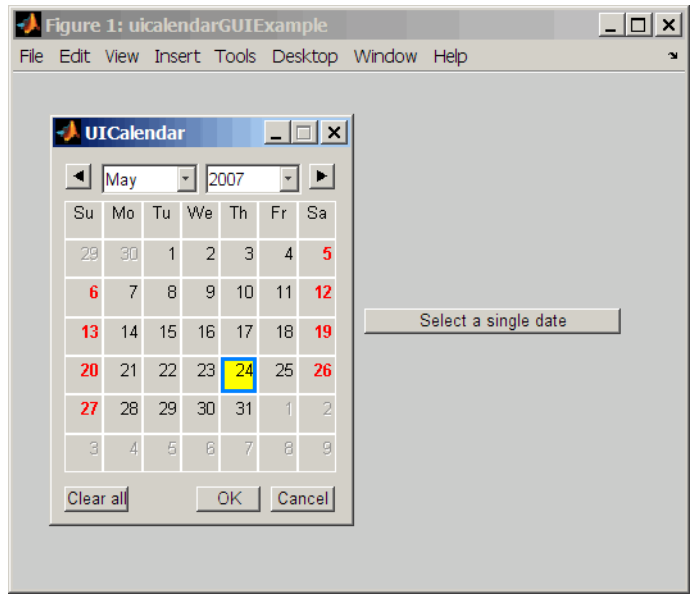
5. To start up UICalendar, create a nested function (callback function) for the push button.

```
function pushbutton_cb(hcbo, eventStruct)
% Create a UICALENDAR with the following properties:
% 1) Highlight weekend dates.
% 2) Only allow a single date to be selected at a time.
% 3) Send the selected date to the edit box uicontrol.
uicalendar('Weekend', [1 0 0 0 0 0 1], ...
'SelectionType', 1, ...
'DestinationUI', dateEditBoxHandle);
end
end
```

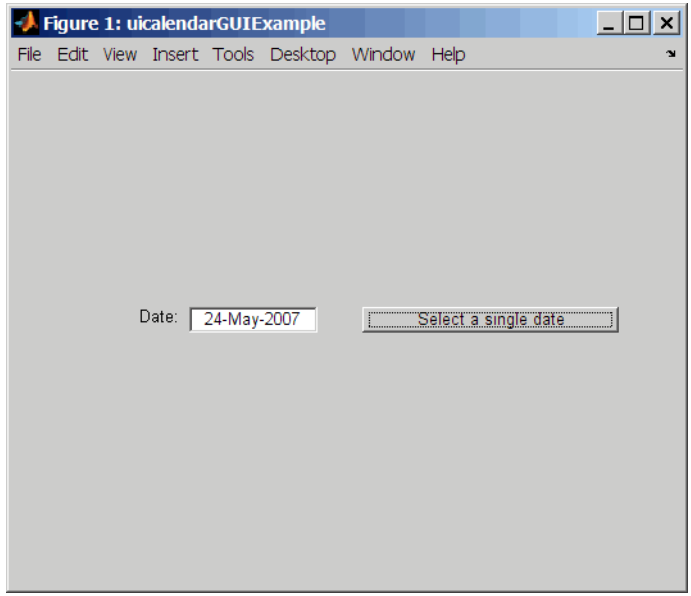
6. Run the function `uicalendarGUIExample` to display the application interface:



7 Click **Select a single date** to display the UICalendar user interface:



8 Select a date and click **OK** to display the date in the text field:



See Also

`createholidays` | `holidays` | `nyseclosures`

Related Examples

- “Trading Calendars User Interface” on page 15-2
- “Handle and Convert Dates” on page 2-2

Technical Analysis

Technical Indicators

Technical analysis (or charting) is used by some investment managers to help manage portfolios. Technical analysis relies heavily on the availability of historical data. Investment managers calculate different indicators from available data and plot them as charts. Observations of price, direction, and volume on the charts assist managers in making decisions on their investment portfolios.

The technical analysis functions in Financial Toolbox are tools to help analyze your investments. The functions in themselves will not make any suggestions or perform any qualitative analysis of your investment.

Technical Analysis: Oscillators

Function	Type
`adosc`	Accumulation/distribution oscillator
`chaikosc`	Chaikin oscillator
`macd`	Moving Average Convergence/Divergence
`stochosc`	Stochastic oscillator
`tsaccel`	Acceleration
`tsmom`	Momentum

Technical Analysis: Stochastics

Function	Type
`chaikvolat`	Chaikin volatility
`fpctkd`	Fast stochastics
`spctkd`	Slow stochastics
`willpctr`	Williams %R

Technical Analysis: Indexes

Function	Type
`negvolidx`	Negative volume index
`posvolidx`	Positive volume index
`rsindex`	Relative strength index

Technical Analysis: Indicators

Function	Type
`adline`	Accumulation/distribution line
`bollinger`	Bollinger band
`hhigh`	Highest high
`llow`	Lowest low
`medprice`	Median price
`onbalvol`	On balance volume
`prcroc`	Price rate of change
`pvtrend`	Price-volume trend
`typprice`	Typical price
`volroc`	Volume rate of change
`wclose`	Weighted close
`willad`	Williams accumulation/distribution

See Also

`adline` | `adosc` | `bollinger` | `chaikosc` | `chaikvolat` | `fpctkd` | `hhigh` | `llow` | `macd` | `medprice` | `negvolidx` | `onbalvol` | `posvolidx` | `prcroc` | `pvtrend` | `rsindex` | `spctkd` | `stochosc` | `tsaccel` | `tsmom` | `typprice` | `volroc` | `wclose` | `willad` | `willpctr`

Related Examples

- “Technical Analysis Examples” on page 16-4

Technical Analysis Examples

In this section...

Overview

To illustrate some of the technical analysis functions, this section uses the IBM stock price data contained in the supplied file `ibm9599.dat`. First create a financial time series object from the data using `ascii2fts`:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
```

The time series data contains the open, close, high, and low prices, and 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. Use the IBM® stock price data contained in the supplied file `ibm9599.dat`. First, create a financial time series object from the data using `ascii2fts`. Look at the portion of the time series covering the 3-month period between October 1, 1995 and December 31, 1995. At the same time fill any missing values due to holidays within the time period specified:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
part_ibm = fillts(ibm('10/01/95::12/31/95'));
```

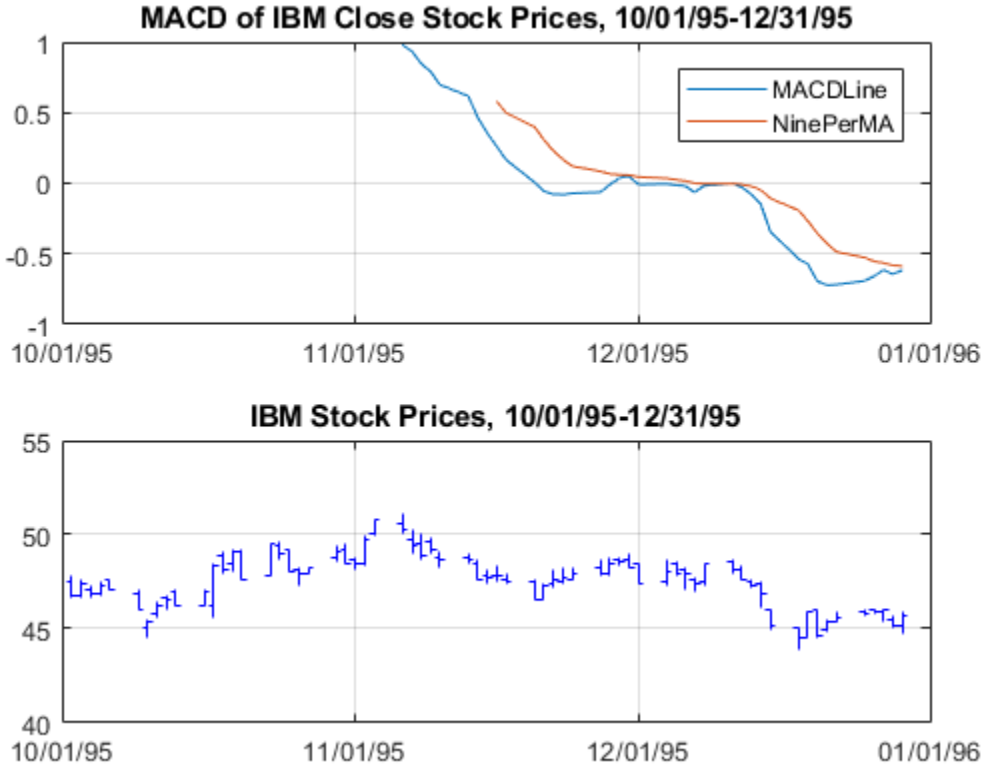
Calculate the MACD, which when plotted produces two lines; the first line is the MACD line itself and the second is the nine-period moving average line:

```
macd_ibm = macd(part_ibm);
```

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')
```



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').
```

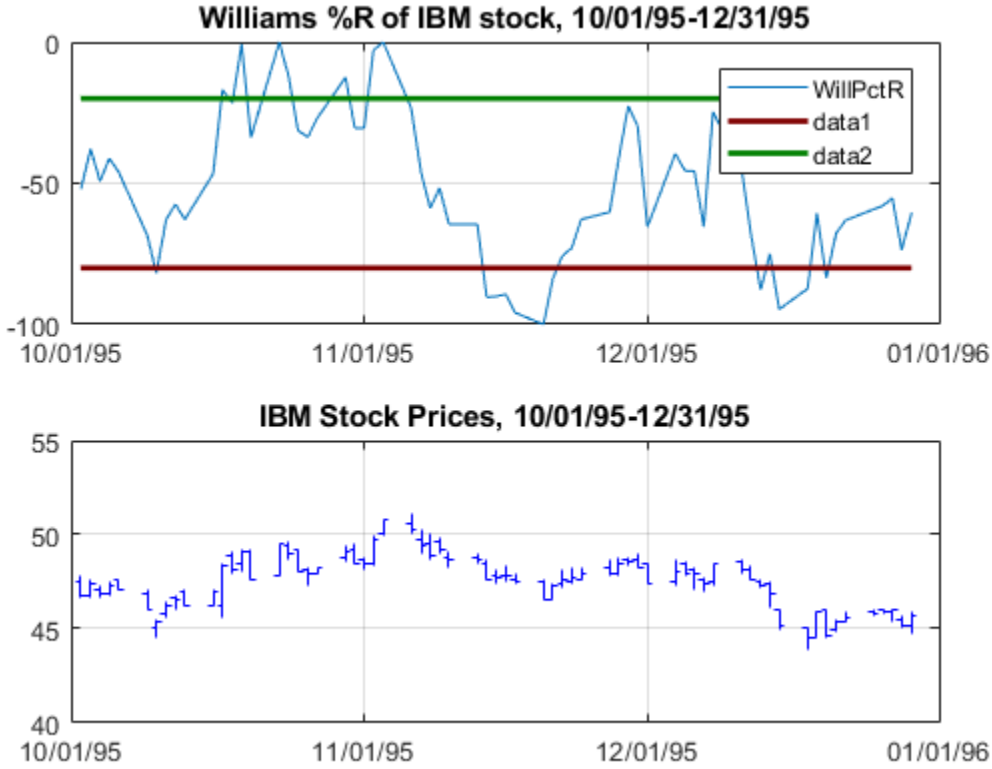
The function `willpctr` now assumes that your high price series is named `Hi`, low price series is named `Lo`, and closing price series is named `Closing`. Use the IBM® stock price data contained in the supplied file `ibm9599.dat`. First, create a financial time series object from the data using `ascii2fts`. Look at the portion of the time series covering the 3-month period between October 1, 1995 and December 31, 1995. At the same time fill any missing values due to holidays within the time period specified:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
part_ibm = fillts(ibm('10/01/95::12/31/95'));
```

Since the time series object `part_ibm` has its data series names identical to the required names, name adjustments are not needed. The input argument to the function is only the name of the time series object itself.

Calculate and plot the Williams %R indicator for IBM stock along with the price range using these commands:

```
wpctr_ibm = willpctr(part_ibm);
subplot(2, 1, 1);
plot(wpctr_ibm);
title('Williams %R of IBM stock, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
hold on;
plot(wpctr_ibm.dates, -80*ones(1, length(wpctr_ibm)),...
'color', [0.5 0 0], 'linewidth', 2)
plot(wpctr_ibm.dates, -20*ones(1, length(wpctr_ibm)),...
'color', [0 0.5 0], 'linewidth', 2)
subplot(2, 1, 2);
highlow(part_ibm);
title('IBM Stock Prices, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
```



The 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. First, create a financial time series object from the data using `ascii2fts`. Look at the portion of the time series covering the

3-month period between October 1, 1995 and December 31, 1995. At the same time fill any missing values due to holidays within the time period specified:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
part_ibm = fillts(ibm('10/01/95::12/31/95'));
```

Calculate and plot the RSI for IBM® stock along with the price range using these commands:

```
rsi_ibm = rsindex(part_ibm);
subplot(2, 1, 1);
plot(rsi_ibm);
title('RSI of IBM stock, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
hold on;
wpctr_ibm = willpctr(part_ibm);
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');
```

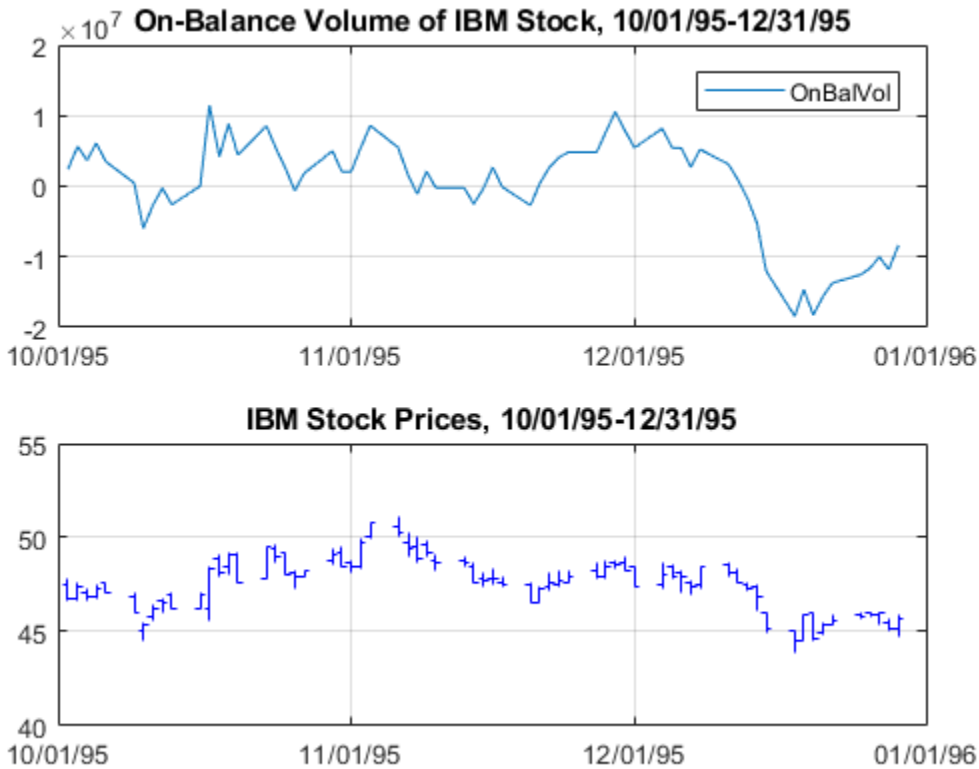
On-Balance Volume (OBV)

On-Balance Volume (OBV) relates volume to price change. The function `onbalvol` requires you to have the closing price (`Close`) series and the volume traded (`Volume`) series. First, create a financial time series object from the data using `ascii2fts`. Look at the portion of the time series covering the 3-month period between October 1, 1995 and December 31, 1995. At the same time fill any missing values due to holidays within the time period specified:

```
ibm = ascii2fts('ibm9599.dat', 1, 3, 2);
part_ibm = fillts(ibm('10/01/95::12/31/95'));
```

Calculate and plot the OBV for IBM® stock along with the price range using these commands:

```
obv_ibm = onbalvol(part_ibm);
subplot(2, 1, 1);
plot(obv_ibm);
title('On-Balance Volume of IBM Stock, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
subplot(2, 1, 2);
highlow(part_ibm);
title('IBM Stock Prices, 10/01/95-12/31/95');
datetick('x', 'mm/dd/yy');
```



See Also

`adline` | `adosc` | `bollinger` | `chaikosc` | `chaikvolat` | `fpctkd` | `hhigh` | `llow` | `macd` | `medprice` | `negvolidx` | `onbalvol` | `posvolidx` | `prcroc` | `pvtrend` | `rsindex` | `spctkd` | `stochosc` | `tsaccel` | `tsmom` | `typprice` | `volroc` | `wclose` | `willad` | `willpctr`

Related Examples

- “Technical Indicators” on page 16-2

Stochastic Differential Equations

SDEs

In this section...

SDE Modeling

Financial Toolbox enables you to model dependent financial and economic variables, such as interest rates and equity prices, by performing Monte Carlo simulation of stochastic differential equations (SDEs). The flexible architecture of the SDE engine provides efficient simulation methods that allow you to create new simulation and derivative pricing methods.

The following table lists tasks you can perform using the SDE functionality.

To perform this task ...	Use these types of models ...
“Simulating Equity Prices” on page 17-34	• Geometric Brownian Motion (GBM) on page 17-27 • Constant Elasticity of Variance (CEV) on page 17-26 • Stochastic Differential Equation (SDE) on page 17-16 • Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) on page 17-19 • Stochastic Differential Equations from Linear Drift (SDELD) on page 17-23 • Heston Stochastic Volatility (Heston) on page 17-31

To perform this task ...	Use these types of models ...
"Simulating Interest Rates" on page 17-59	• Hull-White-Vasicek (HWV) on page 17-30 • Cox-Ingersoll-Ross (CIR) on page 17-29 • Stochastic Differential Equation (SDE) on page 17-16 • Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) on page 17-19 • Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models on page 17-28
"Pricing Equity Options" on page 17-55	Geometric Brownian Motion (GBM) on page 17-27
"Stratified Sampling" on page 17-70	All supported models on page 17-14
"Performance Considerations" on page 17-76	All supported models on page 17-14

Trials vs. Paths

Monte Carlo simulation literature often uses different terminology for the evolution of the simulated variables of interest, such as trials and paths. The following sections use the terms *trial* and *path* interchangeably.

However, there are situations where you should distinguish between these terms. Specifically, the term *trial* often implies the result of an independent random experiment (for example, the evolution of the price of a single stock or portfolio of stocks). Such an experiment computes the average or expected value of a variable of interest (for example, the price of a derivative security) and its associated confidence interval.

By contrast, the term *path* implies the result of a random experiment that is different or unique from other results, but that may or may not be independent.

The distinction between these terms is usually unimportant. It may, however, be useful when applied to variance reduction techniques that attempt to increase the efficiency of Monte Carlo simulation by inducing dependence across sample paths. A classic example

involves pairwise dependence induced by antithetic sampling, and applies to more sophisticated variance reduction techniques, such as stratified sampling.

NTRIALS, NPERIODS, and NSTEPS

SDE methods in the Financial Toolbox software use the parameters `NTRIALS`, `NPERIODS`, and `NSTEPS` as follows:

- The input argument `NTRIALS` specifies the number of simulated trials or sample paths to generate. This argument always determines the size of the third dimension (the number of pages) of the output three-dimensional time series array `Paths`. Indeed, in a traditional Monte Carlo simulation of one or more variables, each sample path is independent and represents an independent trial.
- The parameters `NPERIODS` and `NSTEPS` represent the number of simulation periods and time steps, respectively. Both periods and time steps are related to time increments that determine the exact sequence of observed sample times. The distinction between these terms applies only to issues of accuracy and memory management. For more information, see “Optimizing Accuracy: About Solution Precision and Error” on page 17-78 and “Managing Memory” on page 17-76.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDE Class Hierarchy” on page 17-5
- “SDE Models” on page 17-8

SDE Class Hierarchy

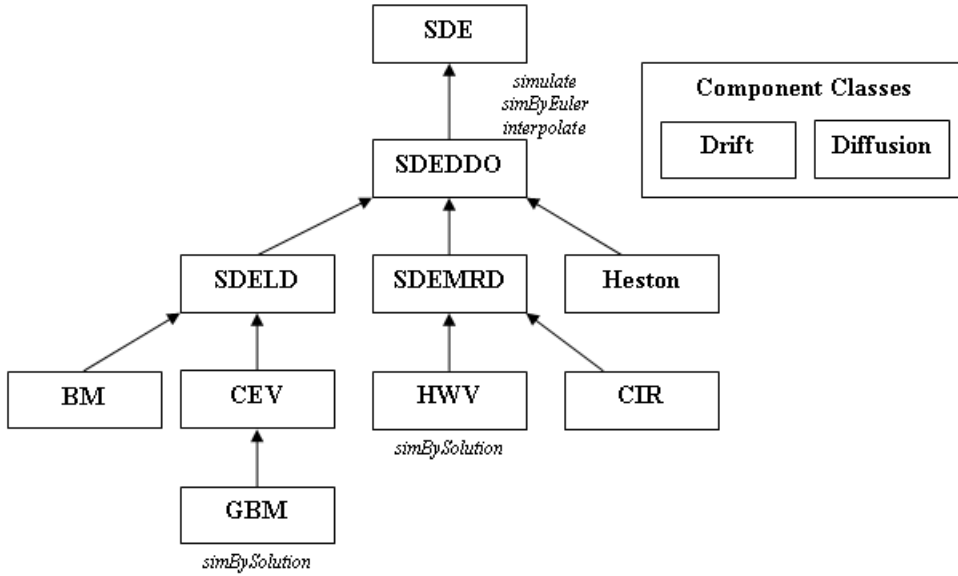
TheFinancial Toolbox SDE class structure represents a generalization and specialization hierarchy. The top-level class provides the most general model interface and offers the default Monte Carlo simulation and interpolation methods. In turn, derived classes offer restricted interfaces that simplify model creation and manipulation while providing detail regarding model structure.

The following table lists the SDE classes. The introductory examples in “Available Models” on page 17-14 show how to use these classes to create objects associated with univariate models. Although the Financial Toolbox SDE engine supports multivariate models, univariate models facilitate object creation and display, and allow you to easily associate inputs with object parameters.

SDE Classes

Class Name	For More Information, See ...
`SDE`	`sde` and "Base SDE Models" on page 17-16
`Drift`, `Diffusion`	`drift`, `diffusion`, and "Overview" on page 17-19
`SDEDDO`	`sdeddo` and "Drift and Diffusion Models" on page 17-19
`SDELD`	`sdeld` and "Linear Drift Models" on page 17-23
`CEV`	`cev` and "Creating Constant Elasticity of Variance (CEV) Models" on page 17-26
`BM`	`bm` and "Creating Brownian Motion (BM) Models" on page 17-25
`SDEMRD`	`sdemrd` and "Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models" on page 17-28
`GBM`	`gbm` and "Creating Geometric Brownian Motion (GBM) Models" on page 17-27
`HWV`	`hwv` and "Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 17-30
`CIR`	`cir` and "Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models" on page 17-29
`Heston`	`heston` and "Creating Heston Stochastic Volatility Models" on page 17-31

The following figure illustrates the inheritance relationships among SDE classes.



See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

More About

- "SDEs" on page 17-2
- "SDE Models" on page 17-8

SDE Models

In this section...

Introduction

Most models and utilities available with Monte Carlo Simulation of SDEs are represented as MATLAB objects. Therefore, this documentation often uses the terms model and object interchangeably.

However, although all models are represented as objects, not all objects represent models. In particular, `drift` and `diffusion` objects are used in model specification, but neither of these types of objects in and of themselves makes up a complete model. In most cases, you do not need to create `drift` and `diffusion` objects directly, so you do not need to differentiate between objects and models. It is important, however, to understand the distinction between these terms.

In many of the following examples, most model parameters are evaluated or invoked like any MATLAB function. Although it is helpful to examine and access model parameters as you would data structures, think of these parameters as *functions* that perform *actions*.

Creating SDE Objects

Constructing Objects

You use *constructors* to create SDE objects.

For examples and more information, see:

- "Available Models" on page 17-14
- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59

Displaying Objects

- Objects display like traditional MATLAB data structures.
- Displayed object parameters appear as nouns that begin with capital letters. In contrast, parameters such as `simulate` and `interpolate` appear as verbs that begin with lowercase letters, which indicate tasks to perform.

Assigning and Referencing Object Parameters

- Objects support referencing similar to data structures. For example, statements like the following are generally valid:

  ```
  A = obj.A
  ```

- Objects support complete parameter assignment similar to data structures. For example, statements like the following are generally valid:

  ```
  obj.A = 3
  ```

- Objects do not support partial parameter assignment as data structures do. Therefore, statements like the following are generally **invalid**:

  ```
  obj.A(i,j) = 0.3
  ```

Constructing and Evaluating Models

- You can construct objects of any model class only if enough information is available to determine unambiguously the dimensionality of the model. Because various class constructors offer unique input interfaces, some models require additional information to resolve model dimensionality.
- You need only enter required input parameters in placeholder format, where a given input argument is associated with a specific position in an argument list. You can enter optional inputs in any order as parameter name-value pairs, where the name of

a given parameter appears in single quotation marks and precedes its corresponding value.

- Association of dynamic (time-variable) behavior with function evaluation, where *time* and *state* (t,X_t) are passed to a common, published interface, is pervasive throughout the SDE class system. You can use this function evaluation approach to model or construct powerful analytics. For a simple example, see "Example: Univariate GBM Models" on page 17-28.

Specifying SDE Simulation Parameters

The SDE engine allows the simulation of generalized multivariate stochastic processes, and provides a flexible and powerful simulation architecture. The framework also provides you with utilities and model classes that offer various parametric specifications and interfaces. The architecture is fully multidimensional in both the state vector and the Brownian motion, and offers both linear and mean-reverting drift-rate specifications.

You can specify most parameters as MATLAB arrays or as functions accessible by a common interface, that supports general dynamic/nonlinear relationships common in SDE simulation. Specifically, you can simulate correlated paths of any number of state variables driven by a vector-valued Brownian motion of arbitrary dimensionality. This simulation approximates the underlying multivariate continuous-time process using a vector-valued stochastic difference equation.

Consider the following general stochastic differential equation:

$$dX_t = F(t,X_t)dt + G(t,X_t)dW_t$$

where:

- X is an *NVARS*-by-`1` state vector of process variables (for example, short rates or equity prices) to simulate.
- W is an *NBROWNS*-by-`1` Brownian motion vector.
- F is an *NVARS*-by-`1` vector-valued drift-rate function.
- G is an *NVARS*-by-*NBROWNS* matrix-valued diffusion-rate function.

The drift and diffusion rates, F and G, respectively, are general functions of a real-valued scalar sample time t and state vector X_t. Also, static (non-time-variable) coefficients are simply a special case of the more general dynamic (time-variable) situation, just as a function can be a trivial constant; for example, $f(t,X_t) = 4$. The SDE in "Equation 17-1" on

page 17-10 is useful in implementing derived classes that impose additional structure on the drift and diffusion-rate functions.

Specifying User-Defined Functions as Model Parameters

Several examples in this documentation emphasize the evaluation of object parameters as functions accessible by a common interface. In fact, you can evaluate object parameters by passing to them time and state, regardless of whether the underlying user-specified parameter is a function. However, it is helpful to compare the behavior of object parameters that are specified as functions to that of user-specified noise and end-of-period processing functions.

Model parameters that are specified as functions are evaluated in the same way as user-specified random number (noise) generation functions. (For more information, see "Evaluating Different Types of Functions" on page 17-12.) Model parameters that are specified as functions are inputs to remove object constructors. User-specified noise and processing functions are *optional* inputs to simulation methods.

Because class constructors offer unique interfaces, and simulation methods of any given model have different implementation details, models often call parameter functions for validation purposes a different number of times, or in a different order, during object creation, simulation, and interpolation.

Therefore, although parameter functions, user-specified noise generation functions, and end-of-period processing functions all share the same interface and are validated at the same initial time and state (`obj.StartTime` and `obj.StartState`), parameter functions are not guaranteed to be invoked only once before simulation as noise generation and end-of-period processing functions are. In fact, parameter functions might not even be invoked the same number of times during a given Monte Carlo simulation process.

In most applications in which you specify parameters as functions, they are simple, deterministic functions of time and/or state. There is no need to count periods, count trials, or otherwise accumulate information or synchronize time.

However, if parameter functions require more sophisticated bookkeeping, the correct way to determine when a simulation has begun (or equivalently, to determine when model validation is complete) is to determine when the input time and/or state differs from the initial time and state (`obj.StartTime` and `obj.StartState`, respectively). Because the input time is a known scalar, detecting a change from the initial time is likely the best choice in most situations. This is a general mechanism that you can apply to any type of user-defined function.

Evaluating Different Types of Functions

It is useful to compare the evaluation rules of user-specified noise generation functions to those of end-of-period processing functions. These functions have the following in common:

- They both share the same general interface, returning a column vector of appropriate length when evaluated at the current time and state:

  $X_t = f(t, X_t)$

  $z_t = Z(t, X_t)$

- Before simulation, the simulation method itself calls each function once to validate the size of the output at the initial time and state, `obj.StartTime`, and `obj.StartState`, respectively.
- During simulation, the simulation method calls each function the same number of times: `NPERIODS` * `NSTEPS`.

However, there is an important distinction regarding the timing between these two types of functions. It is most clearly drawn directly from the generic `SDE` model:

$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$

This equation is expressed in continuous time, but the simulation methods approximate the model in discrete time as:

$X_{t+\Delta t} = X_t + F(t, X_t)\Delta t + G(t, X_t)\sqrt{\Delta t}Z(t, X_t)$

where $\Delta t > 0$ is a small (and not necessarily equal) period or time increment into the future. This equation is often referred to as a *Euler approximation*. All functions on the rightmost side are evaluated at the current time and state (t, X_t).

In other words, over the next small time increment, the simulation evolves the state vector based only on information available at the current time and state. In this sense, you can think of the noise function as a beginning-of-period function, or as a function evaluated from the left. This is also true for any user-supplied drift or diffusion function.

In contrast, user-specified end-of-period processing functions are applied only at the end of each simulation period or time increment. For more information about processing functions, see “Pricing Equity Options” on page 17-55.

Therefore, all simulation methods evaluate noise generation functions as:

$z_t = Z(t, X_t)$

for $t = t_0, t_0 + \Delta t, t_0 + 2\Delta t, \ldots, T - \Delta t$.

Yet simulation methods evaluate end-of-period processing functions as:

$$X_t = f(t, X_t)$$

for $t = t_0 + \Delta t, t_0 + 2\Delta t, \ldots, T$.

where t_0 and T are the initial time (taken from the object) and the terminal time (derived from inputs to the simulation method), respectively. These evaluations occur on all sample paths. Therefore, during simulation, noise functions are never evaluated at the final (terminal) time, and end-of-period processing functions are never evaluated at the initial (starting) time.

Drift and Diffusion

For example, an SDE with a linear drift rate has the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where *A* is an *NVARS*-by-`1` vector-valued function and *B* is an *NVARS*-by-*NVARS* matrix-valued function.

As an alternative, consider a drift-rate specification expressed in mean-reverting form:

$$F(t, X_t) = S(t)[L(t) - X_t]$$

where *S* is an *NVARS*-by-*NVARS* matrix-valued function of mean reversion speeds (that is, rates of mean reversion), and *L* is an *NVARS*-by-`1` vector-valued function of mean reversion levels (that is, long run average level).

Similarly, consider the following diffusion-rate specification:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where *D* is an *NVARS*-by-*NVARS* diagonal matrix-valued function. Each diagonal element of *D* is the corresponding element of the state vector raised to the corresponding element of an exponent *Alpha*, which is also an *NVARS*-by-`1` vector-valued function. *V* is an *NVARS*-by-*NBROWNS* matrix-valued function of instantaneous volatility rates. Each row of *V* corresponds to a particular state variable, and each column corresponds to a

particular Brownian source of uncertainty. V associates the exposure of state variables with sources of risk.

The parametric specifications for the drift and diffusion-rate functions associate parametric restrictions with familiar models derived from the general SDE class, and provide coverage for many models.

The class system and hierarchy of the SDE engine use industry-standard terminology to provide simplified interfaces for many models by placing user-transparent restrictions on drift and diffusion specifications. This design allows you to mix and match existing models, and customize drift-rate or diffusion-rate functions.

Available Models

For example, the following models are special cases of the general SDE model.

SDE Models

Model Name	Specification
Brownian Motion (BM)	$dX_t = A(t)dt + V(t)dW_t$
Geometric Brownian Motion (GBM)	$dX_t = B(t)X_t dt + V(t)X_t dW_t$
Constant Elasticity of Variance (CEV)	$dX_t = B(t)X_t dt + V(t)X_t^{\alpha(t)} dW_t$
Cox-Ingersoll-Ross (CIR)	$dX_t = S(t)(L(t) - X_t)dt + V(t)X_t^{\frac{1}{2}} dW_t$
Hull-White/Vasicek (HWV)	$dX_t = S(t)(L(t) - X_t)dt + V(t)dW_t$
Heston	$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$ $dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$

SDE Methods

The `sde` class provides default simulation and interpolation methods for all derived classes:

- `simulate`: High-level wrapper around the user-specified simulation method stored in the `Simulation` property
- `simByEuler`: Default Euler approximation simulation method
- `interpolate`: Stochastic interpolation method (that is, Brownian bridge)

The `gbm` and `hwv` classes feature an additional method, `simBySolution` for a `gbm` object and `simBySolution` for an `hwv` object, that simulates approximate solutions of diagonal-drift processes.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Class Hierarchy” on page 17-5

Base SDE Models

In this section...

Overview

The base `sde` class:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

represents the most general model.

Tip The `sde` class is not an abstract class. You can instantiate `sde` objects directly to extend the set of core models.

Constructing an `sde` object using the `sde` constructor requires the following inputs:

- A drift-rate function `F`. This function returns an `NVARS`-by-`1` drift-rate vector when run with the following inputs:
  - A real-valued scalar observation time t.
  - An `NVARS`-by-`1` state vector X_t.
- A diffusion-rate function `G`. This function returns an `NVARS`-by-`NBROWNS` diffusion-rate matrix when run with the inputs t and X_t.

Evaluating object parameters by passing (t, X_t) to a common, published interface allows most parameters to be referenced by a common input argument list that reinforces common method programming. You can use this simple function evaluation approach to model or construct powerful analytics, as in the following example.

Example: Base SDE Models

Construct an `sde` object `obj` using the `sde` constructor to represent a univariate geometric Brownian Motion model of the form:

$$dX_t = 0.1X_t dt + 0.3X_t dW_t$$

1. Create drift and diffusion functions that are accessible by the common *(t,X_t)* interface:

```
F = @(t,X) 0.1 * X;
G = @(t,X) 0.3 * X;
```

2. Pass the functions to the`sde` constructor to create an object `obj` of class `sde`:

```
obj = sde(F, G)     % dX = F(t,X)dt + G(t,X)dW

obj =
   Class SDE: Stochastic Differential Equation
   -------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

`obj` displays like a MATLAB structure, with the following information:

- The object's class
- A brief description of the object
- A summary of the dimensionality of the model

The object's displayed parameters are as follows:

- `StartTime`: The initial observation time (real-valued scalar)
- `StartState`: The initial state vector (`NVARS`-by-`1` column vector)
- `Correlation`: The correlation structure between Brownian process
- `Drift`: The drift-rate function *F(t,X_t)*
- `Diffusion`: The diffusion-rate function *G(t,X_t)*
- `Simulation`: The simulation method or function.

Of these displayed parameters, only `Drift` and `Diffusion` are required inputs.

The only exception to the (t, X_t) evaluation interface is `Correlation`. Specifically, when you enter `Correlation` as a function, the SDE engine assumes that it is a deterministic function of time, *C(t)*. This restriction on `Correlation` as a deterministic function of time allows Cholesky factors to be computed and stored before the formal simulation. This inconsistency dramatically improves run-time performance for dynamic correlation structures. If `Correlation` is stochastic, you can also include it within the simulation architecture as part of a more general random number generation function.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5

Drift and Diffusion Models

In this section...

Overview

Because base-level SDE objects accept drift and diffusion objects in lieu of functions accessible by (t, X_t), you can create `SDE` objects with combinations of customized drift or diffusion functions and objects. The `drift` and`diffusion` rate classes encapsulate the details of input parameters to optimize run-time efficiency for any given combination of input parameters.

Although `drift` and `diffusion` objects differ in the details of their representation, they are identical in their basic implementation and interface. They look, feel like, and are evaluated as functions:

- The `drift` class allows you to create drift-rate objects of the form:
  $F(t,X_t) = A(t) + B(t)X_t$

  where:

  - `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
  - `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.
- Similarly, the `diffusion` class allows you to create diffusion-rate objects:
  $G(t,X_t) = D(t,X_t^{\alpha(t)})V(t)$

  where:

  - `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
  - Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.

- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: Drift and Diffusion Rates

In this example, you create `drift` and `diffusion` rate objects using the `drift` and `diffusion` constructors to create the same model as in "Example: Base SDE Models" on page 17-16.

Create a drift-rate function `F` and a diffusion-rate function `G`:

```
F = drift(0, 0.1)       % Drift rate function F(t,X)

F =
   Class DRIFT: Drift Rate Specification
   -------------------------------------
      Rate: drift rate function F(t,X(t))
         A: 0
         B: 0.1

G = diffusion(1, 0.3)  % Diffusion rate function G(t,X)

G =
   Class DIFFUSION: Diffusion Rate Specification
   ---------------------------------------------
       Rate: diffusion rate function G(t,X(t))
      Alpha: 1
      Sigma: 0.3
```

Each object displays like a MATLAB structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because `drift` and `diffusion` classes create model components rather than models. Neither `F` nor `G` contains enough information to characterize the dimensionality of a problem.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

Again, `Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Example: SDEDDO Models

The `sdeddo` class derives from the base`sde` class. To use this class, you must pass drift and diffusion-rate objects to the `sdeddo` constructor.

1. Create `drift` and `diffusion` rate objects using the `drift` and `diffusion` constructors:

```
F = drift(0, 0.1);       % Drift rate function F(t,X)
G = diffusion(1, 0.3);   % Diffusion rate function G(t,X)
```

2. Pass these objects to the `sdeddo` constructor:

```
obj = sdeddo(F, G)       % dX = F(t,X)dt + G(t,X)dW
```

```
obj =
   Class SDEDDO: SDE from Drift and Diffusion Objects
   ---------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ---------------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
```

```
      Drift: drift rate function F(t,X(t))
  Diffusion: diffusion rate function G(t,X(t))
 Simulation: simulation method/function simByEuler
          A: 0
          B: 0.1
      Alpha: 1
      Sigma: 0.3
```

In this example, the object displays the additional parameters associated with input drift and diffusion objects.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- "Base SDE Models" on page 17-16
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

More About

- "SDEs" on page 17-2
- "SDE Models" on page 17-8
- "SDE Class Hierarchy" on page 17-5

Linear Drift Models

In this section...

Overview

The `sdeld` class derives from the `sdeddo` class. These objects allow you to simulate correlated paths of `NVARS` state variables expressed in linear drift-rate form:

$$dX_t = (A(t) + B(t)X_t)dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

`sdeld` objects provide a parametric alternative to the mean-reverting drift form, as discussed in “Example: SDEMRD Models” on page 17-28. They also provide an alternative interface to the `sdeddo` parent class, because you can create an object without first having to create its drift and diffusion-rate components.

Example: SDELD Models

Create the same model as in “Example: Base SDE Models” on page 17-16 using the `sdeld` constructor:

```
obj = sdeld(0, 0.1, 1, 0.3) % (A, B, Alpha, Sigma)
```

```
obj =
   Class SDELD: SDE with Linear Drift
   ---------------------------------------
     Dimensions: State = 1, Brownian = 1
   ---------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 0
              B: 0.1
          Alpha: 1
          Sigma: 0.3
```

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5

Parametric Models

In this section...

Creating Brownian Motion (BM) Models

The Brownian Motion (BM) model (`bm`) derives directly from the linear drift (`sdeld`) class:

$$dX_t = \mu(t)dt + V(t)dW_t$$

Example: BM Models

Create a univariate Brownian motion (`bm`) object to represent the model using the `bm` constructor:

$dX_t = 0.3dW_t$.

```
obj = bm(0, 0.3) % (A = Mu, Sigma)
```

```
obj =
   Class BM: Brownian Motion
   ---------------------------------------
     Dimensions: State = 1, Brownian = 1
   ---------------------------------------
      StartTime: 0
     StartState: 0
    Correlation: 1
          Drift: drift rate function F(t,X(t))
```

```
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
             Mu: 0
          Sigma: 0.3
```

`bm` objects display the parameter `A` as the more familiar `Mu`.

The`bm` class also provides an overloaded Euler simulation method that improves run-time performance in certain common situations. This specialized method is invoked automatically only if *all* the following conditions are met:

- The expected drift, or trend, rate `Mu` is a column vector.
- The volatility rate, `Sigma`, is a matrix.
- No end-of-period adjustments and/or processes are made.
- If specified, the random noise process `Z` is a three-dimensional array.
- If `Z` is unspecified, the assumed Gaussian correlation structure is a double matrix.

Creating Constant Elasticity of Variance (CEV) Models

The Constant Elasticity of Variance (CEV) model (`cev`) also derives directly from the linear drift (`sdeld`) class:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

The `cev` class constrains A to an `NVARS`-by-`1` vector of zeros. D is a diagonal matrix whose elements are the corresponding element of the state vector X, raised to an exponent $\alpha(t)$.

Example: Univariate CEV Models

Create a univariate `cev` object to represent the model using the `cev` constructor:

$$dX_t = 0.25X_t + 0.3X_t^{\frac{1}{2}}dW_t.$$

```
obj = cev(0.25, 0.5, 0.3) % (B = Return, Alpha, Sigma)
```

```
obj =
   Class CEV: Constant Elasticity of Variance
   -------------------------------------------
     Dimensions: State = 1, Brownian = 1
```

```
-----------------------------------------
   StartTime: 0
  StartState: 1
 Correlation: 1
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
      Return: 0.25
       Alpha: 0.5
       Sigma: 0.3
```

`cev` and `gbm` objects display the parameter `B` as the more familiar `Return`.

Creating Geometric Brownian Motion (GBM) Models

The Geometric Brownian Motion (GBM) model (`gbm`) derives directly from the CEV (`cev`) model:

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$$

Compared to the `cev` object, a `gbm` object constrains all elements of the *alpha* exponent vector to one such that D is now a diagonal matrix with the state vector X along the main diagonal.

The `gbm` class also provides two simulation methods that can be used by separable models:

- An overloaded Euler simulation method that improves run-time performance in certain common situations. This specialized method is invoked automatically only if *all* the following conditions are true:
  - The expected rate of return (`Return`) is a diagonal matrix.
  - The volatility rate (`Sigma`) is a matrix.
  - No end-of-period adjustments/processes are made.
  - If specified, the random noise process `Z` is a three-dimensional array.
  - If `Z` is unspecified, the assumed Gaussian correlation structure is a double matrix.
- An approximate analytic solution (`simBySolution`) obtained by applying a Euler approach to the transformed (using Ito's formula) logarithmic process. In general, this is *not* the exact solution to this GBM model, as the probability distributions of the simulated and true state vectors are identical *only* for piecewise constant parameters.

If the model parameters are piecewise constant over each observation period, the state vector X_t is lognormally distributed and the simulated process is exact for the observation times at which X_t is sampled.

Example: Univariate GBM Models

Create a univariate `gbm` object to represent the model using the `gbm` constructor:
$dX_t = 0.25X_t dt + 0.3X_t dW_t$

```
obj = gbm(0.25, 0.3)  % (B = Return, Sigma)
```

```
obj =
   Class GBM: Generalized Geometric Brownian Motion
   -------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.25
          Sigma: 0.3
```

Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models

The `sdemrd` class derives directly from the `sdeddo` class. It provides an interface in which the drift-rate function is expressed in mean-reverting drift form:
$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\alpha(t)})V(t)dW_t$

`sdemrd` objects provide a parametric alternative to the linear drift form by reparameterizing the general linear drift such that:
$A(t) = S(t)L(t), B(t) = -S(t)$

Example: SDEMRD Models

Create an `sdemrd` object using the `sdemrd` constructor with a square root exponent to represent the model:

$$dX_t = 0.2(0.1 - X_t)dt + 0.05X_t^{\frac{1}{2}}dW_t.$$

```
obj = sdemrd(0.2, 0.1, 0.5, 0.05)
```

```
obj =
   Class SDEMRD: SDE with Mean-Reverting Drift
   -------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Alpha: 0.5
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2

    % (Speed, Level, Alpha, Sigma)
```

`sdemrd` objects display the familiar `Speed` and `Level` parameters instead of `A` and `B`.

Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models

The Cox-Ingersoll-Ross (CIR) short rate class, `cir`, derives directly from the SDE with mean-reverting drift (`sdemrd`) class:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\frac{1}{2}})V(t)dW_t$$

where D is a diagonal matrix whose elements are the square root of the corresponding element of the state vector.

Example: CIR Models

Create a `cir` object using the `cir` constructor to represent the same model as in "Example: SDEMRD Models" on page 17-28:

```
obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
obj =
   Class CIR: Cox-Ingersoll-Ross
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

Although the last two objects are of different classes, they represent the same mathematical model. They differ in that you create the `cir` object by specifying only three input arguments. This distinction is reinforced by the fact that the `Alpha` parameter does not display – it is defined to be `1/2`.

Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models

The Hull-White/Vasicek (HWV) short rate class, `hwv`, derives directly from SDE with mean-reverting drift (`sdemrd`) class:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$$

Example: HWV Models

Using the same parameters as in the previous example, create an `hwv` object using the `hwv` constructor to represent the model:

$$dX_t = 0.2(0.1 - X_t)dt + 0.05dW_t.$$

```
obj = hwv(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)

obj =
   Class HWV: Hull-White/Vasicek
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
```

```
Correlation: 1
      Drift: drift rate function F(t,X(t))
  Diffusion: diffusion rate function G(t,X(t))
 Simulation: simulation method/function simByEuler
      Sigma: 0.05
      Level: 0.1
      Speed: 0.2
```

`cir` and `hwv` constructors share the same interface and display methods. The only distinction is that `cir` and `hwv` model objects constrain `Alpha` exponents to `1/2` and `0`, respectively. Furthermore, the`hwv` class also provides an additional method that simulates approximate analytic solutions (`simBySolution`) of separable models. This method simulates the state vector X_t using an approximation of the closed-form solution of diagonal drift `HWV` models. Each element of the state vector X_t is expressed as the sum of `NBROWNS` correlated Gaussian random draws added to a deterministic time-variable drift.

When evaluating expressions, all model parameters are assumed piecewise constant over each simulation period. In general, this is *not* the exact solution to this `hwv` model, because the probability distributions of the simulated and true state vectors are identical *only* for piecewise constant parameters. If $S(t,X_t)$, $L(t,X_t)$, and $V(t,X_t)$ are piecewise constant over each observation period, the state vector X_t is normally distributed, and the simulated process is exact for the observation times at which X_t is sampled.

Hull-White vs. Vasicek Models

Many references differentiate between Vasicek models and Hull-White models. Where such distinctions are made, Vasicek parameters are constrained to be constants, while Hull-White parameters vary deterministically with time. Think of Vasicek models in this context as constant-coefficient Hull-White models and equivalently, Hull-White models as time-varying Vasicek models. However, from an architectural perspective, the distinction between static and dynamic parameters is trivial. Since both models share the same general parametric specification as previously described, a single`hwv` class encompasses the models.

Creating Heston Stochastic Volatility Models

The Heston (`heston`) class derives directly from SDE from Drift and Diffusion (`sdeddo`) class. Each Heston model is a bivariate composite model, consisting of two coupled univariate models:

$$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

$$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

"Equation 17-5" on page 17-32 is typically associated with a price process.
"Equation 17-6" on page 17-32 represents the evolution of the price process' variance.
Models of type `heston` are typically used to price equity options.

Example: Heston Models

Create a `heston` object using the `heston` constructor to represent the model:

$$dX_{1t} = 0.1X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$
$$dX_{2t} = 0.2[0.1 - X_{2t}]dt + 0.05\sqrt{X_{2t}}dW_{2t}$$

```
obj = heston (0.1, 0.2, 0.1, 0.05)
```

```
obj =
   Class HESTON: Heston Bivariate Stochastic Volatility
   -------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   -------------------------------------------------
      StartTime: 0
     StartState: 1 (2x1 double array)
    Correlation: 2x2 diagonal double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.1
          Speed: 0.2
          Level: 0.1
     Volatility: 0.05
```

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5

Simulating Equity Prices

In this section...

Simulating Multidimensional Market Models

This example compares alternative implementations of a separable multivariate geometric Brownian motion process that is often referred to as a *multidimensional market model*. It simulates sample paths of an equity index portfolio using `sde`, `sdeddo`, `sdeld`, `cev`, and `gbm` objects.

The market model to simulate is:

$$dX_t = \mu X_t dt + D(X_t)\sigma dW_t$$

where:

- μ is a diagonal matrix of expected index returns.
- D is a diagonal matrix with X_t along the diagonal.
- σ is a diagonal matrix of standard deviations of index returns.

Representing Market Models Using SDE Objects

Create an `sde` object using the `sde` constructor to represent the equity market model.

1 Load the `Data_GlobalIdx2` data set:

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
Dataset.NIK Dataset.FTSE Dataset.SP];
```

2 Convert daily prices to returns:

```
returns =  tick2ret(prices);
```

3 Compute data statistics to input to simulation methods:

```
nVariables  = size(returns, 2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));
```

4 Create simple anonymous drift and diffusion functions accessible by (t, X_t):

```
F = @(t,X) diag(expReturn) * X;
G = @(t,X) diag(X) * diag(sigma);
```

5 Use these functions with the `sde` constructor to create an `sde` object to represent the market model in "Equation 17-7" on page 17-34:

```
SDE = sde(F, G, 'Correlation', correlation, 'StartState', X)
```

```
SDE =
   Class SDE: Stochastic Differential Equation
   -------------------------------------------
     Dimensions: State = 6, Brownian = 6
   -------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

The `sde` constructor requires additional information to determine the dimensionality of the model, because the functions passed to the`sde` constructor are known only by their (t, X_t) interface. In other words, the`sde` constructor requires only two inputs: a drift-rate function and a diffusion-rate function, both accessible by passing the sample time and the corresponding state vector (t, X_t).

In this case, this information is insufficient to determine unambiguously the dimensionality of the state vector and Brownian motion. You resolve the dimensionality by specifying an initial state vector, `StartState`. The SDE engine has assigned the default simulation method, `simByEuler`, to the `Simulation` parameter.

Representing Market Models Using SDEDDO Objects

Create an `sdeddo` object using the `sdeddo` constructor to represent the market model in "Equation 17-7" on page 17-34:

1 Load the `Data_GlobalIdx2` data set:

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
Dataset.NIK Dataset.FTSE Dataset.SP];
```

2 Convert daily prices to returns:

```
returns =  tick2ret(prices);
```

3 Compute data statistics to input to simulation methods:

```
nVariables  = size(returns, 2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
```

4 Create `drift` and `diffusion` objects using the`drift` and `diffusion` constructors:

```
F = drift(zeros(nVariables,1), diag(expReturn))
```

```
F =
   Class DRIFT: Drift Rate Specification
   -------------------------------------
      Rate: drift rate function F(t,X(t))
         A: 6x1 double array
         B: 6x6 diagonal double array
```

```
G = diffusion(ones(nVariables,1), diag(sigma))
```

```
G =
   Class DIFFUSION: Diffusion Rate Specification
   ---------------------------------------------
       Rate: diffusion rate function G(t,X(t))
      Alpha: 6x1 double array
      Sigma: 6x6 diagonal double array
```

5 Pass the `drift` and `diffusion` objects to the `sdeddo` constructor:

```
SDEDDO = sdeddo(F, G, 'Correlation', correlation, ...
'StartState', 100)
```

```
SDEDDO =
   Class SDEDDO: SDE from Drift and Diffusion Objects
```

```
-------------------------------------------------
  Dimensions: State = 6, Brownian = 6
-------------------------------------------------
   StartTime: 0
  StartState: 100 (6x1 double array)
 Correlation: 6x6 double array
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
           A: 6x1 double array
           B: 6x6 diagonal double array
       Alpha: 6x1 double array
       Sigma: 6x6 diagonal double array
```

The `sdeddo` constructor requires two input objects that provide more information than the two functions from step 4 on page 17-35 of "Representing Market Models Using SDE Objects" on page 17-34. Thus, the dimensionality is more easily resolved. In fact, the initial price of each index is as a scalar (`StartState` = `100`). This is in contrast to the`sde` constructor, which required an explicit state vector to uniquely determine the dimensionality of the problem.

Once again, the class of each object is clearly identified, and parameters display like fields of a structure. In particular, the `Rate` parameter of drift and diffusion objects is identified as a callable function of time and state, $F(t,X_t)$ and $G(t,X_t)$, respectively. The additional parameters, `A`, `B`, `Alpha`, and `Sigma`, are arrays of appropriate dimension, indicating static (non-time-varying) parameters. In other words, $A(t,X_t)$, $B(t,X_t)$, $Alpha(t,X_t)$, and $Sigma(t,X_t)$ are constant functions of time and state.

Representing Market Models Using SDELD, CEV, and GBM Objects

Create `sdeld`, `cev`, and`gbm` objects to represent the market model in "Equation 17-7" on page 17-34.

1 Load the `Data_GlobalIdx2` data set:

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];
```

2 Convert daily prices to returns:

```
returns =  tick2ret(prices);
```

3 Compute data statistics to input to simulation methods:

```
nVariables  = size(returns, 2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));
```

4 Create an `sdeld` object using the `sdeld` constructor:

```
SDELD = sdeld(zeros(nVariables,1), diag(expReturn), ...
    ones(nVariables,1), diag(sigma),'Correlation', ...
    correlation, 'StartState', X)
```

```
SDELD =
   Class SDELD: SDE with Linear Drift
   ----------------------------------------
     Dimensions: State = 6, Brownian = 6
   ----------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 6x1 double array
              B: 6x6 diagonal double array
          Alpha: 6x1 double array
          Sigma: 6x6 diagonal double array
```

5 Create a `cev` object using the `cev` constructor:

```
CEV = cev(diag(expReturn), ones(nVariables,1), ...
    diag(sigma), 'Correlation', correlation, ...
    'StartState', X)
```

```
CEV =
   Class CEV: Constant Elasticity of Variance
   ----------------------------------------
     Dimensions: State = 6, Brownian = 6
   ----------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
```

```
     Simulation: simulation method/function simByEuler
         Return: 6x6 diagonal double array
          Alpha: 6x1 double array
          Sigma: 6x6 diagonal double array
```

6 Create a `gbm` object using the `gbm` constructor:

```
GBM = gbm(diag(expReturn), diag(sigma), 'Correlation', ...
    correlation, 'StartState', X)
```

```
GBM =
   Class GBM: Generalized Geometric Brownian Motion
   -------------------------------------------------
     Dimensions: State = 6, Brownian = 6
   -------------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 6x6 diagonal double array
          Sigma: 6x6 diagonal double array
```

Note the succession of interface restrictions:

- `sdeld` objects require you to specify `A`, `B`, `Alpha`, and `Sigma`.
- `cev` objects require you to specify `Return`, `Alpha`, and `Sigma`.
- `gbm` objects require you to specify only `Return` and `Sigma`.

However, all three objects represent the same multidimensional market model.

Also, `cev` and `gbm` objects display the underlying parameter `B` derived from the `sdeld` class as `Return`. This is an intuitive name commonly associated with equity models.

Simulating Equity Markets Using the Default Simulate Method

1 Load the `Data_GlobalIdx2` data set and use the `sde` constructor to specify the SDE model as in “Representing Market Models Using SDE Objects” on page 17-34.

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];
```

```
returns =  tick2ret(prices);

nVariables  = size(returns,2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));

F = @(t,X) diag(expReturn)* X;
G = @(t,X) diag(X) * diag(sigma);

SDE = sde(F, G, 'Correlation', ...
    correlation, 'StartState', X);
```

2 Simulate a single path of correlated equity index prices over one calendar year (defined as approximately 250 trading days) using the default`simulate` method:

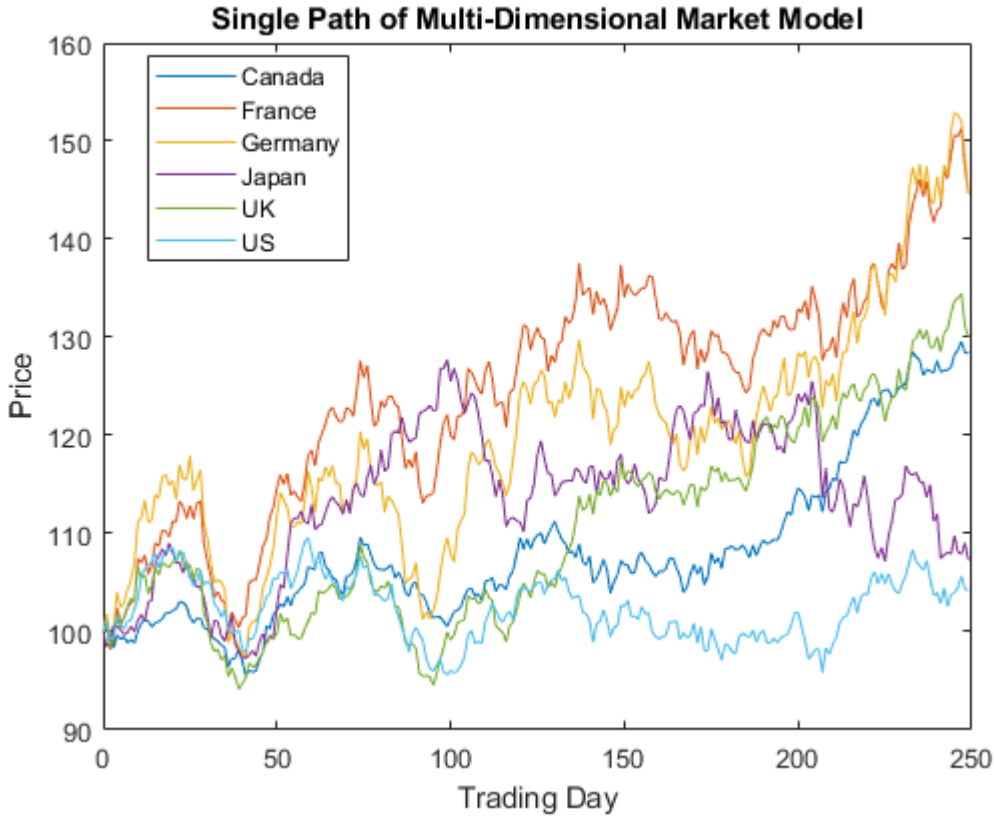
```
nPeriods = 249;      % # of simulated observations
dt       =   1;      % time increment = 1 day
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt);

whos S
```

```
  Name        Size             Bytes  Class     Attributes

  S         250x6              12000  double
```

The output array `S` is a 250-by-6 = (`NPERIODS + 1`-by-`nVariables`-by-`1`) array with the same initial value, `100`, for all indices. Each row of `S` is an observation of the state vector X_t at time t.

3 Plot the simulated paths.

```
plot(T, S), xlabel('Trading Day'), ylabel('Price')
title('Single Path of Multi-Dimensional Market Model')
legend({'Canada' 'France' 'Germany' 'Japan' 'UK' 'US'}, ...
    'Location', 'Best')
```



Simulating Equity Markets Using the SimByEuler Method

Because `simByEuler` is a valid simulation method, you can call it directly, overriding the `Simulation` parameter's current method or function (which in this case is `simByEuler`). The following statements produce the same price paths as generated in "Simulating Equity Markets Using the Default Simulate Method" on page 17-39:

1 Load the `Data_GlobalIdx2` data set and use the `sde` constructor to specify the SDE model as in "Representing Market Models Using SDE Objects" on page 17-34.

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];
```

```
returns =  tick2ret(prices);

nVariables  = size(returns,2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));

F = @(t,X) diag(expReturn)* X;
G = @(t,X) diag(X) * diag(sigma);

SDE = sde(F, G, 'Correlation', ...
    correlation, 'StartState', X);
```

2 Simulate a single path using `simByEuler`.

```
nPeriods = 249;       % # of simulated observations
dt       =   1;       % time increment = 1 day
rng(142857,'twister')
[S,T] = simByEuler(SDE, nPeriods, 'DeltaTime', dt);
```

3 Simulate 10 trials with the same initial conditions, and examine `S`:

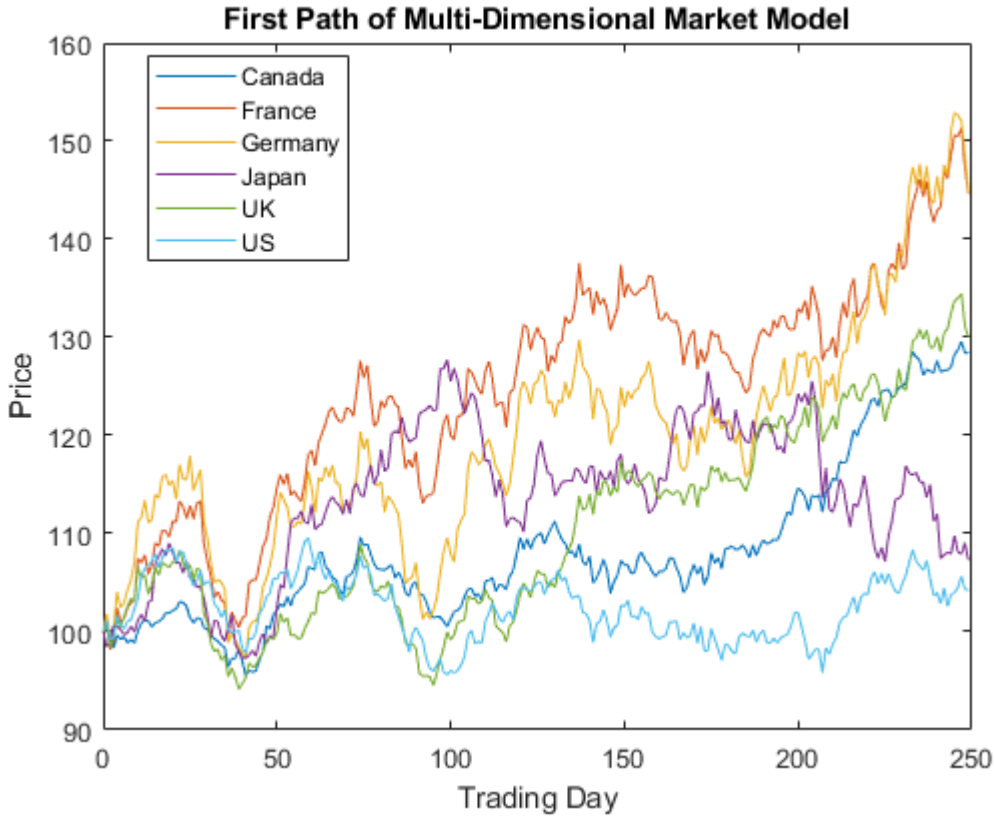
```
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);

whos S
```

```
  Name        Size                Bytes  Class     Attributes

  S         250x6x10             120000  double
```

Now the output array `S` is an `NPERIODS + 1`-by-`nVariables`-by-`nTrials` time series array.

4 Plot the first paths.

```
plot(T, S(:,:,1)), xlabel('Trading Day'), ylabel('Price')
title('First Path of Multi-Dimensional Market Model')
legend({'Canada' 'France' 'Germany' 'Japan' 'UK' 'US'},...
    'Location', 'Best')
```



The first realization of `S` is identical to the paths in the plot.

Simulating Equity Markets Using GBM Simulation Methods

Finally, consider simulation using `GBM` simulation methods. Separable `GBM` models have two specific simulation methods:

- An overloaded Euler simulation method, `simByEuler`, designed for optimal performance
- A method, `simBySolution`, provides an approximate solution of the underlying stochastic differential equation, designed for accuracy

1 Load the `Data_GlobalIdx2` data set and use the `sde` constructor to specify the SDE model as in "Representing Market Models Using SDE Objects" on page 17-34, and the GBM model as in "Representing Market Models Using SDELD, CEV, and GBM Objects" on page 17-37.

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];

returns =  tick2ret(prices);

nVariables  = size(returns,2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));

F = @(t,X) diag(expReturn)* X;
G = @(t,X) diag(X) * diag(sigma);

SDE = sde(F, G, 'Correlation', ...
    correlation, 'StartState', X);

GBM = gbm(diag(expReturn),diag(sigma), 'Correlation', ...
    correlation, 'StartState', X);
```

2 To illustrate the performance benefit of the overloaded Euler approximation method, increase the number of trials to `10000`.

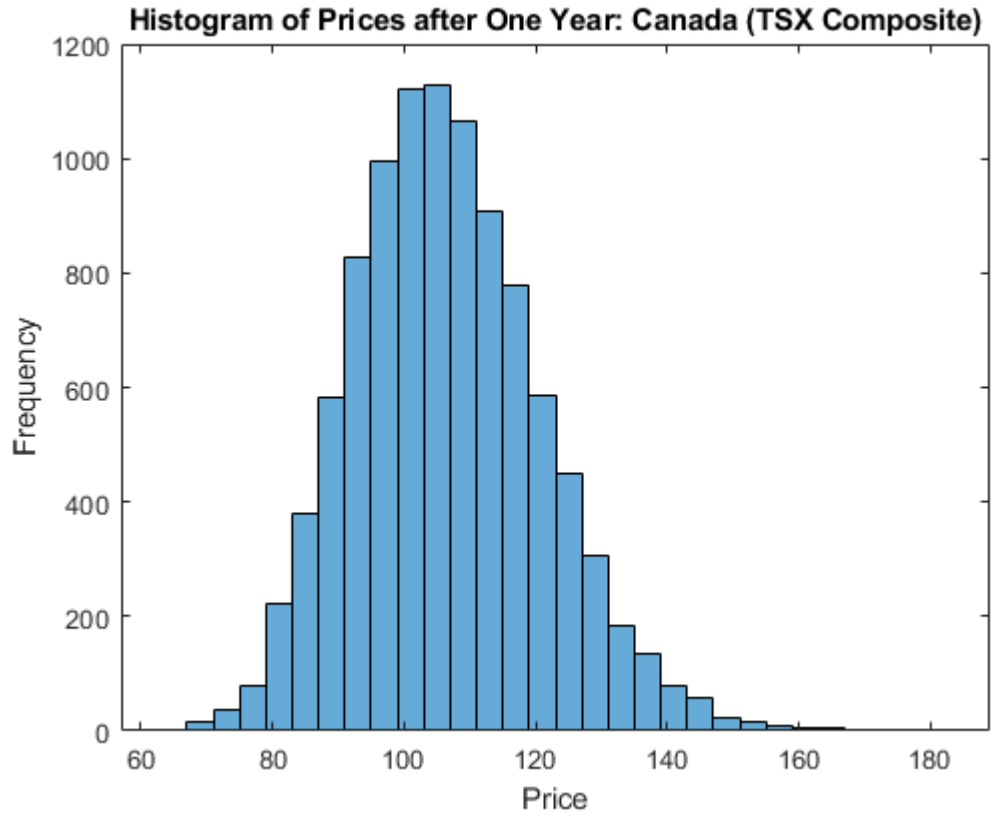
```
nPeriods = 249;       % # of simulated observations
dt       =   1;       % time increment = 1 day
rng(142857,'twister')
[X,T] = simulate(GBM, nPeriods, 'DeltaTime', dt, ...
    'nTrials', 10000);

whos X
```

```
  Name        Size                     Bytes  Class     Attributes

  X         250x6x10000            120000000  double
```

The output `X` is a much larger time series array.

3 Using this sample size, examine the terminal distribution of Canada's TSX Composite to verify qualitatively the lognormal character of the data.

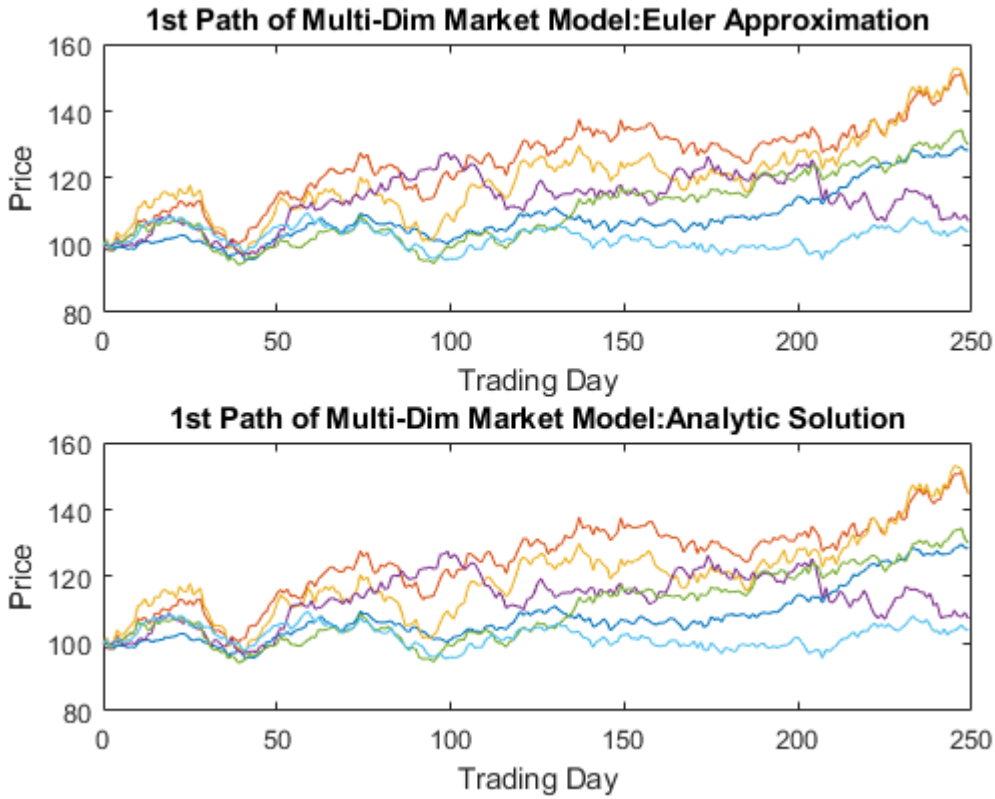
```
histogram(squeeze(X(end,1,:)), 30), xlabel('Price'), ylabel('Frequency')
title('Histogram of Prices after One Year: Canada (TSX Composite)')
```



4 Simulate 10 trials of the solution and plot the first trial:

```
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);
rng(142857,'twister')
[X,T] = simBySolution(GBM, nPeriods,...
    'DeltaTime', dt, 'nTrials', 10);
subplot(2,1,1)
plot(T, S(:,:,1)), xlabel('Trading Day'),ylabel('Price')
```

```
title('1st Path of Multi-Dim Market Model:Euler Approximation')
subplot(2,1,2)
plot(T, X(:,:,1)), xlabel('Trading Day'),ylabel('Price')
title('1st Path of Multi-Dim Market Model:Analytic Solution')
```

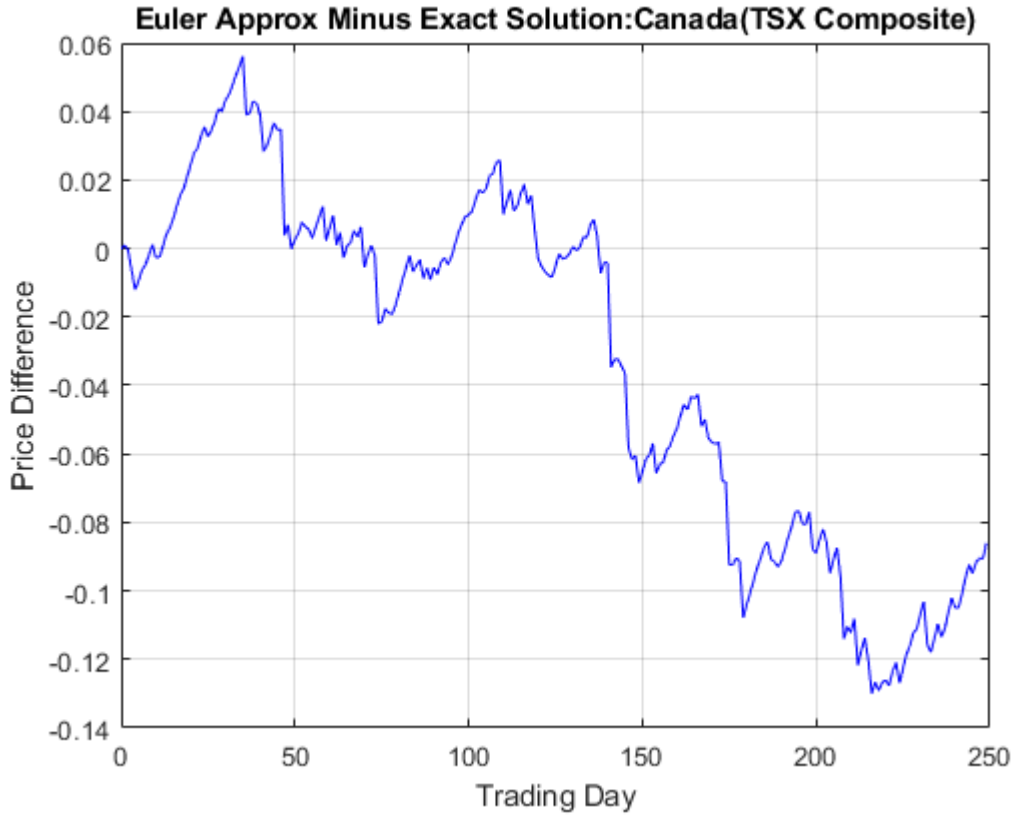


In this example, all parameters are constants, and `simBySolution` does indeed sample the exact solution. The details of a single index for any given trial show that the price paths of the Euler approximation and the exact solution are close, but not identical.

5 The following plot illustrates the difference between the two methods:

```
subplot(1,1,1)
plot(T, S(:,1,1) - X(:,1,1), 'blue'), grid('on')
```

```
xlabel('Trading Day'), ylabel('Price Difference')
title('Euler Approx Minus Exact Solution:Canada(TSX Composite)')
```



The `simByEuler` Euler approximation literally evaluates the stochastic differential equation directly from the equation of motion, for some suitable value of the `dt` time increment. This simple approximation suffers from discretization error. This error can be attributed to the discrepancy between the choice of the `dt` time increment and what in theory is a continuous-time parameter.

The discrete-time approximation improves as `DeltaTime` approaches zero. The Euler method is often the least accurate and most general method available. All models shipped in the simulation suite have this method.

In contrast, the`simBySolution` method provides a more accurate description of the underlying model. This method simulates the price paths by an approximation of the closed-form solution of separable models. Specifically, it applies a Euler approach to a transformed process, which in general is not the exact solution to this `GBM` model. This is because the probability distributions of the simulated and true state vectors are identical only for piecewise constant parameters.

When all model parameters are piecewise constant over each observation period, the simulated process is exact for the observation times at which the state vector is sampled. Since all parameters are constants in this example,`simBySolution` does indeed sample the exact solution.

For an example of how to use `simBySolution` to optimize the accuracy of solutions, see "Optimizing Accuracy: About Solution Precision and Error" on page 17-78.

Inducing Dependence and Correlation

This example illustrates two techniques that induce dependence between individual elements of a state vector. It also illustrates the interaction between `Sigma` and `Correlation`.

The first technique generates correlated Gaussian variates to form a Brownian motion process with dependent components. These components are then weighted by a diagonal volatility or exposure matrix `Sigma`.

The second technique generates independent Gaussian variates to form a standard Brownian motion process, which is then weighted by the lower Cholesky factor of the desired covariance matrix. Although these techniques can be used on many models, the relationship between them is most easily illustrated by working with a separable `GBM` model (see Simulating Equity Prices Using GBM Simulation Methods on page 17-43). The market model to simulate is:

$$dX_t = \mu X_t dt + \sigma X_t dW_t$$

where μ is a diagonal matrix.

1 Load the data set:

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];
```

2. Convert the daily prices to returns:

```
returns =  tick2ret(prices);
```

3. Specify `Sigma` and `Correlation` using the first technique:

   a. Using the first technique, specify `Sigma` as a diagonal matrix of asset return standard deviations:

```
expReturn   = diag(mean(returns));  % expected return vector
sigma       = diag(std(returns));   % volatility of returns
```

   b. Specify `Correlation` as the sample correlation matrix of those returns. In this case, the components of the Brownian motion are dependent:

```
correlation = corrcoef(returns);
GBM1        = gbm(expReturn,sigma,'Correlation',...
                  correlation);
```

4. Specify `Sigma` and `Correlation` using the second technique:

   a. Using the second technique, specify `Sigma` as the lower Cholesky factor of the asset return covariance matrix:

```
covariance = cov(returns);
sigma      = cholcov(covariance)';
```

   b. Set `Correlation` to an identity matrix:

```
GBM2       = gbm(expReturn,sigma);
```

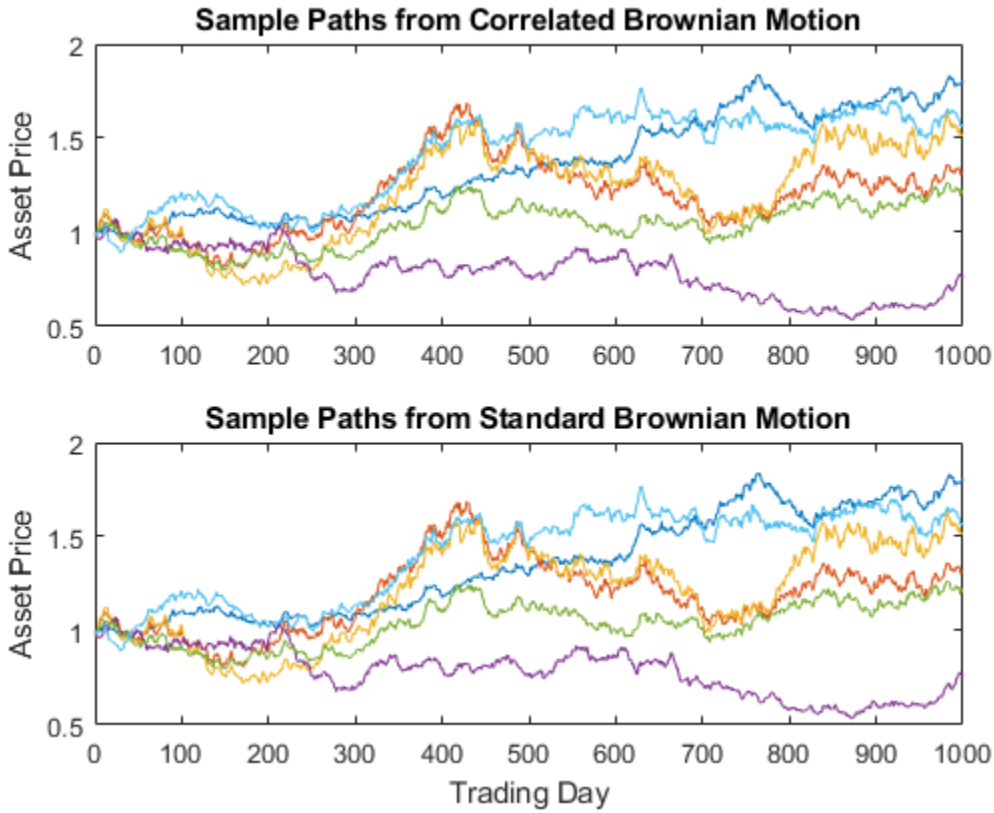
      Here, `sigma` captures both the correlation and magnitude of the asset return uncertainty. In contrast to the first technique, the components of the Brownian motion are independent. Also, this technique accepts the default assignment of an identity matrix to `Correlation`, and is more straightforward.

5. Simulate a single trial of 1000 observations (roughly four years of daily data) using both techniques. By default, all state variables start at `1`:

```
rng(22814,'twister')
[X1,T] = simByEuler(GBM1,1000);  % correlated Brownian motion
rng(22814,'twister')
[X2,T] = simByEuler(GBM2,1000);  % standard Brownian motion
```

When based on the same initial random number state, each technique generates identical asset price paths:

```
subplot(2,1,1)
plot(T, X1)
title('Sample Paths from Correlated Brownian Motion')
ylabel('Asset Price')
subplot(2,1,2)
plot(T, X2)
title('Sample Paths from Standard Brownian Motion')
xlabel('Trading Day')
ylabel('Asset Price')
```



Dynamic Behavior of Market Parameters

As discussed in "Creating SDE Objects" on page 17-8, object parameters may be evaluated as if they are MATLAB functions accessible by a common interface. This accessibility provides the impression of dynamic behavior regardless of whether the underlying parameters are truly time-varying. Furthermore, because parameters are accessible by a common interface, seemingly simple linear constructs may in fact represent complex, nonlinear designs.

For example, consider a univariate geometric Brownian motion (GBM) model of the form:

$$dX_t = \mu(t)X_t dt + \sigma(t)X_t dW_t$$

In this model, the return, $\mu(t)$, and volatility, $\sigma(t)$, are generally dynamic parameters of time alone. However, when creating a `gbm` object to represent the underlying model, such dynamic behavior must be accessible by the common (t, X_t) interface. This reflects the fact that `GBM` models (and others) are restricted parameterizations that derive from the general `SDE` class.

As a convenience, you can specify parameters of restricted models, such as `GBM` models, as traditional MATLAB arrays of appropriate dimension. In this case, such arrays represent a static special case of the more general dynamic situation accessible by the (t, X_t) interface.

Moreover, when you enter parameters as functions, object constructors can verify that they return arrays of correct size by evaluating them at the initial time and state. Otherwise, object constructors have no knowledge of any particular functional form.

The following example illustrates a technique that includes dynamic behavior by mapping a traditional MATLAB time series array to a callable function with a (t, X_t) interface. It also compares the function with an otherwise identical model with constant parameters.

Because time series arrays represent dynamic behavior that must be captured by functions accessible by the (t, X_t) interface, you need utilities to convert traditional time series arrays into callable functions of time and state. The following example shows how to do this using the conversion function `ts2func` (time series to function).

1 **Load the data.** Load a daily historical data set containing 3-month Euribor rates and closing index levels of France's CAC 40 spanning the time interval February 7, 2001 to April 24, 2006:

```
load Data_GlobalIdx2
```

2 **Simulate risk-neutral sample paths.** Simulate risk-neutral sample paths of the CAC 40 index using a geometric Brownian motion (GBM) model:

$$dX_t = r(t)X_t dt + \sigma X_t dW_t$$

where *r(t)* represents evolution of the risk-free rate of return.

Furthermore, assume that you need to annualize the relevant information derived from the daily data (annualizing the data is optional, but is useful for comparison to other examples), and that each calendar year comprises 250 trading days:

```
dt      = 1/250;
returns = tick2ret(Dataset.CAC);
sigma   = std(returns)*sqrt(250);
yields  = Dataset.EB3M;
yields  = 360*log(1 + yields);
```

3 **Compare the sample paths from two risk-neutral historical simulation approaches.** Compare the resulting sample paths obtained from two risk-neutral historical simulation approaches, where the daily Euribor yields serve as a proxy for the risk-free rate of return.

a The first approach specifies the risk-neutral return as the sample average of Euribor yields, and therefore assumes a constant (non-dynamic) risk-free return:

```
nPeriods = length(yields);  % Simulated observations
rng(5713,'twister')
obj    = gbm(mean(yields),diag(sigma),'StartState',100)

obj =
   Class GBM: Generalized Geometric Brownian Motion
   -------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------------
      StartTime: 0
     StartState: 100
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.0278117
          Sigma: 0.231906

[X1,T] = simulate(obj,nPeriods,'DeltaTime',dt);
```

b In contrast, the second approach specifies the risk-neutral return as the historical time series of Euribor yields. It therefore assumes a dynamic, yet deterministic, rate of return; this example does not illustrate stochastic interest rates. To illustrate this dynamic effect, use the `ts2func` utility:

```
r = ts2func(yields,'Times',(0:nPeriods - 1)');
```

`ts2func` packages a specified time series array inside a callable function of time and state, and synchronizes it with an optional time vector. For instance:

```
r(0,100)
```

```
ans = 0.0470
```

evaluates the function at ($t = 0$, $X_t = 100$) and returns the first observed Euribor yield. However, you can also evaluate the resulting function at any intermediate time t and state X_t:

```
r(7.5,200)
```

```
ans = 0.0472
```

Furthermore, the following command produces the same result when called with time alone:

```
r(7.5)
```

```
ans = 0.0472
```

The equivalence of these last two commands highlights some important features.

When you specify parameters as functions, they must evaluate properly when passed a scalar, real-valued sample time (t), and an *NVARS*-by-`1` state vector (X_t). They must also generate an array of appropriate dimensions, which in the first case is a scalar constant, and in the second case is a scalar, piecewise constant function of time alone.

You are not required to use either time (t) or state (X_t). In the current example, the function evaluates properly when passed time followed by state, thereby satisfying the minimal requirements. The fact that it also evaluates correctly when passed only time simply indicates that the function does not require the state vector X_t. The important point to make is that it works when you pass it (t, X_t).

Furthermore, the `ts2func` function performs a zero-order-hold (ZOH) piecewise constant interpolation. The notion of piecewise constant parameters is pervasive throughout the SDE architecture, and is discussed in more detail in "Optimizing Accuracy: About Solution Precision and Error" on page 17-78.

4 **Perform a second simulation using the same initial random number state.** Complete the comparison by performing the second simulation using the same initial random number state:

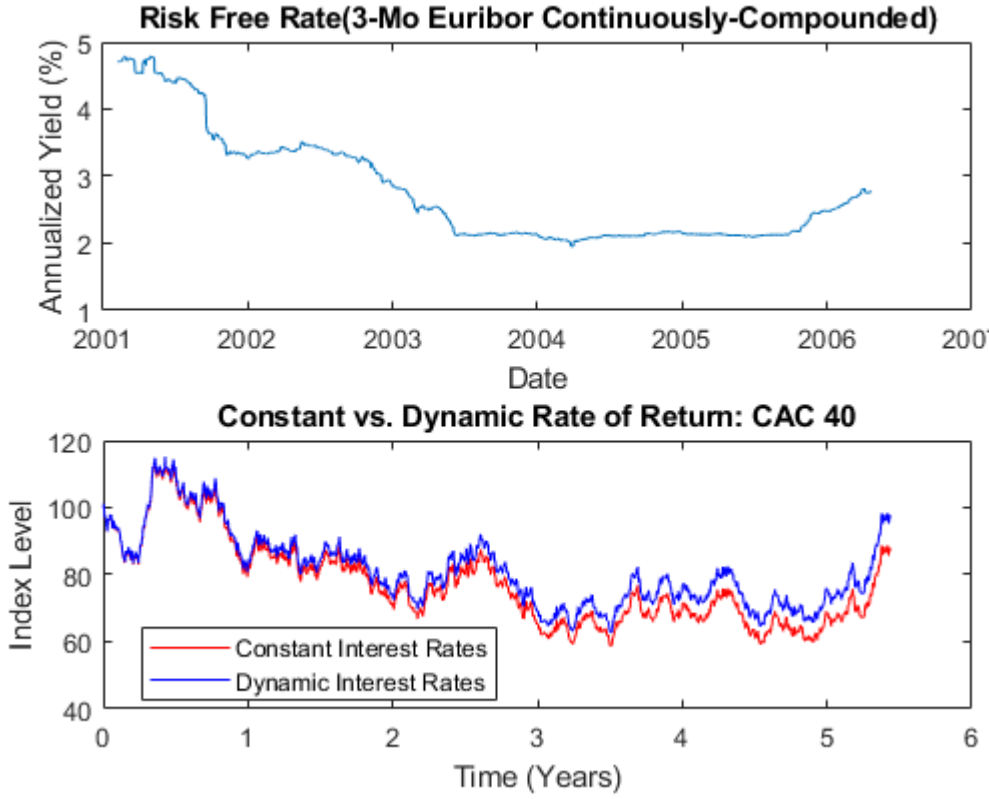
```
rng(5713,'twister')
obj = gbm(r, diag(sigma),'StartState',100)
```

```
obj =
   Class GBM: Generalized Geometric Brownian Motion
   -------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------------
      StartTime: 0
     StartState: 100
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: function ts2func/vector2Function
          Sigma: 0.231906
```

```
X2  = simulate(obj,nPeriods,'DeltaTime',dt);
```

5 **Compare the two simulation trials.** Plot the series of risk-free reference rates to compare the two simulation trials:

```
subplot(2,1,1)
plot(dates,100*yields)
datetick('x')
xlabel('Date')
ylabel('Annualized Yield (%)')
title('Risk Free Rate(3-Mo Euribor Continuously-Compounded)')
subplot(2,1,2)
plot(T,X1,'red',T,X2,'blue')
xlabel('Time (Years)')
ylabel('Index Level')
title('Constant vs. Dynamic Rate of Return: CAC 40')
legend({'Constant Interest Rates' 'Dynamic Interest Rates'},...
    'Location', 'Best')
```



The paths are close but not exact. The blue line in the last plot uses all the historical Euribor data, and illustrates a single trial of a historical simulation.

Pricing Equity Options

As discussed in "Ensuring Positive Interest Rates" on page 17-66, all simulation and interpolation methods allow you to specify one or more functions of the form:

$X_t = f(t, X_t)$

to evaluate at the end of every sample time.

The related example illustrates a simple, common end-of-period processing function to ensure nonnegative interest rates. This example illustrates a processing function that allows you to avoid simulation outputs altogether.

Consider pricing European stock options by Monte Carlo simulation within a Black-Scholes-Merton framework. Assume that the stock has the following characteristics:

- The stock currently trades at 100.
- The stock pays no dividends.
- The stock's volatility is 50% per annum.
- The option strike price is 95.
- The option expires in three months.
- The risk-free rate is constant at 10% per annum.

To solve this problem, model the evolution of the underlying stock by a univariate geometric Brownian motion (GBM) model with constant parameters:
$dX_t = 0.1X_t dt + 0.5X_t dW_t$

Furthermore, assume that the stock price is simulated daily, and that each calendar month comprises 21 trading days:

```
strike   = 95;
rate     = 0.1;
sigma    = 0.5;
dt       = 1/252;
nPeriods = 63;
T        = nPeriods*dt;
obj = gbm(rate,sigma,'StartState',100);
```

The goal is to simulate independent paths of daily stock prices, and calculate the price of European options as the risk-neutral sample average of the discounted terminal option payoff at expiration 63 days from now. This example calculates option prices by two approaches:

- A Monte Carlo simulation that explicitly requests the simulated stock paths as an output. The output paths are then used to price the options.
- An end-of-period processing function, accessible by time and state, that records the terminal stock price of each sample path. This processing function is implemented as a nested function with access to shared information. For more information, see `Example_BlackScholes.m`.

1 Before simulation, invoke the example file to access the end-of-period processing function:

```
nTrials = 10000; % Number of independent trials (i.e., paths)
f = Example_BlackScholes(nPeriods,nTrials)

f = struct with fields:
    BlackScholes: @Example_BlackScholes/saveTerminalStockPrice
       CallPrice: @Example_BlackScholes/getCallPrice
        PutPrice: @Example_BlackScholes/getPutPrice
```

2 Simulate 10000 independent trials (sample paths). Request the simulated stock price paths as an output, and specify an end-of-period processing function:

```
rng(88161,'twister')
X = simBySolution(obj,nPeriods,'DeltaTime',dt,...
    'nTrials',nTrials,'Processes',f.BlackScholes);
```

3 Calculate the option prices directly from the simulated stock price paths. Because these are European options, ignore all intermediate stock prices:

```
call = mean(exp(-rate*T)*max(squeeze(X(end,:,:)) - strike, 0))

call = 13.9342

put  = mean(exp(-rate*T)*max(strike - squeeze(X(end,:,:)), 0))

put = 6.4166
```

4 Price the options indirectly by invoking the nested functions:

```
f.CallPrice(strike,rate)

ans = 13.9342

f.PutPrice(strike,rate)

ans = 6.4166
```

For reference, the theoretical call and put prices computed from the Black-Scholes option formulas are `13.6953` and `6.3497`, respectively.

5 Although steps 3 on page 17-57 and 4 on page 17-57 produce the same option prices, the latter approach works directly with the terminal stock prices of each sample path. Therefore, it is much more memory efficient. In this example, there is no compelling reason to request an output.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation"
- "Improving Performance of Monte Carlo Simulation with Parallel Computing" on page 17-107
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

More About

- "SDEs" on page 17-2
- "SDE Models" on page 17-8
- "SDE Class Hierarchy" on page 17-5
- "Performance Considerations" on page 17-76

Simulating Interest Rates

In this section...

Simulating Interest Rates

All simulation methods require that you specify a time grid by specifying the number of periods (`NPERIODS`). You can also optionally specify a scalar or vector of strictly positive time increments (`DeltaTime`) and intermediate time steps (`NSTEPS`). These parameters, along with an initial sample time associated with the object (`StartTime`), uniquely determine the sequence of times at which the state vector is sampled. Thus, simulation methods allow you to traverse the time grid from beginning to end (that is, from left to right).

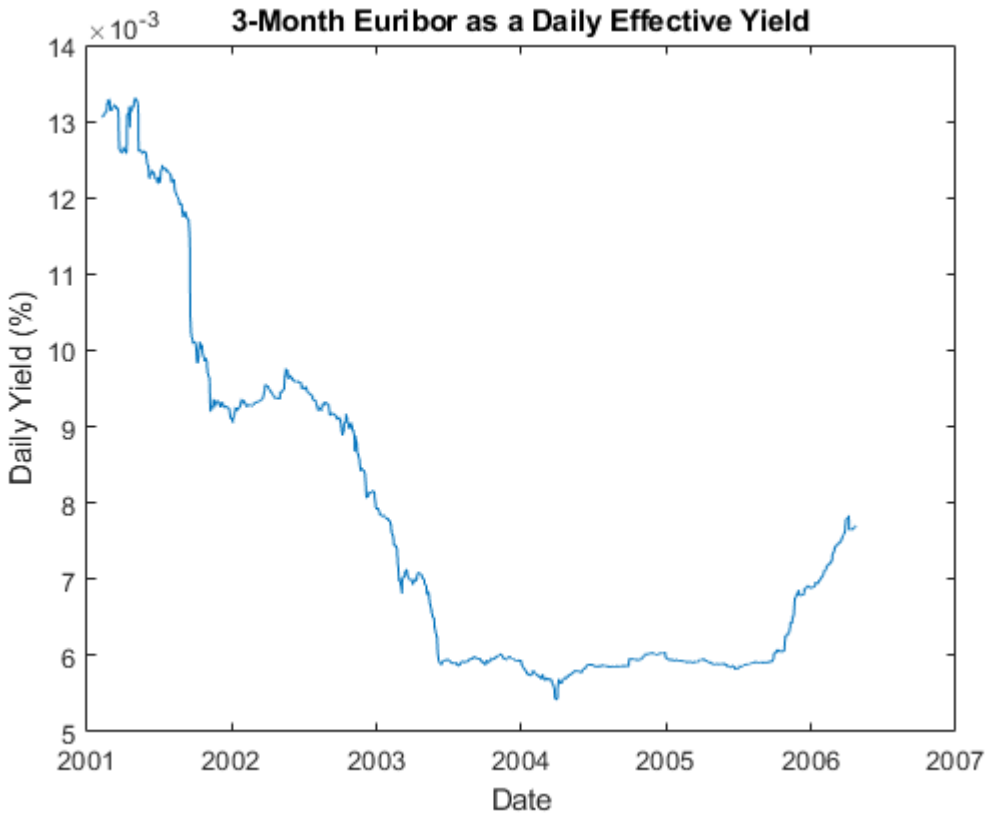
In contrast, interpolation methods allow you to traverse the time grid in any order, allowing both forward and backward movements in time. They allow you to specify a vector of interpolation times whose elements do not have to be unique.

Many references define the Brownian Bridge as a conditional simulation combined with a scheme for traversing the time grid, effectively merging two distinct algorithms. In contrast, the interpolation method offered here provides additional flexibility by intentionally separating the algorithms. In this method for moving about a time grid, you perform an initial Monte Carlo simulation to sample the state at the terminal time, and then successively sample intermediate states by stochastic interpolation. The first few samples determine the overall behavior of the paths, while later samples progressively refine the structure. Such algorithms are often called *variance reduction techniques*. This algorithm is simple when the number of interpolation times is a power of 2. In this case, each interpolation falls midway between two known states, refining the interpolation using a method like bisection. This example highlights the flexibility of refined interpolation by implementing this power-of-two algorithm.

1 **Load the data.** Load a historical data set of three-month Euribor rates, observed daily, and corresponding trading dates spanning the time interval from February 7, 2001 through April 24, 2006:

```
load Data_GlobalIdx2
plot(dates, 100 * Dataset.EB3M)
```

```
datetick('x'), xlabel('Date'), ylabel('Daily Yield (%)')
title('3-Month Euribor as a Daily Effective Yield')
```



2 **Fit a model to the data.** Now fit a simple univariate `Vasicek` model to the daily equivalent yields of the three-month Euribor data:

$$dX_t = S(L - X_t)dt + \sigma dW_t$$

Given initial conditions, the distribution of the short rate at some time T in the future is Gaussian with mean:

$$E(X_T) = X_0 e^{-ST} + L(1 - e^{-ST})$$

and variance:

$$Var(X_T) = \sigma^2(1 - e^{-ST}) / 2S$$

To calibrate this simple short rate model, rewrite it in more familiar regression format:

$$y_t = \alpha + \beta x_t + \varepsilon_t$$

where:

$$y_t = dX_t, \alpha = SLdt, \beta = -Sdt$$

perform an ordinary linear regression where the model volatility is proportional to the standard error of the residuals:

$$\sigma = \sqrt{Var(\varepsilon_t) / dt}$$

```
yields     = Dataset.EB3M;
regressors = [ones(length(yields) - 1, 1) yields(1:end-1)];
[coefficients, intervals, residuals] = ...
   regress(diff(yields), regressors);
dt    = 1;  % time increment = 1 day
speed = -coefficients(2)/dt;
level = -coefficients(1)/coefficients(2);
sigma =  std(residuals)/sqrt(dt);
```

3 **Create an object and set its initial StartState.** Create an `hwv` object using the constructor `hwv` with `StartState` set to the most recently observed short rate:

```
obj = hwv(speed, level, sigma, 'StartState', yields(end))
```

```
obj =
   Class HWV: Hull-White/Vasicek
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 7.70408e-05
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 4.77637e-07
          Level: 6.00424e-05
          Speed: 0.00228854
```

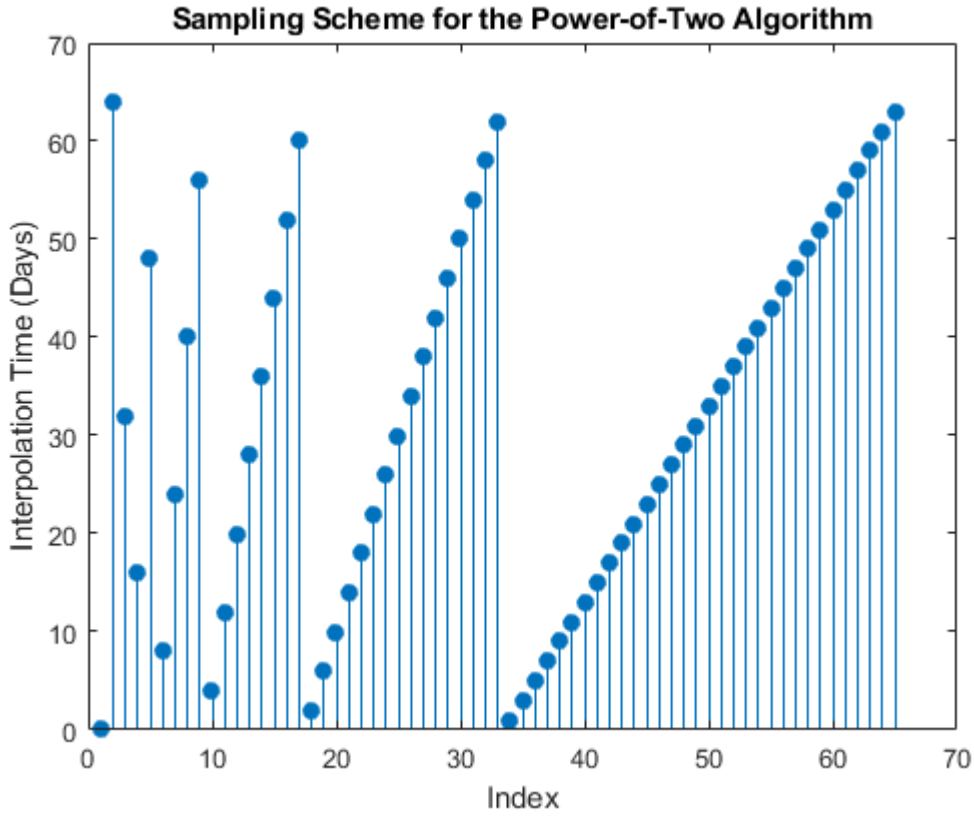
4 **Simulate the fitted model.** Assume, for example, that you simulate the fitted model over 64 (2^6) trading days, using a refined Brownian bridge with the power-of-two algorithm instead of the usual beginning-to-end Monte Carlo simulation approach. Furthermore, assume that the initial time and state coincide with those of the last available observation of the historical data, and that the terminal state is the expected value of the Vasicek model 64 days into the future. In this case, you can assess the behavior of various paths that all share the same initial and terminal states, perhaps to support pricing path-dependent interest rate options over a three-month interval.

Create a vector of interpolation times to traverse the time grid by moving both forward and backward in time. Specifically, the first interpolation time is set to the most recent short rate observation time, the second interpolation time is set to the terminal time, and subsequent interpolation times successively sample intermediate states:

```
T      = 64;
times  = (1:T)';
t      = NaN(length(times) + 1, 1);
t(1)   = obj.StartTime;
t(2)   = T;
delta  = T;
jMax   = 1;
iCount = 3;

for k = 1:log2(T)
    i = delta / 2;
    for j = 1:jMax
        t(iCount) = times(i);
        i         = i + delta;
        iCount    = iCount + 1;
    end
    jMax  = 2 * jMax;
    delta = delta / 2;
end
```

5 **Plot the interpolation times.** Examine the sequence of interpolation times generated by this algorithm:

```
stem(1:length(t), t, 'filled')
xlabel('Index'), ylabel('Interpolation Time (Days)')
title ('Sampling Scheme for the Power-of-Two Algorithm')
```



The first few samples are widely separated in time and determine the course structure of the paths. Later samples are closely spaced and progressively refine the detailed structure.

6 **Initialize the time series grid.** Now that you have generated the sequence of interpolation times, initialize a course time series grid to begin the interpolation. The sampling process begins at the last observed time and state taken from the historical short rate series, and ends 64 days into the future at the expected value of the `Vasicek` model derived from the calibrated parameters:

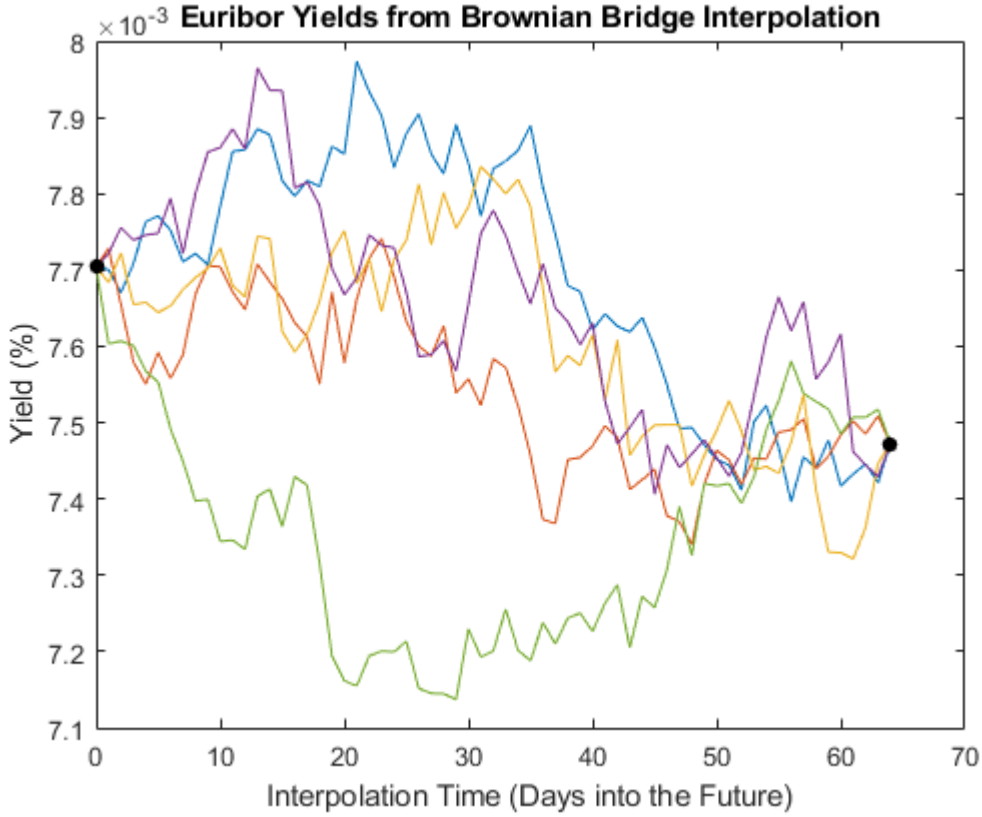
```
average = obj.StartState * exp(-speed * T) + level * ...
(1 - exp(-speed * T));
X       = [obj.StartState ; average];
```

7 **Generate five sample paths.** Generate five sample paths, setting the `Refine` input flag to `TRUE` to insert each new interpolated state into the time series grid as it becomes available. Perform interpolation on a trial-by-trial basis. Because the input time series `X` has five trials (where each page of the three-dimensional time series represents an independent trial), the interpolated output series `Y` also has five pages:

```
nTrials = 5;
rng(63349,'twister')
Y = obj.interpolate(t, X(:,:,ones(nTrials,1)), ...
'Times',[obj.StartTime  T], 'Refine', true);
```

8 **Plot the resulting sample paths.** Because the interpolation times do not monotonically increase, sort the times and reorder the corresponding short rates:

```
[t,i] = sort(t);
Y     = squeeze(Y);
Y     = Y(i,:);
plot(t, 100 * Y), hold('on')
plot(t([1 end]), 100 * Y([1 end],1),'. black','MarkerSize',20)
xlabel('Interpolation Time (Days into the Future)')
ylabel('Yield (%)'), hold('off')
title ('Euribor Yields from Brownian Bridge Interpolation')
```



The short rates in this plot represent alternative sample paths that share the same initial and terminal values. They illustrate a special, though simplistic, case of a broader sampling technique known as stratified sampling. For a more sophisticated example of stratified sampling, see “Stratified Sampling” on page 17-70.

Although this simple example simulated a univariate `Vasicek` interest rate model, it applies to problems of any dimensionality.

Tip Brownian-bridge methods also apply more general variance-reduction techniques. For more information, see “Stratified Sampling” on page 17-70.

Ensuring Positive Interest Rates

All simulation and interpolation methods allow you to specify a sequence of functions, or background processes, to evaluate at the end of every sample time period. This period includes any intermediate time steps determined by the optional `NSTEPS` input, as discussed in "Optimizing Accuracy: About Solution Precision and Error" on page 17-78. These functions are specified as callable functions of time and state, and must return an updated state vector X_t:

$$X_t = f(t, X_t)$$

You must specify multiple processing functions as a cell array of functions. These functions are invoked in the order in which they appear in the cell array.

Processing functions are not required to use time (t) or state (X_t). They are also not required to update or change the input state vector. In fact, simulation and interpolation methods have no knowledge of any implementation details, and in this respect, they only adhere to a published interface.

Such processing functions provide a powerful modeling tool that can solve various problems. Such functions allow you to, for example, specify boundary conditions, accumulate statistics, plot graphs, and price path-dependent options.

Except for Brownian motion (`BM`) models, the individual components of the simulated state vector typically represent variables whose real-world counterparts are inherently positive quantities, such as asset prices or interest rates. However, by default, most of the simulation and interpolation methods provided here model the transition between successive sample times as a scaled (possibly multivariate) Gaussian draw. So, when approximating a continuous-time process in discrete time, the state vector may not remain positive. The only exception is `simBySolution` for `gbm` objects and `simBySolution` for `hwv` objects, a logarithmic transform of separable geometric Brownian motion models. Moreover, by default, none of the simulation and interpolation methods make adjustments to the state vector. Therefore, you are responsible for ensuring that all components of the state vector remain positive as appropriate.

Fortunately, specifying nonnegative states ensures a simple end-of-period processing adjustment. Although this adjustment is widely applicable, it is revealing when applied to a univariate `cir` square-root diffusion model:

$$dX_t = 0.25(0.1 - X_t)dt + 0.2X_t^{\frac{1}{2}}dW_t = S(L - X_t)dt + \sigma X_t^{\frac{1}{2}}dW_t$$

Perhaps the primary appeal of univariate `cir` models where:

$$2SL \geq \sigma^2$$

is that the short rate remains positive. However, the positivity of short rates only holds for the underlying continuous-time model.

1 **Simulate daily short rates of the `cir` model.** To illustrate the latter statement, simulate daily short rates of the `cir` model, using the `cir` constructor, over one calendar year (approximately 250 trading days):

```
rng(14151617,'twister')
obj   = cir(0.25,@(t,X)0.1,0.2,'StartState',0.02);
[X,T] = simByEuler(obj,250,'DeltaTime',1/250,'nTrials',5);
sprintf('%0.4f\t%0.4f+i%0.4f\n',[T(195:205)';...
    real(X(195:205,1,4))'; imag(X(195:205,1,4))'])
```

```
ans =
    '0.7760    0.0003+i0.0000
     0.7800    0.0004+i0.0000
     0.7840    0.0002+i0.0000
     0.7880    -0.0000+i0.0000
     0.7920    0.0001+i0.0000
     0.7960    0.0002+i0.0000
     0.8000    0.0002+i0.0000
     0.8040    0.0008+i0.0001
     0.8080    0.0004+i0.0001
     0.8120    0.0008+i0.0001
     0.8160    0.0008+i0.0001
     '
```

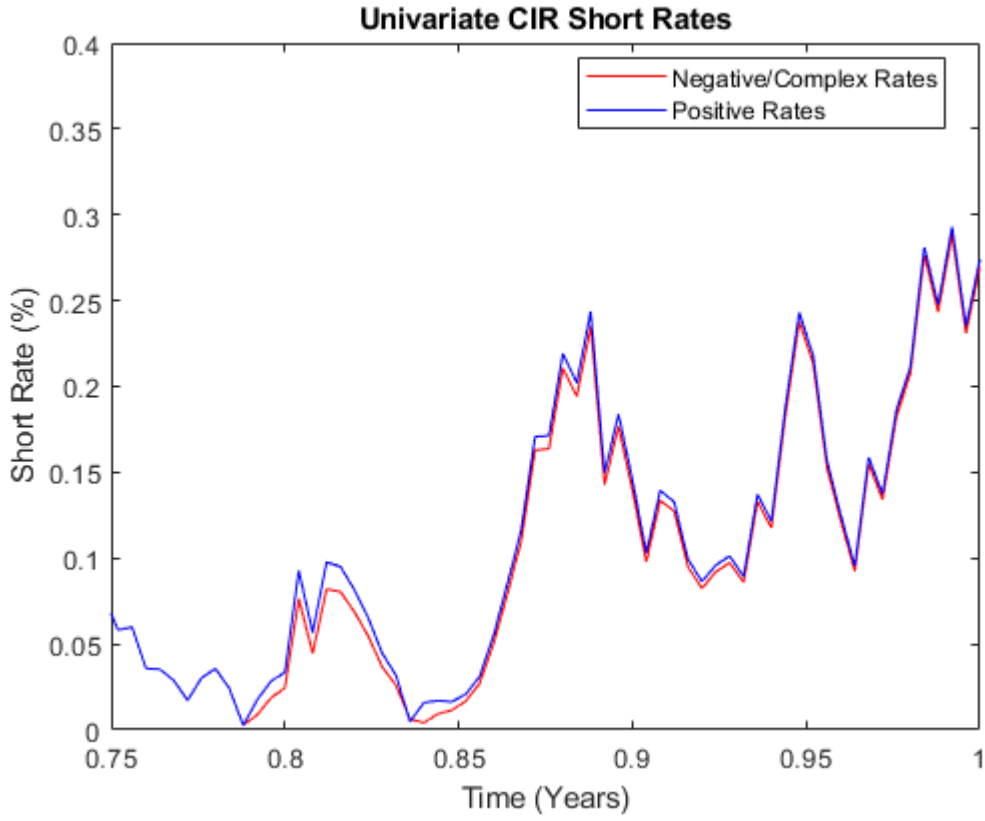
Interest rates can become negative if the resulting paths are simulated in discrete time. Moreover, since `cir` models incorporate a square root diffusion term, the short rates might even become complex.

2 **Repeat the simulation with a processing function.** Repeat the simulation, this time specifying a processing function that takes the absolute magnitude of the short rate at the end of each period. You can access the processing function by time and state (t, X_t), but it only uses the state vector of short rates X_t:

```
rng(14151617,'twister')
[Y,T] = simByEuler(obj,250,'DeltaTime',1/250,...
    'nTrials',5,'Processes',@(t,X)abs(X));
```

3 **Compare the adjusted and non-adjusted paths.** Graphically compare the magnitude of the unadjusted path (with negative and complex numbers!) to the corresponding path kept positive by using an end-of-period processing function over the time span of interest:

```
clf
plot(T,100*abs(X(:,1,4)),'red',T,100*Y(:,1,4),'blue')
axis([0.75 1 0 0.4])
xlabel('Time (Years)'), ylabel('Short Rate (%)')
title('Univariate CIR Short Rates')
legend({'Negative/Complex Rates' 'Positive Rates'}, ...
    'Location', 'Best')
```



Tip You can use this method to obtain more accurate SDE solutions. For more information, see “Performance Considerations” on page 17-76.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Simulating Equity Prices” on page 17-34
- “Stratified Sampling” on page 17-70
- “Pricing American Basket Options by Monte Carlo Simulation”
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 17-107
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5
- “Performance Considerations” on page 17-76

Stratified Sampling

Simulation methods allow you to specify a noise process directly, as a callable function of time and state:

$$z_t = Z(t, X_t)$$

Stratified sampling is a variance reduction technique that constrains a proportion of sample paths to specific subsets (or *strata*) of the sample space.

This example specifies a noise function to stratify the terminal value of a univariate equity price series. Starting from known initial conditions, the function first stratifies the terminal value of a standard Brownian motion, and then samples the process from beginning to end by drawing conditional Gaussian samples using a Brownian bridge.

The stratification process assumes that each path is associated with a single stratified terminal value such that the number of paths is equal to the number of strata. This technique is called *proportional sampling*. This example is similar to, yet more sophisticated than, the one discussed in "Simulating Interest Rates" on page 17-59. Since stratified sampling requires knowledge of the future, it also requires more sophisticated time synchronization; specifically, the function in this example requires knowledge of the entire sequence of sample times. For more information, see the example `stratifiedExample.m`.

The function implements proportional sampling by partitioning the unit interval into bins of equal probability by first drawing a random number uniformly distributed in each bin. The inverse cumulative distribution function of a standard *N(0,1)* Gaussian distribution then transforms these stratified uniform draws. Finally, the resulting stratified Gaussian draws are scaled by the square root of the terminal time to stratify the terminal value of the Brownian motion.

The noise function does not return the actual Brownian paths, but rather the Gaussian draws $Z(t,X_t)$ that drive it.

This example first stratifies the terminal value of a univariate, zero-drift, unit-variance-rate Brownian motion (`bm`) model:

$$dX_t = dW_t$$

1 Assume that 10 paths of the process are simulated daily over a three-month period. Also assume that each calendar month and year consist of 21 and 252 trading days, respectively:

```
rng(10203,'twister')
dt       = 1 / 252;         % 1 day = 1/252 years
nPeriods = 63;              % 3 months = 63 trading days
T        = nPeriods * dt;   % 3 months = 0.25 years
nPaths   = 10;             % # of simulated paths
obj      = bm(0, 1, 'StartState', 0);
sampleTimes = cumsum([obj.StartTime; ...
    dt(ones(nPeriods,1))]);
z        = Example_StratifiedRNG(nPaths, sampleTimes);
```

2 Simulate the standard Brownian paths by explicitly passing the stratified sampling function to the simulation method:

```
X = obj.simulate(nPeriods, 'DeltaTime', dt, ...
    'nTrials', nPaths, 'Z', z);
```

3 For convenience, reorder the output sample paths by reordering the three-dimensional output to a 2-dimensional equivalent array:

```
X = squeeze(X);
```

4 Verify the stratification:

a Recreate the uniform draws with proportional sampling:

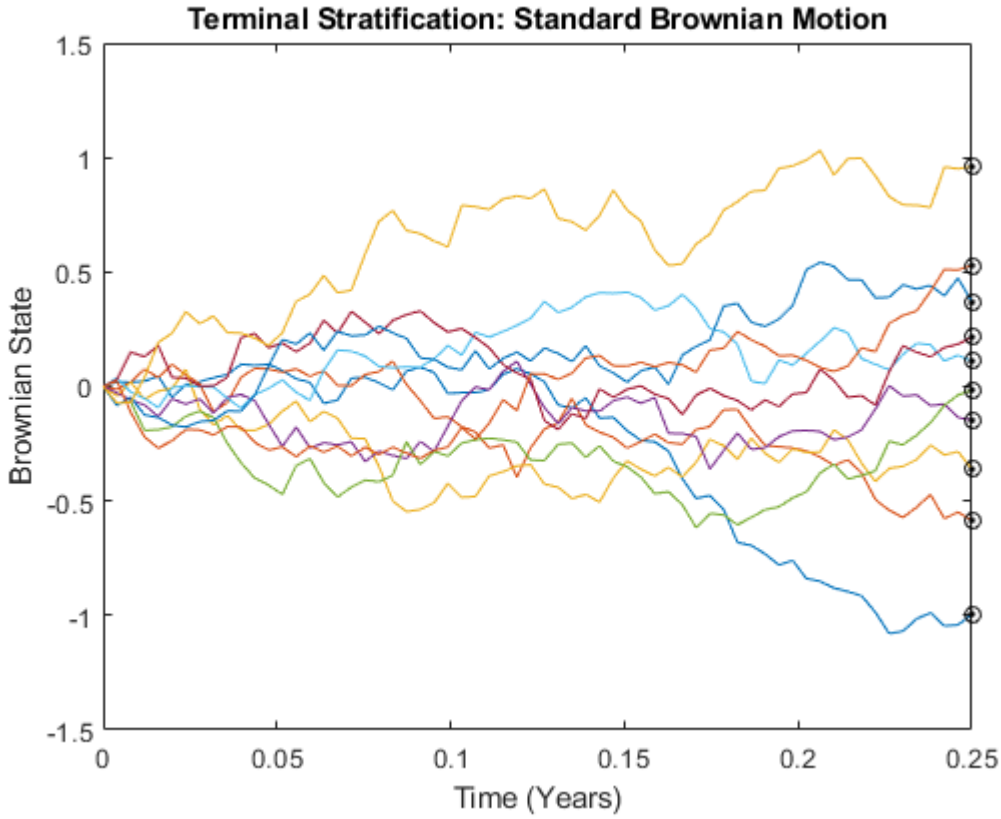
```
rng(10203,'twister')
U  = ((1:nPaths)' - 1 + rand(nPaths,1))/nPaths;
```

b Transform them to obtain the terminal values of standard Brownian motion:

```
WT = norminv(U) * sqrt(T);  % Stratified Brownian motion.
```

c Plot the terminal values and output paths on the same figure:

```
plot(sampleTimes, X), hold('on')
xlabel('Time (Years)'), ylabel('Brownian State')
title('Terminal Stratification: Standard Brownian Motion')
plot(T, WT, '. black', T, WT, 'o black')
hold('off')
```



The last value of each sample path (the last row of the output array `X`) coincides with the corresponding element of the stratified terminal value of the Brownian motion. This occurs because the simulated model and the noise generation function both represent the same standard Brownian motion.

However, you can use the same stratified sampling function to stratify the terminal price of constant-parameter geometric Brownian motion models. In fact, you can use the stratified sampling function to stratify the terminal value of any constant-parameter model driven by Brownian motion if the model's terminal value is a monotonic transformation of the terminal value of the Brownian motion.

To illustrate this, load the data set and simulate risk-neutral sample paths of the FTSE 100 index using a geometric Brownian motion (`GBM`) model with constant parameters:

$$dX_t = rX_t dt + \sigma X_t dW_t$$

where the average Euribor yield represents the risk-free rate of return.

1 Assume that the relevant information derived from the daily data is annualized, and that each calendar year comprises 252 trading days:

```
load Data_GlobalIdx2
returns = tick2ret(Dataset.FTSE);
sigma   = std(returns) * sqrt(252);
rate    = Dataset.EB3M;
rate    = mean(360 * log(1 + rate));
```

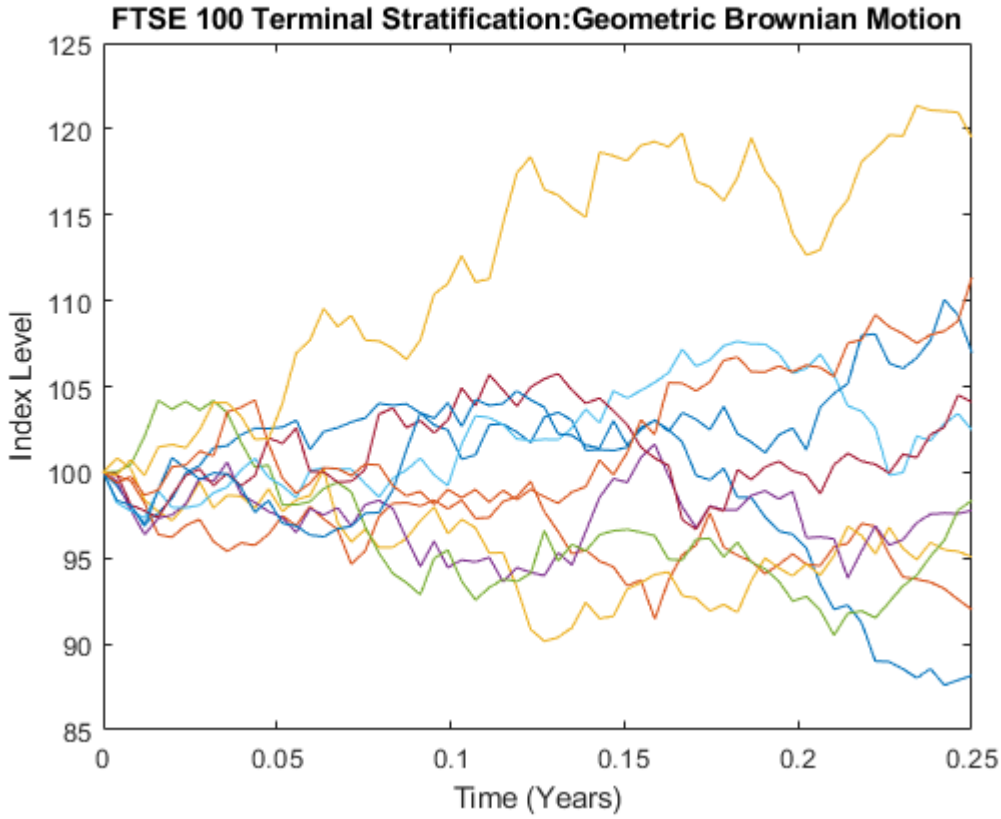
2 Create the `GBM` model using the `gbm` constructor, assuming the FTSE 100 starts at 100:

```
obj = gbm(rate, sigma, 'StartState', 100);
```

3 Determine the sample time and simulate the price paths.

In what follows, `NSTEPS` specifies the number of intermediate time steps within each time increment `DeltaTime`. Each increment `DeltaTime` is partitioned into `NSTEPS` subintervals of length `DeltaTime/nSteps` each, refining the simulation by evaluating the simulated state vector at `NSTEPS`–1 intermediate points. This refinement improves accuracy by allowing the simulation to more closely approximate the underlying continuous-time process without storing the intermediate information:

```
nSteps      = 1;
sampleTimes = cumsum([obj.StartTime ; ...
dt(ones(nPeriods * nSteps,1))/nSteps]);
z           = Example_StratifiedRNG(nPaths, sampleTimes);
rng(10203,'twister')
[Y, Times]  = obj.simBySolution(nPeriods, 'nTrials', nPaths,...
'DeltaTime', dt, 'nSteps', nSteps,  'Z', z);
Y = squeeze(Y);   % Reorder to a 2-D array
plot(Times, Y)
xlabel('Time (Years)'), ylabel('Index Level')
title('FTSE 100 Terminal Stratification:Geometric Brownian Motion')
```



Although the terminal value of the Brownian motion shown in the latter plot is normally distributed, and the terminal price in the previous plot is lognormally distributed, the corresponding paths of each graph are similar.

Tip For another example of variance reduction techniques, see "Simulating Interest Rates" on page 17-59.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Pricing American Basket Options by Monte Carlo Simulation”
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 17-107
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5
- “Performance Considerations” on page 17-76

Performance Considerations

In this section...

Managing Memory

There are two general approaches for managing memory when solving most problems supported by the SDE engine:

- “Managing Memory with Outputs” on page 17-76
- “Managing Memory Using End-of-Period Processing Functions” on page 17-77

Managing Memory with Outputs

Perform a traditional simulation to simulate the underlying variables of interest, specifically requesting and then manipulating the output arrays.

This approach is straightforward and the best choice for small or medium-sized problems. Since its outputs are arrays, it is convenient to manipulate simulated results in the MATLAB matrix-based language. However, as the scale of the problem increases, the benefit of this approach decreases, because the output arrays must store large quantities of possibly extraneous information.

For example, consider pricing a European option in which the terminal price of the underlying asset is the only value of interest. To ease the memory burden of the traditional approach, reduce the number of simulated periods specified by the required input `NPERIODS` and specify the optional input `NSTEPS`. This enables you to manage memory without sacrificing accuracy (see “Optimizing Accuracy: About Solution Precision and Error” on page 17-78).

In addition, simulation methods can determine the number of output arguments and allocate memory accordingly. Specifically, all simulation methods support the same output argument list:

```
[Paths,Times,Z]
```

where `Paths` and `Z` can be large, three-dimensional time series arrays. However, the underlying noise array is typically unnecessary, and is only stored if requested as an output. In other words, `Z` is stored only at your request; do not request it if you do not need it.

If you need the output noise array `Z`, but do not need the `Paths` time series array, then you can avoid storing `Paths` two ways:

- It is best practice to use the ~ output argument placeholder. For example, use the following output argument list to store `Z` and `Times`, but not `Paths`:

  ```
  [~,Times,Z]
  ```

- Use the optional input flag `StorePaths`, which all simulation methods support. By default, `Paths` is stored (`StorePaths` = `true`). However, setting `StorePaths` to `false` returns `Paths` as an empty matrix.

Managing Memory Using End-of-Period Processing Functions

Specify one or more end-of-period processing functions to manage and store only the information of interest, avoiding simulation outputs altogether.

This approach requires you to specify one or more end-of-period processing functions, and is often the preferred approach for large-scale problems. This approach allows you to avoid simulation outputs altogether. Since no outputs are requested, the three-dimensional time series arrays `Paths` and `Z` are not stored.

This approach often requires more effort, but is far more elegant and allows you to customize tasks and dramatically reduce memory usage. See “Pricing Equity Options” on page 17-55.

Enhancing Performance

The following approaches improve performance when solving SDE problems:

- **Specifying model parameters as traditional MATLAB arrays and functions, in various combinations.** This provides a flexible interface that can support virtually any general nonlinear relationship. However, while functions offer a convenient and elegant solution for many problems, simulations typically run faster when you specify parameters as double-precision vectors or matrices. Thus, it is a good practice to specify model parameters as arrays when possible.

- **Use models that have overloaded Euler simulation methods, when possible.** Using Brownian motion (`BM`) and geometric Brownian motion (`GBM`) models that provide overloaded Euler simulation methods take advantage of separable, constant-parameter models. These specialized methods are exceptionally fast, but are only available to models with constant parameters that are simulated without specifying end-of-period processing and noise generation functions.
- **Replace the simulation of a constant-parameter, univariate model derived from the `SDEDDO` class with that of a diagonal multivariate model**. Treat the multivariate model as a portfolio of univariate models. This increases the dimensionality of the model and enhances performance by decreasing the effective number of simulation trials.

  Note This technique is applicable only to constant-parameter univariate models without specifying end-of-period processing and noise generation functions.

- **Take advantage of the fact that simulation methods are designed to detect the presence of `NaN` (not a number) conditions returned from end-of-period processing functions.** A `NaN` represents the result of an undefined numerical calculation, and any subsequent calculation based on a `NaN` produces another `NaN`. This helps improve performance in certain situations. For example, consider simulating paths of the underlier of a knock-out barrier option (that is, an option that becomes worthless when the price of the underlying asset crosses some prescribed barrier). Your end-of-period function could detect a barrier crossing and return a `NaN` to signal early termination of the current trial.

Optimizing Accuracy: About Solution Precision and Error

The simulation architecture does not, in general, simulate *exact* solutions to any SDE. Instead, the simulation architecture provides a discrete-time approximation of the underlying continuous-time process, a simulation technique often known as a *Euler approximation*.

In the most general case, a given simulation derives directly from an SDE. Therefore, the simulated discrete-time process approaches the underlying continuous-time process only in the limit as the time increment *dt* approaches zero. In other words, the simulation architecture places more importance on ensuring that the probability distributions of the discrete-time and continuous-time processes are close, than on the pathwise proximity of the processes.

Before illustrating techniques to improve the approximation of solutions, it is helpful to understand the source of error. Throughout this architecture, all simulation methods assume that model parameters are piecewise constant over any time interval of length *dt*. In fact, the methods even evaluate dynamic parameters at the beginning of each time interval and hold them fixed for the duration of the interval. This sampling approach introduces *discretization error*.

However, there are certain models for which the piecewise constant approach provides exact solutions:

- "Creating Brownian Motion (BM) Models" on page 17-25 with constant parameters, simulated by Euler approximation (`simByEuler`).
- "Creating Geometric Brownian Motion (GBM) Models" on page 17-27 with constant parameters, simulated by closed-form solution (`simBySolution`).
- "Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 17-30 with constant parameters, simulated by closed-form solution (`simBySolution`)

More generally, you can simulate the exact solutions for these models even if the parameters vary with time, if they vary in a piecewise constant way such that parameter changes coincide with the specified sampling times. However, such exact coincidence is unlikely; therefore, the previously discussed constant parameter condition is commonly used in practice.

One obvious way to improve accuracy involves sampling the discrete-time process more frequently. This decreases the time increment (*dt*), causing the sampled process to more closely approximate the underlying continuous-time process. Although decreasing the time increment is universally applicable, however, there is a tradeoff among accuracy, run-time performance, and memory usage.

To manage this tradeoff, specify an optional input argument, `NSTEPS`, for all simulation methods. `NSTEPS` indicates the number of intermediate time steps within each time increment *dt*, at which the process is sampled but not reported.

It is important and convenient at this point to emphasize the relationship of the inputs `NSTEPS`, `NPERIODS`, and `DeltaTime` to the output vector `Times`, which represents the actual observation times at which the simulated paths are reported.

- `NPERIODS`, a required input, indicates the number of simulation periods of length `DeltaTime`, and determines the number of rows in the simulated three-dimensional `Paths` time series array (if an output is requested).

- `DeltaTime` is optional, and indicates the corresponding `NPERIODS`-length vector of positive time increments between successive samples. It represents the familiar *dt* found in stochastic differential equations. If `DeltaTime` is unspecified, the default value of 1 is used.
- `NSTEPS` is also optional, and is only loosely related to `NPERIODS` and `DeltaTime`. `NSTEPS` specifies the number of intermediate time steps within each time increment `DeltaTime`.

  Specifically, each time increment `DeltaTime` is partitioned into `NSTEPS` subintervals of length `DeltaTime/NSTEPS` each, and refines the simulation by evaluating the simulated state vector at `(NSTEPS - 1)` intermediate times. Although the output state vector (if requested) is not reported at these intermediate times, this refinement improves accuracy by causing the simulation to more closely approximate the underlying continuous-time process. If `NSTEPS` is unspecified, the default is `1` (to indicate no intermediate evaluation).
- The output `Times` is an `NPERIODS + 1`-length column vector of observation times associated with the simulated paths. Each element of `Times` is associated with a corresponding row of `Paths`.

The following example illustrates this intermediate sampling by comparing the difference between a closed-form solution and a sequence of Euler approximations derived from various values of `NSTEPS`.

Example: Improving Solution Accuracy

Consider a univariate geometric Brownian motion (`GBM`) model using the `gbm` constructor with constant parameters:
$dX_t = 0.1X_t dt + 0.4X_t dW_t$.

Assume that the expected rate of return and volatility parameters are annualized, and that a calendar year comprises 250 trading days.

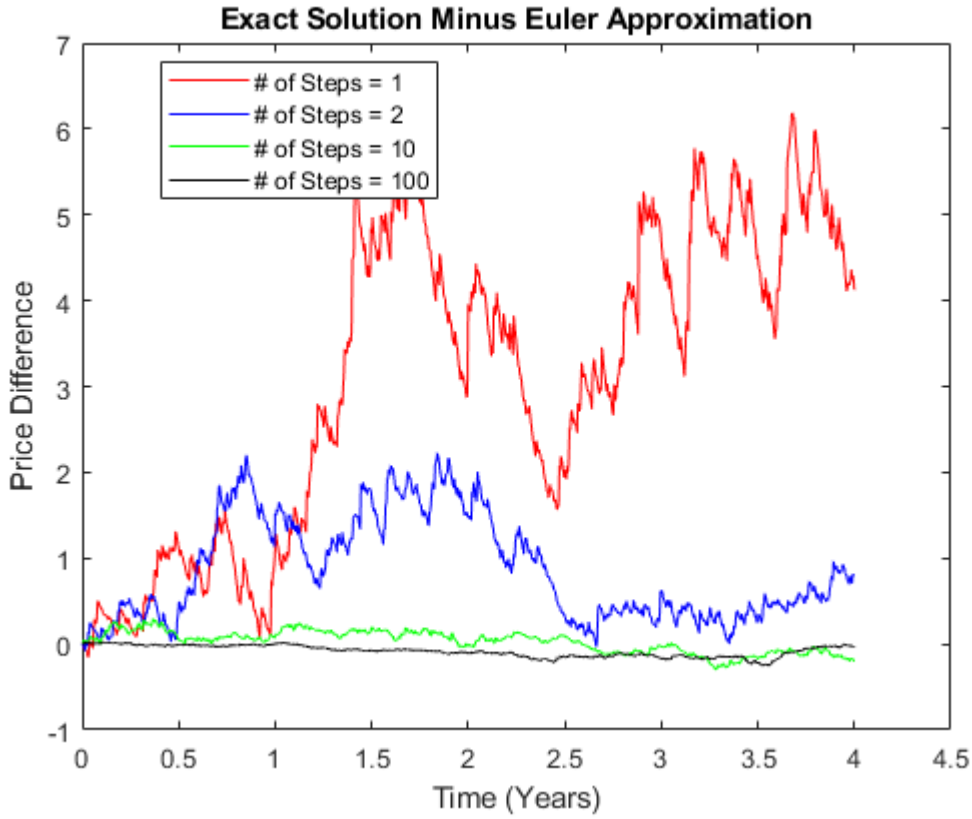
1 Simulate approximately four years of univariate prices for both the exact solution and the Euler approximation for various values of `NSTEPS`:

```
nPeriods = 1000;
dt       = 1/250;
obj      = gbm(0.1,0.4,'StartState',100);
rng(575,'twister')
[X1,T1]    = simBySolution(obj,nPeriods,'DeltaTime',dt);
rng(575,'twister')
```

```
[Y1,T1]    = simByEuler(obj,nPeriods,'DeltaTime',dt);
rng(575,'twister')
[X2,T2]    = simBySolution(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',2);
rng(575,'twister')
[Y2,T2]    = simByEuler(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',2);
rng(575,'twister')
[X3,T3]    = simBySolution(obj,nPeriods, 'DeltaTime',...
    dt,'nSteps',10);
rng(575,'twister')
[Y3,T3]    = simByEuler(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',10);
rng(575,'twister')
[X4,T4]    = simBySolution(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',100);
rng(575,'twister')
[Y4,T4]    = simByEuler(obj,nPeriods,'DeltaTime',...
    dt,'nSteps',100);
```

2 Compare the error (the difference between the exact solution and the Euler approximation) graphically:

```
clf;
plot(T1,X1 - Y1,'red')
hold on;
plot(T2,X2 - Y2,'blue')
plot(T3,X3 - Y3,'green')
plot(T4,X4 - Y4,'black')
hold off
xlabel('Time (Years)')
ylabel('Price Difference')
title('Exact Solution Minus Euler Approximation')
legend({'# of Steps = 1'  '# of Steps = 2' ...
    '# of Steps = 10' '# of Steps = 100'},...
    'Location', 'Best')
hold off
```




```
whos T X Y
```

As expected, the simulation error decreases as the number of intermediate time steps increases. Because the intermediate states are not reported, all simulated time series have the same number of observations regardless of the actual value of `NSTEPS`.

Furthermore, since the previously simulated exact solutions are correct for any number of intermediate time steps, additional computations are not needed for this example. In fact, this assessment is generally correct. The exact solutions are sampled at intermediate times to ensure that the simulation uses the same sequence of Gaussian random variates in the same order. Without this assurance, there is no way to compare simulated prices on a pathwise basis. However, there might be valid reasons for

sampling exact solutions at closely spaced intervals, such as pricing path-dependent options.

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Pricing American Basket Options by Monte Carlo Simulation”
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 17-107
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5

Pricing American Basket Options by Monte Carlo Simulation

This example shows how to model the fat-tailed behavior of asset returns and assess the impact of alternative joint distributions on basket option prices. Using various implementations of a separable multivariate Geometric Brownian Motion (GBM) process, often referred to as a *multi-dimensional market model*, the example simulates risk-neutral sample paths of an equity index portfolio and prices basket put options using the technique of Longstaff & Schwartz.

In addition, this example also illustrates salient features of the Stochastic Differential Equation (SDE) architecture, including

- Customized random number generation functions that compare Brownian motion and Brownian copulas
- End-of-period processing functions that form an equity index basket and price American options on the underlying basket based on the least squares method of Longstaff & Schwartz
- Piecewise probability distributions and Extreme Value Theory (EVT)

This example also highlights important issues of volatility and interest rate scaling. It illustrates how equivalent results can be achieved by working with daily or annualized data. For more information about EVT and copulas, see "Using Extreme Value Theory and Copulas to Evaluate Market Risk" (Econometrics Toolbox).

Overview of the Modeling Framework

The ultimate objective of this example is to compare basket option prices derived from different noise processes. The first noise process is a traditional Brownian motion model whose index portfolio price process is driven by correlated Gaussian random draws. As an alternative, the Brownian motion benchmark is compared to noise processes driven by Gaussian and Student's t copulas, referred to collectively as a *Brownian copula.*

A copula is a multivariate cumulative distribution function (CDF) with uniformly-distributed margins. Although the theoretical foundations were established decades ago, copulas have experienced a tremendous surge in popularity over the last few years, primarily as a technique for modeling non-Gaussian portfolio risks.

Although numerous families exist, all copulas represent a statistical device for modeling the dependence structure between two or more random variables. In addition, important statistics, such as *rank correlation* and *tail dependence* are properties of a given copula and are unchanged by monotonic transforms of their margins.

These copula draws produce dependent random variables, which are subsequently transformed to individual variables (margins). This transformation is achieved with a semi-parametric probability distribution with generalized Pareto tails.

The risk-neutral market model to simulate is
$dX_t = rX_t dt + \sigma X_t dW_t$

where the risk-free rate, `r`, is assumed constant over the life of the option. Because this is a separable multivariate model, the risk-free return is a diagonal matrix in which the same riskless return is applied to all indices. Dividend yields are ignored to simplify the model and its associated data collection.

In contrast, the specification of the exposure matrix, `sigma`, depends on how the driving source of uncertainty is modeled. You can model it directly as a Brownian motion (correlated Gaussian random numbers implicitly mapped to Gaussian margins) or model it as a Brownian copula (correlated Gaussian or *t* random numbers explicitly mapped to semi-parametric margins).

Because the CDF and inverse CDF (quantile function) of univariate distributions are both monotonic transforms, a copula provides a convenient way to simulate dependent random variables whose margins are dissimilar and arbitrarily distributed. Moreover, because a copula defines a given dependence structure regardless of its margins, copula parameter calibration is typically easier than estimation of the joint distribution function.

Once you have simulated sample paths, options are priced by the least squares regression method of Longstaff & Schwartz (see *Valuing American Options by Simulation: A Simple Least-Squares Approach*, The Review of Financial Studies, Spring 2001). This approach uses least squares to estimate the expected payoff of an option if it is not immediately exercised. It does so by regressing the discounted option cash flows received in the future on the current price of the underlier associated with all in-the-money sample paths. The continuation function is estimated by a simple third-order polynomial, in which all cash flows and prices in the regression are normalized by the option strike price, improving numerical stability.

Import the Supporting Historical Dataset

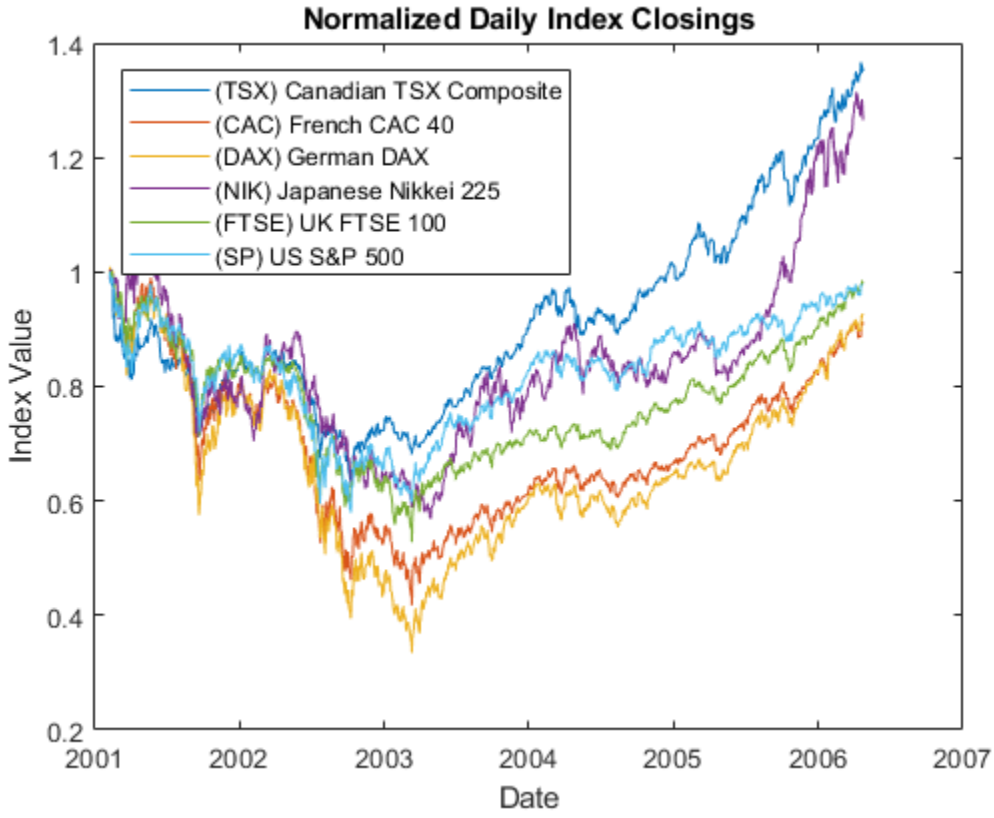
Load a daily historical dataset of 3-month Euribor, the trading dates spanning the interval 07-Feb-2001 to 24-Apr-2006, and the closing index levels of the following representative large-cap equity indices:

- TSX Composite (Canada)
- CAC 40 (France)
- DAX (Germany)
- Nikkei 225 (Japan)
- FTSE 100 (UK)
- S&P 500 (US)

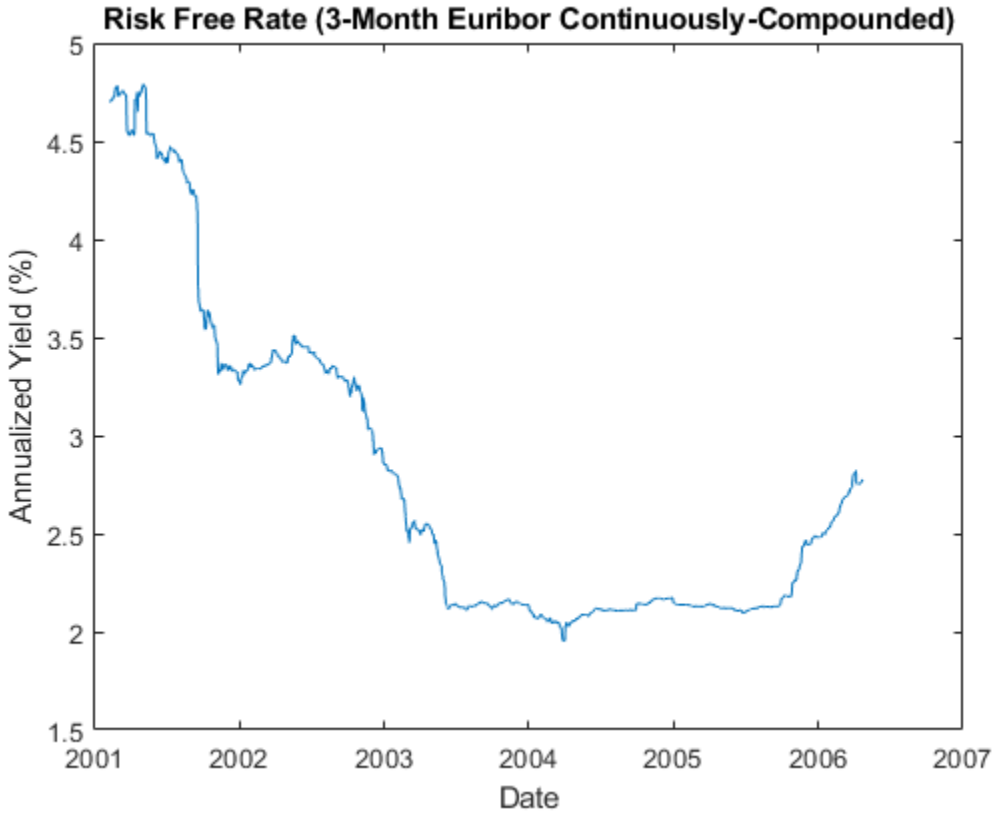
```
clear
load Data_GlobalIdx2
```

The following plots illustrate this data. Specifically, the plots show the relative price movements of each index and the Euribor risk-free rate proxy. The initial level of each index has been normalized to unity to facilitate the comparison of relative performance over the historical record.

```
nIndices  = size(Data,2)-1;     % # of indices

prices = Data(:,1:end-1);

yields = Data(:,end);             % daily effective yields
yields = 360 * log(1 + yields);   % continuously-compounded, annualized yield

plot(dates, ret2tick(tick2ret(prices,[],'continuous'),[],[],[],'continuous'))
datetick('x')

xlabel('Date')
ylabel('Index Value')
title ('Normalized Daily Index Closings')
legend(series{1:end-1}, 'Location', 'NorthWest')
```




```
plot(dates, 100 * yields)
datetick('x')
xlabel('Date')
ylabel('Annualized Yield (%)')
title('Risk Free Rate (3-Month Euribor Continuously-Compounded)')
```



Extreme Value Theory & Piecewise Probability Distributions

To prepare for copula modeling, characterize individually the distribution of returns of each index. Although the distribution of each return series may be characterized parametrically, it is useful to fit a semi-parametric model using a piecewise distribution with generalized Pareto tails. This uses Extreme Value Theory to better characterize the behavior in each tail.

The Statistics and Machine Learning Toolbox™ software currently supports two univariate probability distributions related to EVT, a statistical tool for modeling the fat-tailed behavior of financial data such as asset returns and insurance losses:

- Generalized Extreme Value (GEV) distribution, which uses a modeling technique known as the *block maxima or minima* method. This approach, divides a historical dataset into a set of sub-intervals, or blocks, and the largest or smallest observation in each block is recorded and fitted to a GEV distribution.
- Generalized Pareto (GP) distribution, uses a modeling technique known as the *distribution of exceedances* or *peaks over threshold* method. This approach sorts a historical dataset and fits the amount by which those observations that exceed a specified threshold to a GP distribution.

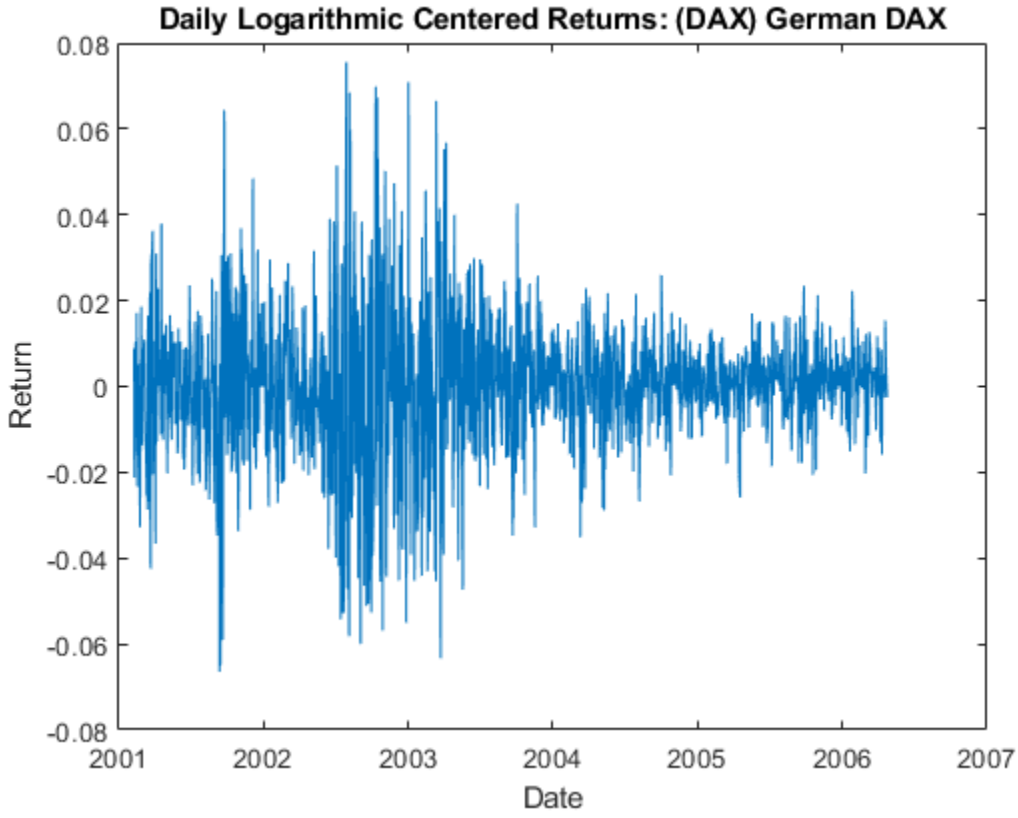
The following analysis highlights the Pareto distribution, which is more widely used in risk management applications.

Suppose we want to create a complete statistical description of the probability distribution of daily asset returns of any one of the equity indices. Assume that this description is provided by a piecewise semi-parametric distribution, where the asymptotic behavior in each tail is characterized by a generalized Pareto distribution.

Ultimately, a copula will be used to generate random numbers to drive the simulations. The CDF and inverse CDF transforms will capture the volatility of simulated returns as part of the diffusion term of the SDE. The mean return of each index is governed by the riskless rate and incorporated in the drift term of the SDE. The following code segment centers the returns (that is, extracts the mean) of each index.

Because the following analysis uses extreme value theory to characterize the distribution of each equity index return series, it is helpful to examine details for a particular country:

```
returns = tick2ret(prices,[],'continuous');        % convert prices to returns
returns = bsxfun(@minus, returns, mean(returns));  % center the returns
index   = 3;                                       % Germany stored in column 3

plot(dates(2:end), returns(:,index))
datetick('x')
xlabel('Date')
ylabel('Return')
title(['Daily Logarithmic Centered Returns: ' series{index}])
```



Note that this code segment can be changed to examine details for any country.

Using these centered returns, estimate the empirical, or non-parametric, CDF of each index with a Gaussian kernel. This smoothes the CDF estimates, eliminating the staircase pattern of unsmoothed sample CDFs. Although non-parametric kernel CDF estimates are well-suited for the interior of the distribution, where most of the data is found, they tend to perform poorly when applied to the upper and lower tails. To better estimate the tails of the distribution, apply EVT to the returns that fall in each tail.

Specifically, find the upper and lower thresholds such that 10% of the returns are reserved for each tail. Then fit the amount by which the extreme returns in each tail fall beyond the associated threshold to a Pareto distribution by maximum likelihood.

The following code segment creates one object of type `paretotails` for each index return series. These Pareto tail objects encapsulate the estimates of the parametric Pareto lower tail, the non-parametric kernel-smoothed interior, and the parametric Pareto upper tail to construct a composite semi-parametric CDF for each index.

```
tailFraction = 0.1;               % decimal fraction allocated to each tail
tails = cell(nIndices,1);  % cell array of Pareto tail objects

for i = 1:nIndices
    tails{i} = paretotails(returns(:,i), tailFraction, 1 - tailFraction, 'kernel');
end
```

The resulting piecewise distribution object allows interpolation within the interior of the CDF and extrapolation (function evaluation) in each tail. Extrapolation allows estimation of quantiles outside the historical record, which is invaluable for risk management applications.

Pareto tail objects also provide methods to evaluate the CDF and inverse CDF (quantile function), and to query the cumulative probabilities and quantiles of the boundaries between each segment of the piecewise distribution.

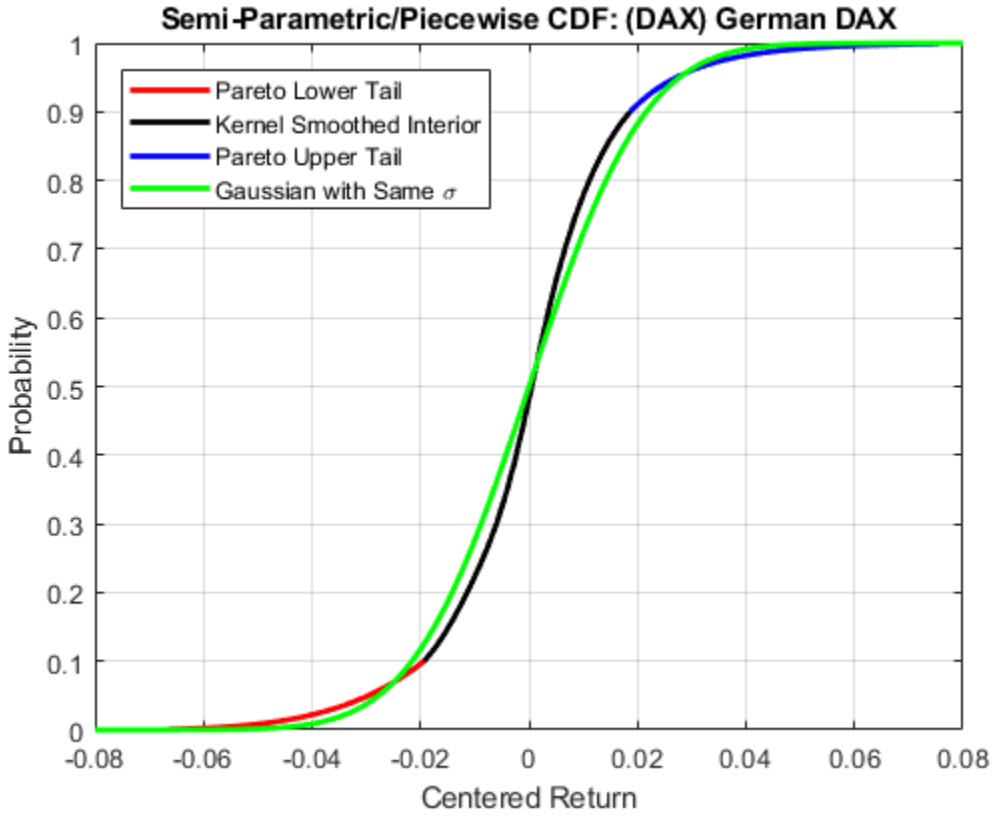
Now that three distinct regions of the piecewise distribution have been estimated, graphically concatenate and display the result.

The following code calls the CDF and inverse CDF methods of the Pareto tails object of interest with data other than that upon which the fit is based. The referenced methods have access to the fitted state. They are now invoked to select and analyze specific regions of the probability curve, acting as a powerful data filtering mechanism.

For reference, the plot also includes a zero-mean Gaussian CDF of the same standard deviation. To a degree, the variation in options prices reflect the extent to which the distribution of each asset differs from this normal curve.

```
minProbability = cdf(tails{index}, (min(returns(:,index))));
maxProbability = cdf(tails{index}, (max(returns(:,index))));

pLowerTail = linspace(minProbability  , tailFraction    , 200); % lower tail
pUpperTail = linspace(1 - tailFraction, maxProbability  , 200); % upper tail
pInterior  = linspace(tailFraction    , 1 - tailFraction, 200); % interior

plot(icdf(tails{index}, pLowerTail), pLowerTail, 'red'  , 'LineWidth', 2)
hold on
grid on
```

```
plot(icdf(tails{index}, pInterior) , pInterior , 'black', 'LineWidth', 2)
plot(icdf(tails{index}, pUpperTail), pUpperTail, 'blue' , 'LineWidth', 2)

limits = axis;
x = linspace(limits(1), limits(2));
plot(x, normcdf(x, 0, std(returns(:,index))), 'green', 'LineWidth', 2)

fig = gcf;
fig.Color = [1 1 1];
hold off
xlabel('Centered Return')
ylabel('Probability')
title (['Semi-Parametric/Piecewise CDF: ' series{index}])
legend({'Pareto Lower Tail' 'Kernel Smoothed Interior' ...
        'Pareto Upper Tail' 'Gaussian with Same \sigma'}, 'Location', 'NorthWest')
```



The lower and upper tail regions, displayed in red and blue, respectively, are suitable for extrapolation, while the kernel-smoothed interior, in black, is suitable for interpolation.

Copula Calibration

The Statistics and Machine Learning Toolbox software includes functionality that calibrates and simulates Gaussian and *t* copulas.

Using the daily index returns, estimate the parameters of the Gaussian and *t* copulas using the function `copulafit`. Since a *t* copula becomes a Gaussian copula as the scalar degrees of freedom parameter (DoF) becomes infinitely large, the two copulas are really of the same family, and therefore share a linear correlation matrix as a fundamental parameter.

Although calibration of the linear correlation matrix of a Gaussian copula is straightforward, the calibration of a *t* copula is not. For this reason, the Statistics and Machine Learning Toolbox software offers two techniques to calibrate a *t* copula:

- The first technique performs maximum likelihood estimation (MLE) in a two-step process. The inner step maximizes the log-likelihood with respect to the linear correlation matrix, given a fixed value for the degrees of freedom. This conditional maximization is placed within a 1-D maximization with respect to the degrees of freedom, thus maximizing the log-likelihood over all parameters. The function being maximized in this outer step is known as the profile log-likelihood for the degrees of freedom.

- The second technique is derived by differentiating the log-likelihood function with respect to the linear correlation matrix, assuming the degrees of freedom is a fixed constant. The resulting expression is a non-linear equation that can be solved iteratively for the correlation matrix. This technique approximates the profile log-likelihood for the degrees of freedom parameter for large sample sizes. This technique is usually significantly faster than the true maximum likelihood technique outlined above; however, you should not use it with small or moderate sample sizes as the estimates and confidence limits may not be accurate.

When the uniform variates are transformed by the empirical CDF of each margin, the calibration method is often known as canonical maximum likelihood (CML). The following code segment first transforms the daily centered returns to uniform variates by the piecewise, semi-parametric CDFs derived above. It then fits the Gaussian and *t* copulas to the transformed data:

```
U = zeros(size(returns));

for i = 1:nIndices
    U(:,i) = cdf(tails{i}, returns(:,i));    % transform each margin to uniform
end

options     = statset('Display', 'off', 'TolX', 1e-4);
[rhoT, DoF] = copulafit('t', U, 'Method', 'ApproximateML', 'Options', options);
rhoG        = copulafit('Gaussian', U);
```

The estimated correlation matrices are quite similar but not identical.

```
corrcoef(returns)  % linear correlation matrix of daily returns


ans =
```

```
    1.0000    0.4813    0.5058    0.1854    0.4573    0.6526
    0.4813    1.0000    0.8485    0.2261    0.8575    0.5102
    0.5058    0.8485    1.0000    0.2001    0.7650    0.6136
    0.1854    0.2261    0.2001    1.0000    0.2295    0.1439
    0.4573    0.8575    0.7650    0.2295    1.0000    0.4617
    0.6526    0.5102    0.6136    0.1439    0.4617    1.0000
```

```
rhoG                % linear correlation matrix of the optimized Gaussian copula
```

```
rhoG =

    1.0000    0.4745    0.5018    0.1857    0.4721    0.6622
    0.4745    1.0000    0.8606    0.2393    0.8459    0.4912
    0.5018    0.8606    1.0000    0.2126    0.7608    0.5811
    0.1857    0.2393    0.2126    1.0000    0.2396    0.1494
    0.4721    0.8459    0.7608    0.2396    1.0000    0.4518
    0.6622    0.4912    0.5811    0.1494    0.4518    1.0000
```

```
rhoT                % linear correlation matrix of the optimized t copula
```

```
rhoT =

    1.0000    0.4671    0.4858    0.1907    0.4734    0.6521
    0.4671    1.0000    0.8871    0.2567    0.8500    0.5122
    0.4858    0.8871    1.0000    0.2326    0.7723    0.5877
    0.1907    0.2567    0.2326    1.0000    0.2503    0.1539
    0.4734    0.8500    0.7723    0.2503    1.0000    0.4769
    0.6521    0.5122    0.5877    0.1539    0.4769    1.0000
```

Note the relatively low degrees of freedom parameter obtained from the *t* copula calibration, indicating a significant departure from a Gaussian situation.

```
DoF                 % scalar degrees of freedom parameter of the optimized t copula
```

```
DoF =

    4.8613
```

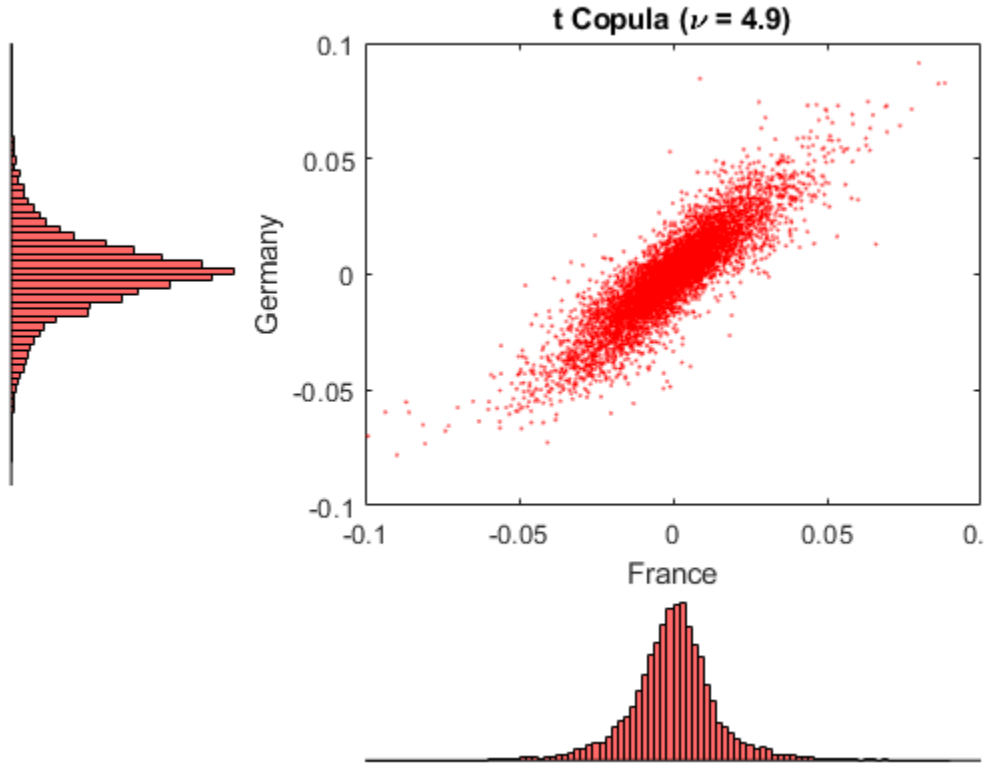
Copula Simulation

Now that the copula parameters have been estimated, simulate jointly-dependent uniform variates using the function `copularnd`.

Then, by extrapolating the Pareto tails and interpolating the smoothed interior, transform the uniform variates derived from `copularnd` to daily centered returns via the inverse CDF of each index. These simulated centered returns are consistent with those obtained from the historical dataset. The returns are assumed to be independent in time, but at any point in time possess the dependence and rank correlation induced by the given copula.

The following code segment illustrates the dependence structure by simulating centered returns using the *t* copula. It then plots a 2-D scatter plot with marginal histograms for the French CAC 40 and German DAX using the Statistics and Machine Learning Toolbox `scatterhist` function. The French and German indices were chosen simply because they have the highest correlation of the available data.

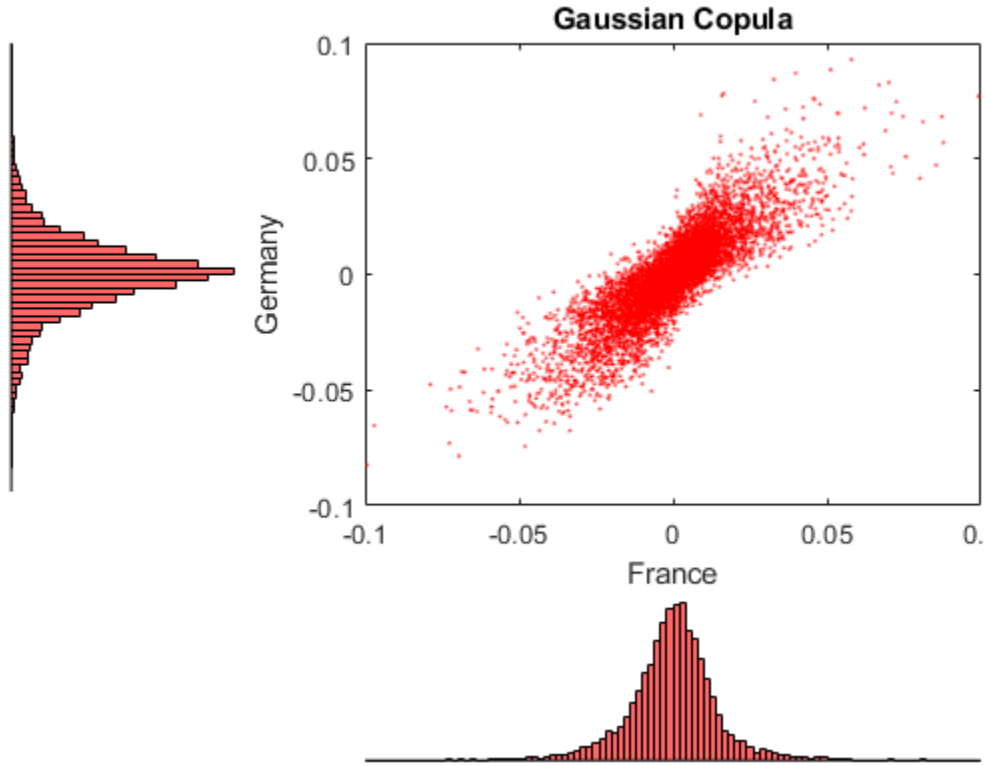
```
nPoints = 10000;                            % # of simulated observations

s = RandStream.getGlobalStream();
reset(s)

R = zeros(nPoints, nIndices);               % pre-allocate simulated returns array
U = copularnd('t', rhoT, DoF, nPoints);     % simulate U(0,1) from t copula

for j = 1:nIndices
    R(:,j) = icdf(tails{j}, U(:,j));
end

h = scatterhist(R(:,2), R(:,3),'Color','r','Marker','.','MarkerSize',1);
fig = gcf;
fig.Color = [1 1 1];
y1 = ylim(h(1));
y3 = ylim(h(3));
xlim(h(1), [-.1 .1])
ylim(h(1), [-.1 .1])
xlim(h(2), [-.1 .1])
ylim(h(3), [(y3(1) + (-0.1 - y1(1)))  (y3(2) + (0.1 - y1(2)))])
xlabel('France')
ylabel('Germany')
title(['t Copula (\nu = ' num2str(DoF,2) ')'])
```



Now simulate and plot centered returns using the Gaussian copula.

```
reset(s)
R = zeros(nPoints, nIndices);              % pre-allocate simulated returns array
U = copularnd('Gaussian', rhoG, nPoints); % simulate U(0,1) from Gaussian copula

for j = 1:nIndices
    R(:,j) = icdf(tails{j}, U(:,j));
end

h = scatterhist(R(:,2), R(:,3),'Color','r','Marker','.','MarkerSize',1);
fig = gcf;
fig.Color = [1 1 1];
y1 = ylim(h(1));
```

```
y3 = ylim(h(3));
xlim(h(1), [-.1 .1])
ylim(h(1), [-.1 .1])
xlim(h(2), [-.1 .1])
ylim(h(3), [(y3(1) + (-0.1 - y1(1)))  (y3(2) + (0.1 - y1(2)))])
xlabel('France')
ylabel('Germany')
title('Gaussian Copula')
```



Examine these two figures. There is a strong similarity between the miniature histograms on the corresponding axes of each figure. This similarity is not coincidental.

Both copulas simulate uniform random variables, which are then transformed to daily centered returns by the inverse CDF of the piecewise distribution of each index.

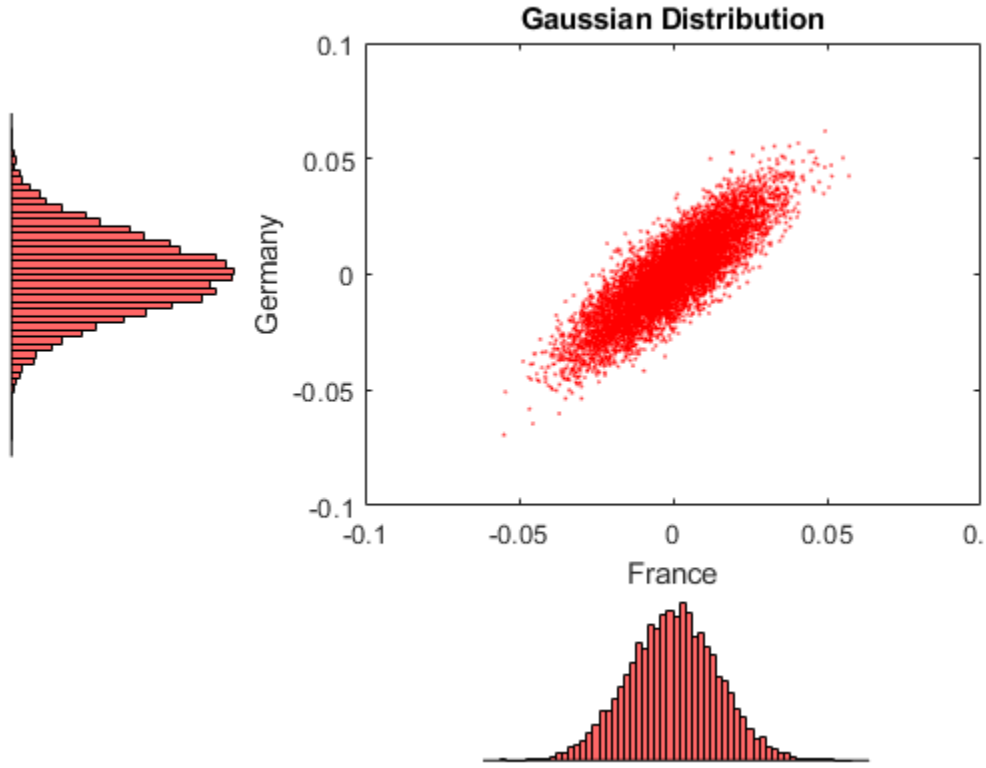
Therefore, the simulated returns of any given index are identically distributed regardless of the copula.

However, the scatter graph of each figure indicates the dependence structure associated with the given copula, and in contrast to the univariate margins shown in the histograms, the scatter graphs are distinct.

Once again, the copula defines a dependence structure regardless of its margins, and therefore offers many features not limited to calibration alone.

For reference, simulate and plot centered returns using the Gaussian distribution, which underlies the traditional Brownian motion model.

```
reset(s)
R = mvnrnd(zeros(1,nIndices), cov(returns), nPoints);
h = scatterhist(R(:,2), R(:,3),'Color','r','Marker','.','MarkerSize',1);
fig = gcf;
fig.Color = [1 1 1];
y1 = ylim(h(1));
y3 = ylim(h(3));
xlim(h(1), [-.1 .1])
ylim(h(1), [-.1 .1])
xlim(h(2), [-.1 .1])
ylim(h(3), [(y3(1) + (-0.1 - y1(1)))  (y3(2) + (0.1 - y1(2)))])
xlabel('France')
ylabel('Germany')
title('Gaussian Distribution')
```



American Option Pricing Using the Longstaff & Schwartz Approach

Now that the copulas have been calibrated, compare the prices of at-the-money American basket options derived from various approaches. To simply the analysis, assume that:

- All indices begin at 100.
- The portfolio holds a single unit, or share, of each index such that the value of the portfolio at any time is the sum of the values of the individual indices.
- The option expires in 3 months.
- The information derived from the daily data is annualized.
- Each calendar year is composed of 252 trading days.

- Index levels are simulated daily.
- The option may be exercised at the end of every trading day and approximates the American option as a Bermudan option.

Now compute the parameters common to all simulation methods:

```
dt       = 1 / 252;                  % time increment = 1 day = 1/252 years
yields   = Data(:,end);              % daily effective yields
yields   = 360 * log(1 + yields);    % continuously-compounded, annualized yields
r        = mean(yields);             % historical 3M Euribor average
X        = repmat(100, nIndices, 1); % initial state vector
strike   = sum(X);                   % initialize an at-the-money basket

nTrials  = 100;                      % # of independent trials
nPeriods = 63;   % # of simulation periods: 63/252 = 0.25 years = 3 months
```

Now create two separable multi-dimensional market models in which the riskless return and volatility exposure matrices are both diagonal.

While both are diagonal GBM models with identical risk-neutral returns, the first is driven by a correlated Brownian motion and explicitly specifies the sample linear correlation matrix of centered returns. This correlated Brownian motion process is then weighted by a diagonal matrix of annualized index volatilities or standard deviations.

As an alternative, the same model could be driven by an uncorrelated Brownian motion (*standard Brownian motion*) by specifying `correlation` as an identity matrix, or by simply accepting the default value. In this case, the exposure matrix `sigma` is specified as the lower Cholesky factor of the index return covariance matrix. Because the copula-based approaches simulate dependent random numbers, the diagonal exposure form is chosen for consistency. For further details, see *Alternatives for Inducing Dependence & Correlation*.

```
sigma       = std(returns) * sqrt(252);    % annualized volatility
correlation = corrcoef(returns);           % correlated Gaussian disturbances
GBM1        = gbm(diag(r(ones(1,nIndices))), diag(sigma), 'StartState', X, ...
                  'Correlation'             , correlation);
```

Now create the second model driven by the Brownian copula with an identity matrix `sigma`.

```
GBM2 = gbm(diag(r(ones(1,nIndices))), eye(nIndices), 'StartState', X);
```

The newly created model may seem unusual, but it highlights the flexibility of the SDE architecture.

When working with copulas, it is often convenient to allow the random number generator function Z(t,X) to induce dependence (of which the traditional notion of linear correlation is a special case) with the copula, and to induce magnitude or scale of variation (similar to volatility or standard deviation) with the semi-parametric CDF and inverse CDF transforms. Since the CDF and inverse CDF transforms of each index inherit the characteristics of historical returns, this also explains why the returns are now centered.

In the following sections, statements like:

```
z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 'Gaussian');
```

or

```
z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 't');
```

fit the Gaussian and *t* copula dependence structures, respectively, and the semi-parametric margins to the centered returns scaled by the square root of the number of trading days per year (252). This scaling does not annualize the daily centered returns. Instead, it scales them such that the volatility remains consistent with the diagonal annualized exposure matrix `sigma` of the traditional Brownian motion model (GBM1) created previously.

In this example, you also specify an end-of-period processing function that accepts time followed by state (t,X), and records the sample times and value of the portfolio as the single-unit weighted average of all indices. This function also shares this information with other functions designed to price American options with a constant riskless rate using the least squares regression approach of Longstaff & Schwartz.

```
f = Example_LongstaffSchwartz(nPeriods, nTrials)
```

```
f =

  struct with fields:

    LongstaffSchwartz: @Example_LongstaffSchwartz/saveBasketPrices
            CallPrice: @Example_LongstaffSchwartz/getCallPrice
             PutPrice: @Example_LongstaffSchwartz/getPutPrice
               Prices: @Example_LongstaffSchwartz/getBasketPrices
```

Now simulate independent trials of equity index prices over 3 calendar months using the default `simByEuler` method. No outputs are requested from the simulation methods; in fact, the simulated prices of the individual indices which comprise the basket are unnecessary. Call option prices are reported for convenience:

```
reset(s)

simByEuler(GBM1, nPeriods, 'nTrials'  , nTrials, 'DeltaTime', dt, ...
                           'Processes', f.LongstaffSchwartz);

BrownianMotionCallPrice = f.CallPrice(strike, r);
BrownianMotionPutPrice  = f.PutPrice (strike, r);

reset(s)

z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 'Gaussian');
f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM2, nPeriods, 'nTrials'  , nTrials, 'DeltaTime', dt, ...
                           'Processes', f.LongstaffSchwartz, 'Z', z);

GaussianCopulaCallPrice = f.CallPrice(strike, r);
GaussianCopulaPutPrice  = f.PutPrice (strike, r);
```

Now repeat the copula simulation with the *t* copula dependence structure. You use the same model object for both copulas; only the random number generator and option pricing functions need to be re-initialized.

```
reset(s)

z = Example_CopulaRNG(returns * sqrt(252), nPeriods, 't');
f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM2, nPeriods, 'nTrials'  , nTrials, 'DeltaTime', dt, ...
                           'Processes', f.LongstaffSchwartz, 'Z', z);

tCopulaCallPrice = f.CallPrice(strike, r);
tCopulaPutPrice  = f.PutPrice (strike, r);
```

Finally, compare the American put and call option prices obtained from all models.

```
disp(' ')
fprintf('                    # of Monte Carlo Trials: %8d\n'    , nTrials)
fprintf('                    # of Time Periods/Trial: %8d\n\n'  , nPeriods)
```

```
fprintf(' Brownian Motion American Call Basket Price: %8.4f\n'  , BrownianMotionCallPri
fprintf(' Brownian Motion American Put  Basket Price: %8.4f\n\n', BrownianMotionPutPric
fprintf(' Gaussian Copula American Call Basket Price: %8.4f\n'  , GaussianCopulaCallPri
fprintf(' Gaussian Copula American Put  Basket Price: %8.4f\n\n', GaussianCopulaPutPric
fprintf('        t Copula American Call Basket Price: %8.4f\n'  , tCopulaCallPrice)
fprintf('        t Copula American Put  Basket Price: %8.4f\n'  , tCopulaPutPrice)
```

```
                  # of Monte Carlo Trials:      100
                  # of Time Periods/Trial:       63

 Brownian Motion American Call Basket Price:  25.9456
 Brownian Motion American Put  Basket Price:  16.4132

 Gaussian Copula American Call Basket Price:  24.5711
 Gaussian Copula American Put  Basket Price:  17.4229

        t Copula American Call Basket Price:  22.6220
        t Copula American Put  Basket Price:  20.9983
```

This analysis represents only a small-scale simulation. If the simulation is repeated with 100,000 trials, the following results are obtained:

```
                  # of Monte Carlo Trials:   100000
                  # of Time Periods/Trial:       63

Brownian Motion American Call Basket Price:  20.2214
Brownian Motion American Put  Basket Price:  16.5355

Gaussian Copula American Call Basket Price:  20.6097
Gaussian Copula American Put  Basket Price:  16.5539

       t Copula American Call Basket Price:  21.1273
       t Copula American Put  Basket Price:  16.6873
```

Interestingly, the results agree closely. Put option prices obtained from copulas exceed those of Brownian motion by less than 1%.

A Note on Volatility and Interest Rate Scaling

The same option prices could also be obtained by working with unannualized (in this case, daily) centered returns and riskless rates, where the time increment `dt` = 1 day rather than 1/252 years. In other words, portfolio prices would still be simulated every trading day; the data is simply scaled differently.

Although not executed, and by first resetting the random stream to its initial internal state, the following code segments work with daily centered returns and riskless rates and produce the same option prices.

Gaussian Distribution/Brownian Motion & Daily Data:

```
reset(s)

f    = Example_LongstaffSchwartz(nPeriods, nTrials);
GBM1 = gbm(diag(r(ones(1,nIndices))/252), diag(std(returns)), 'StartState', X, ...
          'Correlation', correlation);

simByEuler(GBM1, nPeriods, 'nTrials'  , nTrials, 'DeltaTime', 1, ...
                           'Processes', f.LongstaffSchwartz);

BrownianMotionCallPrice = f.CallPrice(strike, r/252)
BrownianMotionPutPrice  = f.PutPrice (strike, r/252)
```

Gaussian Copula & Daily Data:

```
reset(s)

z    = Example_CopulaRNG(returns, nPeriods, 'Gaussian');
f    = Example_LongstaffSchwartz(nPeriods, nTrials);
GBM2 = gbm(diag(r(ones(1,nIndices))/252),   eye(nIndices), 'StartState', X);

simByEuler(GBM2, nPeriods, 'nTrials'  , nTrials, 'DeltaTime',   1, ...
                           'Processes', f.LongstaffSchwartz , 'Z', z);

GaussianCopulaCallPrice = f.CallPrice(strike, r/252)
GaussianCopulaPutPrice  = f.PutPrice (strike, r/252)
```

t Copula & Daily Data:

```
reset(s)

z = Example_CopulaRNG(returns, nPeriods, 't');
f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM2, nPeriods, 'nTrials'  , nTrials, 'DeltaTime',   1, ...
                           'Processes', f.LongstaffSchwartz , 'Z', z);
```

```
tCopulaCallPrice = f.CallPrice(strike, r/252)
tCopulaPutPrice  = f.PutPrice (strike, r/252)
```

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Pricing American Basket Options by Monte Carlo Simulation”
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 17-107
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5

Improving Performance of Monte Carlo Simulation with Parallel Computing

This example shows how to improve the performance of a Monte Carlo simulation using Parallel Computing Toolbox.

Consider a geometric Brownian motion (GBM) process in which you want to incorporate alternative asset price dynamics. Specifically, suppose you want to include a time-varying short rate as well as a volatility surface. The process to simulate is written as

$$dS(t) = r(t)S(t)dt + V(t,S(t))S(t)dW(t)$$

for stock price *S(t)*, rate of return *r(t)*, volatility *V(t,S(t))*, and Brownian motion *W(t)*. In this example, the rate of return is a deterministic function of time and the volatility is a function of both time and current stock price. Both the return and volatility are defined on a discrete grid such that intermediate values is obtained by linear interpolation. For example, such a simulation could be used to support the pricing of thinly traded options.

To include a time series of riskless short rates, suppose that you derive the following deterministic short rate process as a function of time.

```
times = [0 0.25 0.5 1 2 3 4 5 6 7 8 9 10];  % in years
rates = [0.1 0.2 0.3 0.4 0.5 0.8 1.25 1.75 2.0 2.2 2.25 2.50 2.75]/100;
```

Suppose that you then derive the following volatility surface whose columns correspond to simple relative moneyness, or the ratio of the spot price to strike price, and whose rows correspond to time to maturity, or tenor.

```
surface = [28.1 25.3 20.6 16.3 11.2  6.2  4.9  4.9  4.9  4.9  4.9  4.9
           22.7 19.8 15.4 12.6  9.6  6.7  5.2  5.2  5.2  5.2  5.2  5.2
           21.7 17.6 13.7 11.5  9.4  7.3  5.7  5.4  5.4  5.4  5.4  5.4
           19.8 16.4 12.9 11.1  9.3  7.6  6.2  5.6  5.6  5.6  5.6  5.6
           18.6 15.6 12.5 10.8  9.3  7.8  6.6  5.9  5.9  5.9  5.9  5.9
           17.4 13.8 11.7 10.8  9.9  9.1  8.5  7.9  7.4  7.3  7.3  7.3
           17.1 13.7 12.0 11.2 10.6 10.0  9.5  9.1  8.8  8.6  8.4  8.0
           17.5 13.9 12.5 11.9 11.4 10.9 10.5 10.2  9.9  9.6  9.4  9.0
           18.3 14.9 13.7 13.2 12.8 12.4 12.0 11.7 11.4 11.2 11.0 10.8
           19.2 19.6 14.2 13.9 13.4 13.0 13.2 12.5 12.1 11.9 11.8 11.4]/100;

tenor = [0 0.25 0.50 0.75 1 2 3 5 7 10];   % in years
moneyness = [0.25 0.5 0.75 0.8 0.9 1 1.10 1.25 1.50 2 3 5];
```

Set the simulation parameters. The following assumes that the price of the underlying asset is initially equal to the strike price and that the price of the underlying asset is

simulated monthly for 10 years, or 120 months. As a simple illustration, 100 sample paths are simulated.

```
price = 100;
strike = 100;
dt = 1/12;
nPeriods = 120;
nTrials = 100;
```

For reproducibility, set the random number generator to its default, and draw the Gaussian random variates that drive the simulation. Generating the random variates is not necessary to incur the performance improvement of parallel computation, but doing so allows the resulting simulated paths to match those of the conventional (that is, non-parallelized) simulation. Moreover, generating independent Gaussian random variates as inputs also guarantees that all simulated paths are independent.

```
rng default
Z = randn(nPeriods,1,nTrials);
```

Create the return and volatility functions and the GBM model using the `gbm` constructor. Notice that the rate of return is a deterministic function of time, and therefore accepts simulation time as its only input argument. In contrast, the volatility must account for the moneyness and is a function of both time and stock price. Moreover, since the volatility surface is defined as a function of time to maturity rather than simulation time, the volatility function subtracts the current simulation time from the last time at which the price process is simulated (10 years). This ensures that as the simulation time approaches its terminal value, the time to maturity of the volatility surface approaches zero. Although far more elaborate return and volatility functions could be used if desired, the following assumes simple linear interpolation.

```
mu = @(t) interp1(times,rates,t);
sigma = @(t,S) interp2(moneyness,tenor,surface,S/strike,tenor(end)-t);
mdl = gbm(mu,sigma,'StartState',price);
```

Simulate the paths of the underlying geometric Brownian motion without parallelization.

```
tStart = tic;
paths = simBySolution(mdl,nPeriods,'nTrials',nTrials,'DeltaTime',dt,'Z',Z);
time1 = toc(tStart);
```

Simulate the paths in parallel using a `parfor` loop. For users licensed to access the Parallel Computing Toolbox, the following code segment automatically creates a parallel pool using the default local profile. If desired, this behavior can be changed by first

calling the `parpool` function. If a parallel pool is not already created, the following simulation will likely be slower than the previous simulation without parallelization. In this case, rerun the following simulation to assess the true benefits of parallelization.

```
tStart = tic;
parPaths = zeros(nPeriods+1,1,nTrials);
parfor i = 1:nTrials
    parPaths(:,:,i) = simBySolution(mdl,nPeriods,'DeltaTime',dt,'Z',Z(:,:,i));
end
time2 = toc(tStart);
```

If you examine any given path obtained without parallelization to the corresponding path with parallelization, you see that they are identical. Moreover, although relative performance varies, the results obtained with parallelization will generally incur a significant improvement. To assess the performance improvement, examine the runtime of each approach in seconds and speedup gained from simulating the paths in parallel.

```
time1                % elapsed time of conventional simulation, in seconds
time2                % elapsed time of parallel simulation, in seconds
speedup = time1/time2 % speedup factor

time1 =
    6.1329
time2 =
    2.5918
speedup =
    2.3663
```

See Also

`bm` | `cev` | `cir` | `diffusion` | `drift` | `gbm` | `heston` | `hwv` | `interpolate` | `parpool` | `sde` | `sdeddo` | `sdeld` | `sdemrd` | `simByEuler` | `simBySolution` | `simBySolution` | `simulate` | `ts2func`

Related Examples

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Pricing American Basket Options by Monte Carlo Simulation”
- “Base SDE Models” on page 17-16

- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

More About

- “SDEs” on page 17-2
- “SDE Models” on page 17-8
- “SDE Class Hierarchy” on page 17-5
- “Performance Considerations” on page 17-76

Functions — Alphabetical List

abs2active

Convert constraints from absolute to active format

Syntax

```
ActiveConSet = abs2active(AbsConSet,Index)
```

Description

`ActiveConSet = abs2active(AbsConSet,Index)` transforms a constraint matrix to an equivalent matrix expressed in active weight format (relative to the index).

Input Arguments

`AbsConSet`

Portfolio linear inequality constraint matrix expressed in absolute weight format. `AbsConSet` is formatted as `[A b]` such that `A*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 (for example, a standard portfolio optimization imposes a sum-to-one budget constraint).

Output Arguments

`ActiveConSet`

The transformed portfolio linear inequality constraint matrix expressed in active weight format, also of the form `[A b]` such that `A*w <= b`. The value `w` represents a vector of active asset weights (relative to the index portfolio) whose elements sum to zero.

Examples

Set up constraints for a portfolio optimization for portfolio `w0` with constraints in the form `A*w <= b`, where `w` is absolute portfolio weights. (Absolute weights do not depend on the tracking portfolio.) Use `abs2active` to convert constraints in terms of absolute weights into constraints in terms of active portfolio weights, defined relative to the tracking portfolio `w0`. Assume three assets with the following mean and covariance of asset returns:

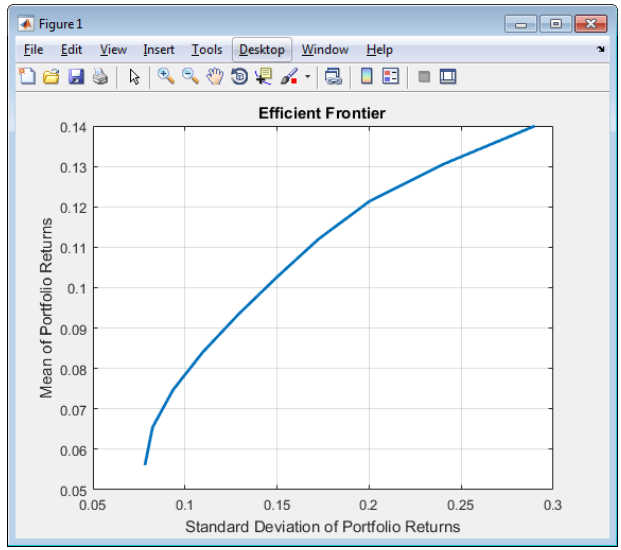
```
m = [ 0.14; 0.10; 0.05 ];
C = [ 0.29^2 0.4*0.29*0.17 0.1*0.29*0.08; 0.4*0.29*0.17 0.17^2 0.3*0.17*0.08;...
0.1*0.29*0.08 0.3*0.17*0.08 0.08^2 ];
```

Absolute portfolio constraints are the typical ones (weights sum to 1 and fall from 0 through 1), create the `A` and `b` matrices using `portcons`:

```
AbsCons = portcons('PortValue',1,3,'AssetLims', [0; 0; 0], [1; 1; 1;]);
```

Use the `Portfolio` object to determine the efficient frontier:

```
p = Portfolio('AssetMean', m, 'AssetCovar', C);
p = p.setInequality(AbsCons(:,1:end-1), AbsCons(:,end));
p.plotFrontier;
```



The tracking portfolio `w0` is:

```
w0 = [ 0.1; 0.55; 0.35 ];
```

Use `abs2active` to compute the constraints for active portfolio weights:

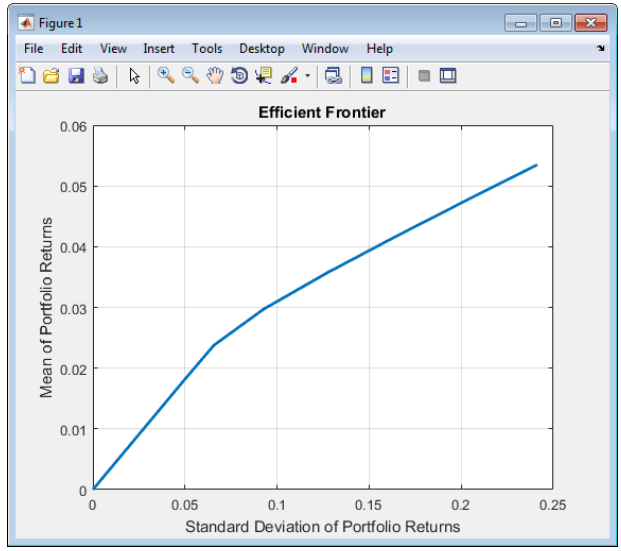
```
ActCons = abs2active(AbsCons, w0)
```

This returns:

```
ActCons =

    1.0000    1.0000    1.0000         0
   -1.0000   -1.0000   -1.0000         0
    1.0000         0         0    0.9000
         0    1.0000         0    0.4500
         0         0    1.0000    0.6500
   -1.0000         0         0    0.1000
         0   -1.0000         0    0.5500
         0         0   -1.0000    0.3500
```

Use the `Portfolio` object `p` and its efficient frontier to demonstrate expected returns and risk relative to the tracking portfolio `w0`:

```
p = p.setInequality(ActCons(:,1:end-1), ActCons(:,end));
p.plotFrontier;
```



Note, when using `abs2active` to compute "active constraints" for use with a Portfolio object, don't use the Portfolio object's default constraints because the relative weights can be positive or negative (the `setDefaultConstraints` function specifies weights to be nonnegative).

Algorithms

`abs2active` transforms a constraint matrix to an equivalent matrix expressed in active weight format (relative to the index). The transformation equation is

$$Aw_{absolute} = A\left(w_{active} + w_{index}\right) \le b_{absolute}.$$

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.

See Also

`Portfolio` | `active2abs` | `pcalims` | `pcglims` | `pcpval` | `portcons` | `setInequality`

Topics

"Data Transformation and Frequency Conversion" on page 12-12

Introduced before R2006a

accrfrac

Fraction of coupon period before settlement

Syntax

```
Fraction = accrfrac(Settle,Maturity)
Fraction = accrfrac( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)
```

Description

`Fraction = accrfrac(Settle,Maturity)` returns the fraction of the coupon period before settlement.

Use `accrfrac` for computing accrued interest. `accrfrac` calculates accrued interest for bonds with regular or odd first or last coupon periods.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`Fraction = accrfrac( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate)` returns the fraction of the coupon period before settlement with optional inputs.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Examples

Find Accrued Interest for a Bond

This example shows how to find the accrued interest for given bond data.

```
Settle = '14-Mar-1997';
Maturity = ['30-Nov-2000'
            '31-Dec-2000'
            '31-Jan-2001'];
Period = 2;
Basis = 0;
EndMonthRule = 1;

Fraction = accrfrac(Settle, Maturity, Period, Basis,...
EndMonthRule)

Fraction =

    0.5714
    0.4033
    0.2320
```

Find Accrued Interest for a Bond Using a datetime Array

This example shows how to find the accrued interest for a given bond's data using a datetime array.

```
Fraction = accrfrac(datetime('14-Mar-1997', 'Locale', 'en_US'), ['30-Nov-2000'; '31-Dec

Fraction =

    0.5714
    0.4033
    0.2320
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date

serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or a `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using a `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`Fraction` — Fraction of coupon period before settlement
vector

Fraction of the coupon period before settlement, returned as an `NUMBONDS`-by-`1` vector.

See Also

`cfamounts` | `cfdates` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz` | `datetime`

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Introduced before R2006a

acrubond

Accrued interest of security with periodic interest payments

Syntax

```
AccruInterest = acrubond(IssueDate,Settle,FirstCouponDate,Face,
CouponRate)
AccruInterest = acrubond(___,Period,Basis)
```

Description

`AccruInterest = acrubond(IssueDate,Settle,FirstCouponDate,Face, CouponRate)` returns the accrued interest for a security with periodic interest payments. `acrubond` 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.

`AccruInterest = acrubond(___,Period,Basis)` adds optional arguments for `Period` and `Basis`.

Examples

Find Accrued Interest of a Bond with Periodic Interest Payments

This example shows how to find the accrued interest for a bond with semiannual interest payments.

```
AccruInterest = acrubond('31-jan-1983',  '1-mar-1993', ...
                '31-jul-1983',  100,  0.1,  2,  0)
```

```
AccruInterest = 0.8011
```

Find Accrued Interest of a Bond with Periodic Interest Payments Using datetime Inputs

This example shows how to use `datetime` inputs to find the accrued interest for a bond with semiannual interest payments.

```
AccruInterest = acrubond(datetime('31-jan-1983','Locale','en_US'),datetime('1-mar-1993'
100,  0.1,  2,  0)

AccruInterest = 0.8011
```

- “Coupon Date Calculations” on page 2-34

Input Arguments

`IssueDate` — Issue date of security

serial date number | date character vector | datetime

Issue date of the security, specified as a scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Settle` — Settlement date of security

serial date number | date character vector | datetime

Settlement date of the security, specified as a scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — First coupon date of security

serial date number | date character vector | datetime

First coupon date of the security, specified as a scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value of security
numeric

Redemption value (par value) of the security, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Coupon rate of security
decimal fraction

Coupon rate of the security, specified as a scalar or a `NINST`-by-`1` vector of decimal fraction values.

Data Types: `double`

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

(Optional) Number of coupon payments per year for the security, specified as scalar or a `NINST`-by-`1` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis for the security, specified as a scalar or a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`AccruInterest` — Accrued interest

numeric

Accrued interest for the security, returned as a scalar or a `NINST`-by-`1` vector.

See Also

`accrfrac` | `acrudisc` | `bndprice` | `bndyield` | `cfamounts` | `datenum` | `datetime`

Topics

"Coupon Date Calculations" on page 2-34
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

acrudisc

Accrued interest of discount security paying at maturity

Syntax

```
AccruInterest = acrudisc(Settle,Maturity,Face,Discount)
AccruInterest = acrudisc(___,Period,Basis)
```

Description

`AccruInterest = acrudisc(Settle,Maturity,Face,Discount)` returns the accrued interest of a discount security paid at maturity.

`AccruInterest = acrudisc(___,Period,Basis)` adds optional arguments for `Period` and `Basis`.

Examples

Find Accrued Interest of a Discount Security Paid at Maturity

This example shows how to find the accrued interest of a discount security paid at maturity.

```
AccruInterest = acrudisc('05/01/1992',  '07/15/1992', ...
                100,  0.1,  2,  0)
```

```
AccruInterest = 2.0604
```

Find Accrued Interest of a Discount Security Paid at Maturity Using datetime Inputs

This example shows how to use `datetime` inputs to find the accrued interest of a discount security paid at maturity.

```
AccruInterest = acrudisc(datetime('1-May-1992','Locale','en_US'),datetime('15-Jul-1992'
100,  0.1,  2,  0)

AccruInterest = 2.0604
```

- “Coupon Date Calculations” on page 2-34

Input Arguments

`Settle` — Settlement date of security

serial date number | date character vector | datetime

Settlement date of the security, specified as a scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of security

serial date number | date character vector | datetime

Maturity date of the security, specified as a scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value of security

numeric

Redemption value (par value) of the security, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`Discount` — Discount rate of security

decimal fraction

Discount rate of the security, specified as a scalar or a `NINST`-by-`1` vector of decimal fraction values.

Data Types: `double`

`Period` — Number of coupon payments per year

`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

(Optional) Number of coupon payments per year for security, specified as scalar or a `NINST`-by-`1` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis for security, specified as a scalar or a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`AccruInterest` — Accrued interest
numeric

Accrued interest, returned as a scalar or a `NINST`-by-`1` vector.

References

[1] Mayle, J. *Standard Securities Calculation Methods.* Volumes I-II, 3rd edition. Formula D.

See Also

`acrubond` | `datetime` | `prdisc` | `prmat` | `ylddisc` | `yldmat`

Topics

“Coupon Date Calculations” on page 2-34
“Fixed-Income Terminology” on page 2-25

Introduced before R2006a

active2abs

Convert constraints from active to absolute format

Syntax

```
AbsConSet = active2abs(ActiveConSet,Index)
```

Arguments

`ActiveConSet`	Portfolio linear inequality constraint matrix expressed in active weight format. `ActiveConSet` is formatted as `[A b]` such that `A*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 (for example, a standard portfolio optimization imposes a sum-to-one budget constraint).

Description

`AbsConSet = active2abs(ActiveConSet,Index)` transforms a constraint matrix to an equivalent matrix expressed in absolute weight format. The transformation equation is

$$Aw_{active} = A\left(w_{absolute} - w_{index}\right) \le b_{active}.$$

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`

Topics

"Data Transformation and Frequency Conversion" on page 12-12

Introduced before R2006a

addEquality

Add linear equality constraints for portfolio weights to existing constraints

Use the `addEquality` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to add linear equality constraints for portfolio weights to existing constraints.

For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-21, “PortfolioCVaR Object Workflow” on page 5-20, and “PortfolioMAD Object Workflow” on page 6-19.

Syntax

```
obj = addEquality(obj,AEquality,bEquality)
```

Description

`obj = addEquality(obj,AEquality,bEquality)` adds linear equality constraints for portfolio weights to existing constraints.

Given a linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

```
AEquality * Port = bEquality
```

This function "stacks" additional linear equality constraints onto any existing linear equality constraints that exist in the input portfolio object. If no constraints exist, this method is the same as `setEquality`.

Examples

Add a Linear Equality Constraint for a Portfolio Object

Use the `addEquality` method to create linear equality constraints. Add another linear equality constraint to ensure that the last three assets constitute 50% of a portfolio.

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % First equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % Second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AEquality);

     1     1     1     0     0
     0     0     1     1     1

disp(p.bEquality);

    0.5000
    0.5000
```

Add a Linear Equality Constraint for a PortfolioCVaR Object

Use the `addEquality` method to create linear equality constraints. Add another linear equality constraint to ensure that the last three assets constitute 50% of a portfolio.

```
p = PortfolioCVaR;
A = [ 1 1 1 0 0 ];    % First equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % Second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AEquality);
```

```
     1     1     1     0     0
     0     0     1     1     1

disp(p.bEquality);

    0.5000
    0.5000
```

Add a Linear Equality Constraint for a PortfolioMAD Object

Use the `addEquality` method to create linear equality constraints. Add another linear equality constraint to ensure that the last three assets constitute 50% of a portfolio.

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ];    % First equality constraint
b = 0.5;
p = setEquality(p, A, b);

A = [ 0 0 1 1 1 ];    % Second equality constraint
b = 0.5;
p = addEquality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AEquality);

     1     1     1     0     0
     0     0     1     1     1

disp(p.bEquality);

    0.5000
    0.5000
```

- "Working with Linear Equality Constraints Using Portfolio Object" on page 4-86
- "Working with Linear Equality Constraints Using PortfolioCVaR Object" on page 5-82
- "Setting Linear Inequality Constraints Using the setInequality and addInequality Functions" on page 6-82

- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`AEquality` — Linear equality constraints formed from matrix

matrix

Linear equality constraints, specified as a matrix.

Note An error results if `AEquality` is empty and `bEquality` is nonempty.

Data Types: `double`

`bEquality` — Linear equality constraints formed from vector

vector

Linear equality constraints, specified as a vector.

Note An error results if `bEquality` is empty and `AEquality` is nonempty.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to add the linear equality constraints for portfolio weights.

  ```
  obj = obj.addEquality(AEquality, bEquality)
  ```

- You can also remove linear equality constraints from a portfolio object using dot notation.

  ```
  obj = obj.setEquality([ ], [ ])
  ```

See Also

`setEquality`

Topics

“Working with Linear Equality Constraints Using Portfolio Object” on page 4-86
“Working with Linear Equality Constraints Using PortfolioCVaR Object” on page 5-82
“Setting Linear Inequality Constraints Using the setInequality and addInequality Functions” on page 6-82
“Portfolio Optimization Examples” on page 4-147
“Portfolio Set for Optimization Using Portfolio Object” on page 4-10
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-10
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-10

Introduced in R2011a

addGroupRatio

Add group ratio constraints for portfolio weights to existing group ratio constraints

Use the `addGroupratio` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to add group ratio constraints for portfolio weights to existing group ratio constraints.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)
```

Description

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)` adds group ratio constraints for portfolio weights to existing group ratio constraints.

Given base and comparison group matrices `GroupA` and `GroupB` and, either `LowerRatio`, or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Note This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` adds group ratio constraints for portfolio weights to existing group ratio constraints with an additional option for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and, either `LowerRatio`, or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Note This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Examples

Add Group Ratio Constraints to a Portfolio Object

Set a group ratio constraint to ensure that the weight in financial assets does not exceed 50% of the weight in nonfinancial assets. Then add another group ratio constraint to ensure that the weight in financial assets constitute at least 20% of the weight in nonfinancial assets of the portfolio.

```
p = Portfolio;
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ];    % odd-numbered companies
GB = [ false false false true true true ];    % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);

     6

disp(p.GroupA);

     1     1     1     0     0     0
     1     0     1     0     1     0

disp(p.GroupB);
```

```
     0     0     0     1     1     1
     0     0     0     1     1     1

disp(p.LowerRatio);

      -Inf
    0.2000

disp(p.UpperRatio);

    0.5000
       Inf
```

Add Group Ratio Constraints to a PortfolioCVaR Object

Set a group ratio constraint to ensure that the weight in financial assets does not exceed 50% of the weight in nonfinancial assets. Then add another group ratio constraint to ensure that the weight in financial assets constitute at least 20% of the weight in nonfinancial assets of the portfolio.

```
p = PortfolioCVaR;
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ];    % odd-numbered companies
GB = [ false false false true true true ];    % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);

     6

disp(p.GroupA);

     1     1     1     0     0     0
     1     0     1     0     1     0

disp(p.GroupB);

     0     0     0     1     1     1
     0     0     0     1     1     1

disp(p.LowerRatio);
```

```
      -Inf
    0.2000

disp(p.UpperRatio);

    0.5000
       Inf
```

Add Group Ratio Constraints to a PortfolioMAD Object

Set a group ratio constraint to ensure that the weight in financial assets does not exceed 50% of the weight in nonfinancial assets. Then add another group ratio constraint to ensure that the weight in financial assets constitute at least 20% of the weight in nonfinancial assets of the portfolio.

```
p = PortfolioMAD;
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = setGroupRatio(p, GA, GB, [], 0.5);

GA = [ true false true false true false ];    % odd-numbered companies
GB = [ false false false true true true ];    % nonfinancial companies
p = addGroupRatio(p, GA, GB, 0.2);

disp(p.NumAssets);

     6

disp(p.GroupA);

     1     1     1     0     0     0
     1     0     1     0     1     0

disp(p.GroupB);

     0     0     0     1     1     1
     0     0     0     1     1     1

disp(p.LowerRatio);

      -Inf
    0.2000

disp(p.UpperRatio);
```

```
    0.5000
       Inf
```

- "Working with Group Ratio Constraints Using Portfolio Object" on page 4-82
- "Working with Group Ratio Constraints Using PortfolioCVaR Object" on page 5-78
- "Working with Group Ratio Constraints Using PortfolioMAD Object" on page 6-75
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`GroupA` — Base groups for comparison

matrix of logical or numerical arrays

Base groups for comparison, specified as a matrix of logical or numerical arrays.

Note The group matrices `GroupA` and `GroupB` are usually indicators of membership in groups, which means that their elements are usually either `0` or `1`. Because of this interpretation, the `GroupA` and `GroupB` matrices can be logical or numerical arrays.

Data Types: `double`

`GroupB` — Comparison group

matrix of logical or numerical arrays

Comparison group, specified as a matrix of logical or numerical arrays.

Note The group matrices `GroupA` and `GroupB` are usually indicators of membership in groups, which means that their elements are usually either `0` or `1`. Because of this interpretation, the `GroupA` and `GroupB` matrices can be logical or numerical arrays.

Data Types: `double`

`LowerRatio` — Lower-bound for ratio of `GroupB` groups to `GroupA` groups

vector

Lower-bound for ratio of `GroupB` groups to `GroupA` groups, specified as a vector.

Note If input is scalar, `LowerRatio` undergoes scalar expansion to be conformable with the group matrices.

Data Types: `double`

`UpperRatio` — Upper-bound for ratio of `GroupB` groups to `GroupA` groups

vector

Upper-bound for ratio of `GroupB` groups to `GroupA` groups, specified as a vector.

Note If input is scalar, `UpperRatio` undergoes scalar expansion to be conformable with the group matrices.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`

- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to add group ratio constraints for the portfolio weights to existing group ratio constraints.

  ```
  obj = obj.addGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio)
  ```

- To remove group ratio constraints from any of the portfolio objects using dot notation, enter empty arrays for the corresponding arrays.

See Also

`setGroupRatio`

Topics

"Working with Group Ratio Constraints Using Portfolio Object" on page 4-82
"Working with Group Ratio Constraints Using PortfolioCVaR Object" on page 5-78
"Working with Group Ratio Constraints Using PortfolioMAD Object" on page 6-75
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

addGroups

Add group constraints for portfolio weights to existing group constraints

Use the `addGroups` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to add group constraints for portfolio weights to existing group constraints.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = addGroups(obj,GroupMatrix,LowerGroup)
obj = addGroups(obj,GroupMatrix,LowerGroup,UpperGroup)
```

Description

`obj = addGroups(obj,GroupMatrix,LowerGroup)` adds group constraints for portfolio weights to existing group constraints.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

```
LowerGroup <= GroupMatrix * Port <= UpperGroup
```

`obj = addGroups(obj,GroupMatrix,LowerGroup,UpperGroup)` adds group constraints for portfolio weights to existing group constraints with an additional option for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

```
LowerGroup <= GroupMatrix * Port <= UpperGroup
```

Examples

Add Group Constraints to a Portfolio Object

Set a group constraint to ensure that the first three assets constitute at most 30% of a portfolio. Then add another group constraint to ensure that the odd-numbered assets constitute at least 20% of a portfolio.

```
p = Portfolio;
G = [ true true true false false ];    % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ];    % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);

     5

disp(p.GroupMatrix);

     1     1     1     0     0
     1     0     1     0     1

disp(p.LowerGroup);

      -Inf
    0.2000

disp(p.UpperGroup);

    0.3000
       Inf
```

Add Group Constraints to a PortfolioCVaR Object

Set a group constraint to ensure that the first three assets constitute at most 30% of a portfolio. Then add another group constraint to ensure that the odd-numbered assets constitute at least 20% of a portfolio.

```
p = PortfolioCVaR;
G = [ true true true false false ];    % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
```

```
G = [ true false true false true ];    % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);

     5

disp(p.GroupMatrix);

     1     1     1     0     0
     1     0     1     0     1

disp(p.LowerGroup);

      -Inf
    0.2000

disp(p.UpperGroup);

    0.3000
       Inf
```

Add Group Constraints to a PortfolioMAD Object

Set a group constraint to ensure that the first three assets constitute at most 30% of a portfolio. Then add another group constraint to ensure that the odd-numbered assets constitute at least 20% of a portfolio.

```
p = PortfolioMAD;
G = [ true true true false false ];    % group matrix for first group constraint
p = setGroups(p, G, [], 0.3);
G = [ true false true false true ];    % group matrix for second group constraint
p = addGroups(p, G, 0.2);
disp(p.NumAssets);

     5

disp(p.GroupMatrix);

     1     1     1     0     0
     1     0     1     0     1

disp(p.LowerGroup);
```

```
      -Inf
    0.2000

disp(p.UpperGroup);

    0.3000
       Inf
```

- "Working with Group Constraints Using Portfolio Object" on page 4-78
- "Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
- "Working with Group Constraints Using PortfolioMAD Object" on page 6-71
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`GroupMatrix` — Group constraint matrix

matrix

Group constraint matrix, specified as a matrix.

Note The group matrix `GroupMatrix` often indicates membership in groups, which means that its elements are usually either `0` or `1`. Because of this interpretation,`GroupMatrix` can be a logical or numerical matrix.

Data Types: `double`

`LowerGroup` — Lower bound for group constraints
vector

Lower bound for group constraints, specified as a vector.

Note If input is scalar, `LowerGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

Data Types: `double`

`UpperGroup` — Upper bound for group constraints
vector

Upper bound for group constraints, specified as a vector.

Note If input is scalar, `UpperGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object
object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to add group constraints for portfolio weights.

  ```
  obj = obj.addGroups(GroupMatrix, LowerGroup, UpperGroup)
  ```

- To remove group constraints from any of the portfolio objects using dot notation, enter empty arrays for the corresponding arrays.

See Also

`setGroups`

Topics

"Working with Group Constraints Using Portfolio Object" on page 4-78
"Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
"Working with Group Constraints Using PortfolioMAD Object" on page 6-71
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

addInequality

Add linear inequality constraints for portfolio weights to existing constraints

Use the `addInequality` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to add linear inequality constraints for portfolio weights to existing constraints.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = addInequality(obj,AInequality,bInequality)
```

Description

`obj = addInequality(obj,AInequality,bInequality)` adds linear inequality constraints for portfolio weights to existing constraints.

Given a linear inequality constraint matrix `AInequality` and vector `bInequality`, every weight in a portfolio `Port` must satisfy the following:

```
AInequality * Port = bInequality
```

This function "stacks" additional linear inequality constraints onto any existing linear inequality constraints that exist in the input portfolio object. If no constraints exist, this function is the same as `setInequality`.

Examples

Add Linear Inequality Constraint to a Portfolio Object

Set a linear inequality constraint to ensure that the first three assets constitute at most 50% of a portfolio. Then add another linear inequality constraint to ensure that the last three assets constitute at least 50% of a portfolio.

```
p = Portfolio;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AInequality);

     1     1     1     0     0
     0     0    -1    -1    -1

disp(p.bInequality);

    0.5000
   -0.5000
```

Add Linear Inequality Constraint to a PortfolioCVaR Object

Set a linear inequality constraint to ensure that the first three assets constitute at most 50% of a portfolio. Then add another linear inequality constraint to ensure that the last three assets constitute at least 50% of a portfolio.

```
p = PortfolioCVaR;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
```

```
p = addInequality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AInequality);

     1     1     1     0     0
     0     0    -1    -1    -1

disp(p.bInequality);

    0.5000
   -0.5000
```

Add Linear Inequality Constraint to a PortfolioMAD Object

Set a linear inequality constraint to ensure that the first three assets constitute at most 50% of a portfolio. Then add another linear inequality constraint to ensure that the last three assets constitute at least 50% of a portfolio.

```
p = PortfolioMAD;
A = [ 1 1 1 0 0 ];    % first inequality constraint
b = 0.5;
p = setInequality(p, A, b);

A = [ 0 0 -1 -1 -1 ];    % second inequality constraint
b = -0.5;
p = addInequality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AInequality);

     1     1     1     0     0
     0     0    -1    -1    -1

disp(p.bInequality);

    0.5000
   -0.5000
```

- “Working with Linear Inequality Constraints Using Portfolio Object” on page 4-89
- “Working with Linear Inequality Constraints Using PortfolioCVaR Object” on page 5-85
- “Working with Linear Inequality Constraints Using PortfolioMAD Object” on page 6-82
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`AInequality` — Linear inequality constraints formed from matrix

matrix

Linear inequality constraints, specified as a matrix.

Note An error results if `AInequality` is empty and `bInequality` is nonempty.

Data Types: `double`

`bInequality` — Linear inequality constraints formed from vector

vector

Linear inequality constraints, specified as a vector.

Note An error results if `bInequality` is empty and `AInequality` is nonempty.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to add the linear inequality constraints for portfolio weights.

  ```
  obj = obj.addInequality(AInequality, bInequality)
  ```

- You can also remove linear inequality constraints from any of the portfolio objects using dot notation.

  ```
  obj = obj.setInequality([ ], [ ])
  ```

See Also

`setInequality`

Topics

"Working with Linear Inequality Constraints Using Portfolio Object" on page 4-89
"Working with Linear Inequality Constraints Using PortfolioCVaR Object" on page 5-85
"Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-82
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

adline

Accumulation/Distribution line

Syntax

```
adln = adline(highp,lowp,closep,tvolume)

adln = adline([highp lowp closep tvolume])

adlnts = adline(tsobj)

adlnts = adline(tsobj,'ParameterName',ParameterValue, ...)
```

Arguments

`highp`	High price (vector)
`lowp`	Low price (vector)
`closep`	Closing price (vector)
`tvolume`	Volume traded (vector)
`tsobj`	Time series object

Description

`adln = adline(highp,lowp,closep,tvolume)` computes the Accumulation/Distribution line for a set of stock price and volume traded data. The prices required for this function are the high (`highp`), low (`lowp`), and closing (`closep`) prices.

`adln = adline([highp lowp closep tvolume])` accepts a four-column matrix as input. The first column contains the high prices, the second contains the low prices, the third contains the closing prices, and the fourth contains the volume traded.

`adlnts = adline(tsobj)` computes the Williams Accumulation/Distribution line for a set of stock price data contained in the financial time series object `tsobj`. The object

must contain the high, low, and closing prices plus the volume traded. The function assumes that the series are named `High`, `Low`, `Close`, and `Volume`. All are required. `adlnts` is a financial time series object with the same dates as `tsobj` but with a single series named `ADLine`.

`adlnts = adline(tsobj,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name
- `VolumeName`: volume traded series name

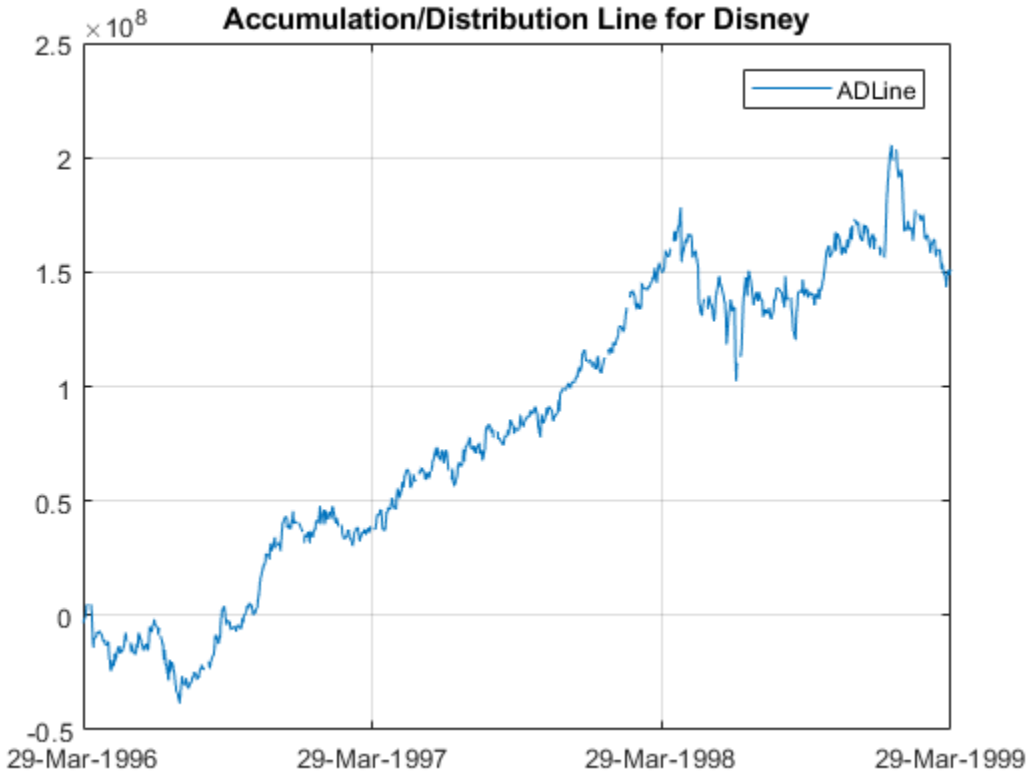
Parameter values are the character vectors that represent the valid parameter names.

Examples

Calculate the Accumulation/Distribution Line for a Stock

This example shows how to compute the Accumulation/Distribution line for Disney stock and plot the results.

```
load disney.mat
dis_ADLine = adline(dis);
plot(dis_ADLine)
title('Accumulation/Distribution Line for Disney')
```



- "Technical Analysis Examples" on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 56–58.

See Also

`adosc` | `willad` | `willpctr`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

adosc

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 one or more names for the required data series if it is different from one or more expected default names. Valid parameter names are

- `HighName` — High prices series name
- `LowName` — Low prices series name
- `OpenName` — Opening prices series name
- `CloseName` — Closing prices series name

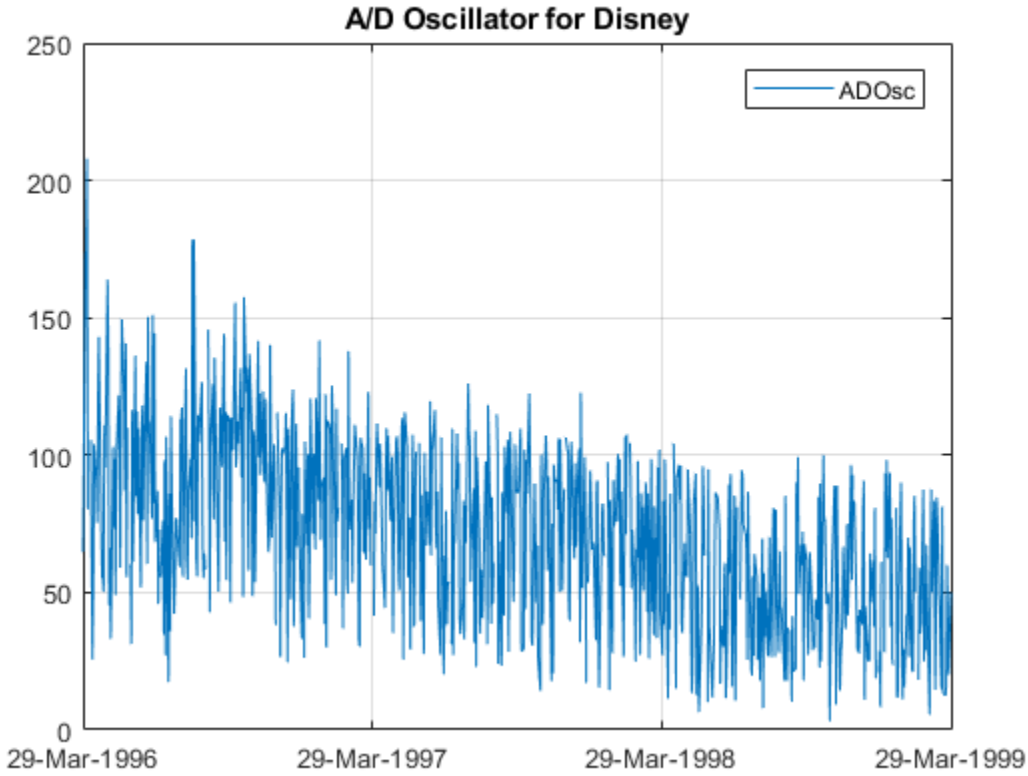
Parameter values are the character vectors that represent the valid parameter names.

Examples

Calculate the Accumulation/Distribution Oscillator for a Stock

This example shows how to find the Accumulation/Distribution oscillator for Disney stock and plot the results.

```
load disney.mat
dis_ADOsc = adosc(dis);
plot(dis_ADOsc)
title('A/D Oscillator for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

[1] Kaufman, P. J. *The New Commodity Trading Systems and Methods*. John Wiley and Sons, New York, 1987.

See Also

`adline` | `willad`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

amortize

Amortization schedule

Syntax

```
[Principal,Interest,Balance,Payment] = amortize(Rate,NumPeriods,PresentValue,FutureValu
```

Arguments

`Rate`	Interest rate per period, as a decimal fraction.
`NumPeriods`	Number of payment periods.
`PresentValue`	Present value of the loan.
`FutureValue`	(Optional) Future value of the loan. Default = `0`.
`Due`	(Optional) When payments are due: `0` = end of period (default), or `1` = beginning of period.

Description

`[Principal,Interest,Balance,Payment] = amortize(Rate,NumPeriods,PresentValue,FutureValue,Due)` returns the principal and interest payments of a loan, the remaining balance of the original loan amount, and the periodic payment.

`Principal`	Principal paid in each period. A `1`-by-`NumPeriods` vector.
`Interest`	Interest paid in each period. A `1`-by-`NumPeriods` vector.
`Balance`	Remaining balance of the loan in each payment period. A `1`-by-`NumPeriods` vector.
`Payment`	Payment per period. A scalar.

Examples

Compute an Amortization Schedule for a Conventional 30-Year, Fixed-Rate Mortgage With Fixed Monthly Payments

Compute an amortization schedule for a conventional 30-year, fixed-rate mortgage with fixed monthly payments and assume a fixed rate of 12% APR and an initial loan amount of $100,000.

```
Rate         = 0.12/12;   % 12 percent APR = 1 percent per month
NumPeriods   = 30*12;     % 30 years = 360 months
PresentValue = 100000;

[Principal, Interest, Balance, Payment] = amortize(Rate, ...
NumPeriods, PresentValue);
```

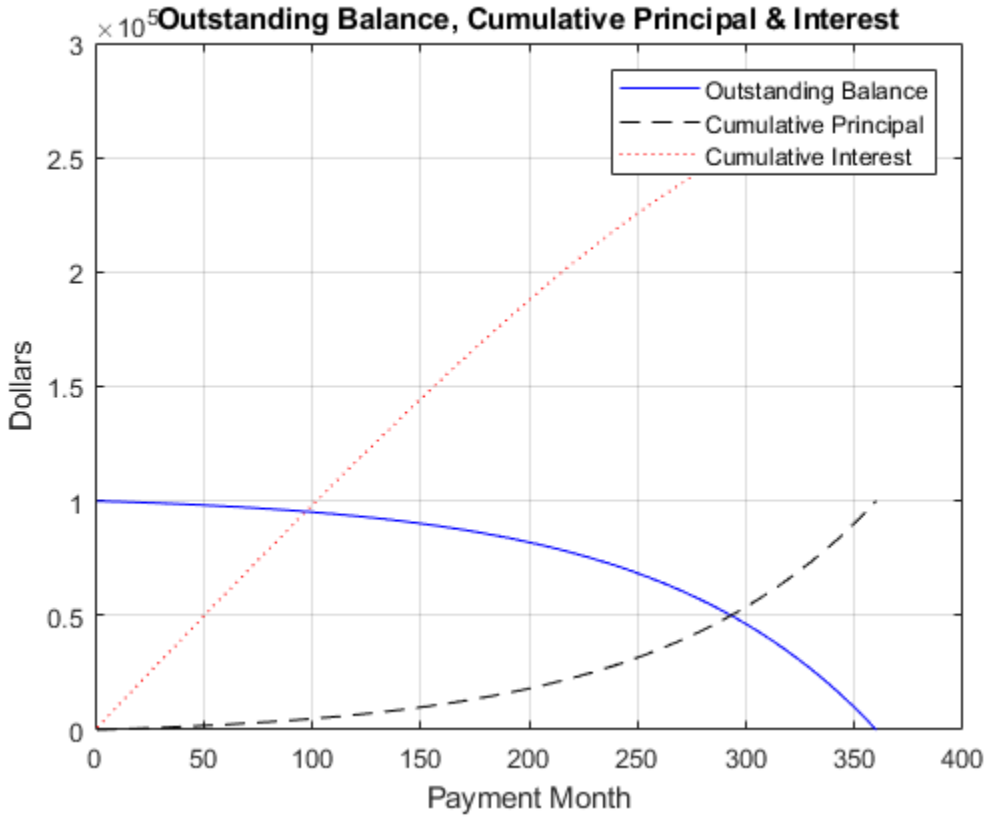
The output argument `Payment` contains the fixed monthly payment.

```
format bank

Payment
```

```
Payment =
       1028.61
```

Summarize the amortization schedule graphically by plotting the current outstanding loan balance, the cumulative principal, and the interest payments over the life of the mortgage. In particular, note that total interest paid over the life of the mortgage exceeds $270,000, far in excess of the original loan amount.

```
plot(Balance,'b'), hold('on')
plot(cumsum(Principal),'--k')
plot(cumsum(Interest),':r')

xlabel('Payment Month')
ylabel('Dollars')
grid('on')
title('Outstanding Balance, Cumulative Principal & Interest')
legend('Outstanding Balance', 'Cumulative Principal', ...
'Cumulative Interest')
```



The solid blue line represents the declining principal over the 30-year period. The dotted red line indicates the increasing cumulative interest payments. Finally, the dashed black line represents the cumulative principal payments, reaching $100,000 after 30 years.

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`annurate` | `annuterm` | `payadv` | `payodd` | `payper`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

annurate

Periodic interest rate of annuity

Syntax

```
Rate = annurate(NumPeriods,Payment,PresentValue,FutureValue,Due)
```

Arguments

`NumPeriods`	Number of payment periods.
`Payment`	Payment per period.
`PresentValue`	Present value of the loan or annuity.
`FutureValue`	(Optional) Future value of the loan or annuity. Default = `0`.
`Due`	(Optional) When payments are due: `0` = end of period (default), or `1` = beginning of period.

Description

`Rate = annurate(NumPeriods,Payment,PresentValue,FutureValue,Due)` returns the periodic interest rate paid on a loan or annuity.

Examples

Calculate the Periodic Interest Rate Paid on a Loan or Annuity

This example shows how to find the periodic interest rate of a four-year, $5000 loan with a $130 monthly payment made at the end of each month.

```
Rate = annurate(4*12, 130, 5000, 0, 0)
```

```
Rate = 0.0094
```

Rate multiplied by 12 gives an annual interest rate of 11.32% on the loan.

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`amortize` | `annuterm` | `bndyield` | `irr`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

annuterm

Number of periods to obtain value

Syntax

```
NumPeriods = annuterm(Rate,Payment,PresentValue,FutureValue,Due)
```

Arguments

`Rate`	Interest rate per period, as a decimal fraction.
`Payment`	Payment per period.
`PresentValue`	Present value.
`FutureValue`	(Optional) Future value. Default = `0`.
`Due`	(Optional) When payments are due: `0` = end of period (default), or `1` = beginning of period.

Description

`NumPeriods = annuterm(Rate,Payment,PresentValue,FutureValue,Due)` calculates the number of periods needed to obtain a future value. To calculate the number of periods needed to pay off a loan, enter the payment or the present value as a negative value.

Examples

Calculate the Number of Periods Needed to Obtain a Future Value

This example shows a savings account with a starting balance of $1500. $200 is added at the end of each month and the account pays 9% interest, compounded monthly. How many years will it take to save $5,000?

```
NumPeriods = annuterm(0.09/12, 200, 1500, 5000, 0)

NumPeriods = 15.6752
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`amortize` | `annurate` | `fvfix` | `pvfix`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

arith2geom

Arithmetic to geometric moments of asset returns

Syntax

```
[mg, Cg] = arith2geom(ma,Ca)

[mg, Cg] = arith2geom(ma,Ca,t)
```

Arguments

`ma`	Arithmetic mean of asset-return data (n-vector).
`Ca`	Arithmetic covariance of asset-return data, an `n`-by-`n` symmetric, positive-semidefinite matrix.
`t`	(Optional) Target period of geometric moments in terms of periodicity of arithmetic moments with default value `1` (scalar).

Description

`arith2geom` transforms moments associated with a simple Brownian motion into equivalent continuously compounded moments associated with a geometric Brownian motion with a possible change in periodicity.

`[mg,Cg] = arith2geom(ma,Ca,t)` returns `mg`, continuously compounded or "geometric" mean of asset returns over the target period (n-vector), and `Cg`, which is a continuously compounded or "geometric" covariance of asset returns over the target period (`n`-by-`n` matrix).

Arithmetic returns over period t_A are modeled as multivariate normal random variables with moments

$E[\mathrm{X}] = \mathrm{m}_A$

and

$$\mathrm{cov}(\mathrm{X}) = \mathrm{C}_A$$

Geometric returns over period t_G are modeled as multivariate lognormal random variables with moments

$$E[\mathrm{Y}] = 1 + \mathrm{m}_G$$

$$\mathrm{cov}(\mathrm{Y}) = \mathrm{C}_G$$

Given $t = t_G / t_A$, the transformation from geometric to arithmetic moments is

$$1 + \mathrm{m}_{G_i} = \exp(t\mathrm{m}_{A_i} + \frac{1}{2}t\mathrm{C}_{A_{ii}})$$

$$\mathrm{C}_{G_{ij}} = (1 + \mathrm{m}_{G_i})(1 + \mathrm{m}_{G_j})(\exp(t\mathrm{C}_{Aij}) - 1)$$

For $i,j = 1,..., n$.

Note If $t = 1$, then $\mathbf{Y} = \exp(\mathbf{X})$.

This function has no restriction on the input mean `ma` but requires the input covariance `Ca` to be a symmetric positive-semidefinite matrix.

The functions `arith2geom` and `geom2arith` are complementary so that, given `m`, `C`, and `t`, the sequence

```
[mg,Cg] = arith2geom(m,C,t);
[ma,Ca] = geom2arith(mg,Cg,1/t);
```

yields `ma` = `m` and `Ca` = `C`.

Examples

Example 1. Given arithmetic mean `m` and covariance `C` of monthly total returns, obtain annual geometric mean `mg` and covariance `Cg`. In this case, the output period (1 year) is 12 times the input period (1 month) so that `t = 12` with

```
[mg, Cg] = arith2geom(m, C, 12);
```

Example 2. Given annual arithmetic mean `m` and covariance `C` of asset returns, obtain monthly geometric mean `mg` and covariance `Cg`. In this case, the output period (1 month) is 1/12 times the input period (1 year) so that `t = 1/12` with

```
[mg, Cg] = arith2geom(m, C, 1/12);
```

Example 3. Given arithmetic means `m` and standard deviations `s` of daily total returns (derived from 260 business days per year), obtain annualized continuously compounded mean `mg` and standard deviations `sg` with

```
[mg, Cg] = arith2geom(m, diag(s .^2), 260);
sg = sqrt(diag(Cg));
```

Example 4. Given arithmetic mean `m` and covariance `C` of monthly total returns, obtain quarterly continuously compounded return moments. In this case, the output is 3 of the input periods so that `t = 3` with

```
[mg, Cg] = arith2geom(m, C, 3);
```

Example 5. Given arithmetic mean `m` and covariance `C` of 1254 observations of daily total returns over a 5-year period, obtain annualized continuously compounded return moments. Since the periodicity of the arithmetic data is based on 1254 observations for a 5-year period, a 1-year period for geometric returns implies a target period of `t = 1254/5` so that

```
[mg, Cg] = arith2geom(m, C, 1254/5);
```

See Also

`geom2arith`

Topics

"Data Transformation and Frequency Conversion" on page 12-12

Introduced before R2006a

ascii2fts

Create financial time series object from ASCII file

Syntax

```
tsobj = ascii2fts(filename,descrow,colheadrow,skiprows)

tsobj = ascii2fts(filename,timedata,descrow,colheadrow,skiprows)
```

Arguments

`filename`	ASCII data file
`descrow`	(Optional) Row number in the data file that contains the description to be used for the description field of the financial time series object
`colheadrow`	(Optional) Row number that has the column headers/names
`skiprows`	(Optional) Scalar or vector of row numbers to be skipped in the data file
`timedata`	Set to `'T'` if time-of-day data is present in the ASCII data file or to `'NT'` if no time-of-day data is present.

Description

`tsobj = ascii2fts(filename,descrow,colheadrow,skiprows)` creates a financial time series object `tsobj` from the ASCII file named `filename`. This form of the function can only read a data file without time-of-day information and create a financial time series object without time information. If time information is present in the ASCII file, an error message appears.

The general format of the text data file is

- Can contain header text lines.

- Can contain column header information. The column header information must immediately precede the data series columns unless `skiprows` is specified.
- Leftmost column must be the date column.
- Dates must be in a valid character vector date 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'`
- Time information must be in `'hh:mm'` format.
- Each column must be separated either by spaces or a tab.

`tsobj = ascii2fts(filename, timedata, descrow, colheadrow, skiprows)` creates a financial time series object containing time-of-day data. Set `timedata` to `'T'` to create a financial time series object containing time-of-day data. The ascii time information must be in `'hh:mm'` format for `ascii2fts`.

Examples

Example 1. If your data file contains no description or column header rows,

```
1/3/95   36.75   36.9063   36.6563   36.875    1167900
1/4/95   37      37.2813   36.625    37.1563   1994700  ...
```

you can create a financial time series object from it with the simplest form of the `ascii2fts` function:

```
myinc = ascii2fts('my_inc.dat');

myinc =

desc:  my_inc.dat
freq:  Unknown (0)

'dates:  (2)'  'series1: (2)'  'series2: (2)'   'series3: (2)'...
'03-Jan-1995'  [   36.7500]    [    36.9063]    [   36.6563]
'04-Jan-1995'  [        37]    [    37.2813]    [   36.6250]
```

Example 2: If your data file contains description and column header information with the data series immediately following the column header row,

```
International Business Machines Corporation (IBM)
Daily prices (1/3/95 to 4/5/99)
DATE     OPEN    HIGH      LOW       CLOSE     VOLUME
1/3/95   36.75   36.9063   36.6563   36.875    1167900
1/4/95   37      37.2813   36.625    37.1563   1994700  ...
```

you must specify the row numbers containing the description and column headers:

```
ibm = ascii2fts('ibm9599.dat', 1, 3);

ibm =

desc:  International Business Machines Corporation (IBM)
freq:  Unknown (0)
'dates:  (2)'  'OPEN:  (2)'    'HIGH:  (2)'    'LOW:  (2)' ...
'03-Jan-1995'  [  36.7500]    [   36.9063]    [  36.6563]
'04-Jan-1995'  [       37]    [   37.2813]    [  36.6250]
```

Example 3: If your data file contains rows between the column headers and the data series, for example,

```
Staples, Inc. (SPLS)
Daily prices
DATE     OPEN    HIGH     LOW      CLOSE    VOLUME
Starting date: 04/08/1996
Ending date:   04/07/1999
4/8/96   19.50   19.75    19.25    19.375   548500
4/9/96   19.75   20.125   19.375   20       1135900  ...
```

you need to indicate to `ascii2fts` the rows in the file that must be skipped. Assume that you have called the data file containing the Staples data (`staples.dat`).

```
spls = ascii2fts('staples.dat', 1, 3, [4 5]);
```

The command above 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`

Topics

“Creating a Financial Time Series Object” on page 13-11
“Working with Financial Time Series Objects” on page 12-3

Introduced before R2006a

bar, barh

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, barh` draw vertical and horizontal bar charts.

`bar(tsobj)` draws the columns of data series of the object `tsobj`. The number of data series dictates the number of vertical bars per group. Each group is the data for one particular date.

`bar(tsobj,width)` specifies the width of the bars.

`bar(..., 'style')` changes the style of the bar chart.

`hbar = bar(...)` returns a vector of bar handles.

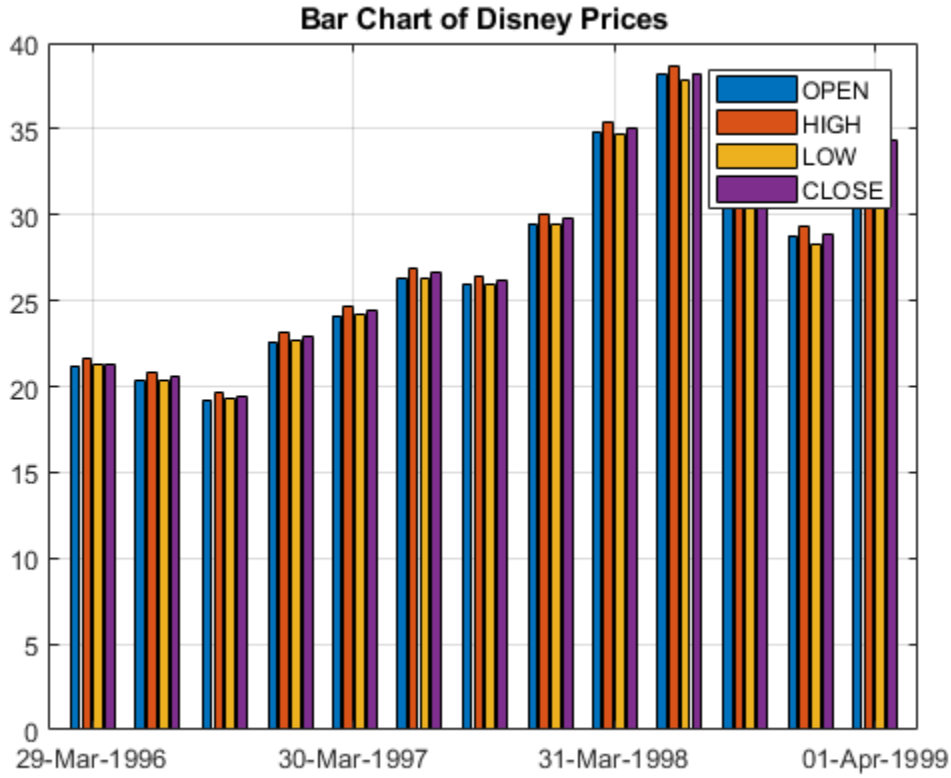
Use the MATLAB command `shading faceted` to put edges on the bars. Use `shading flat` to turn edges off.

Examples

Create a Bar Chart for a Stock

This example shows how to create a bar chart for Disney stock showing high, low, opening, and closing prices.

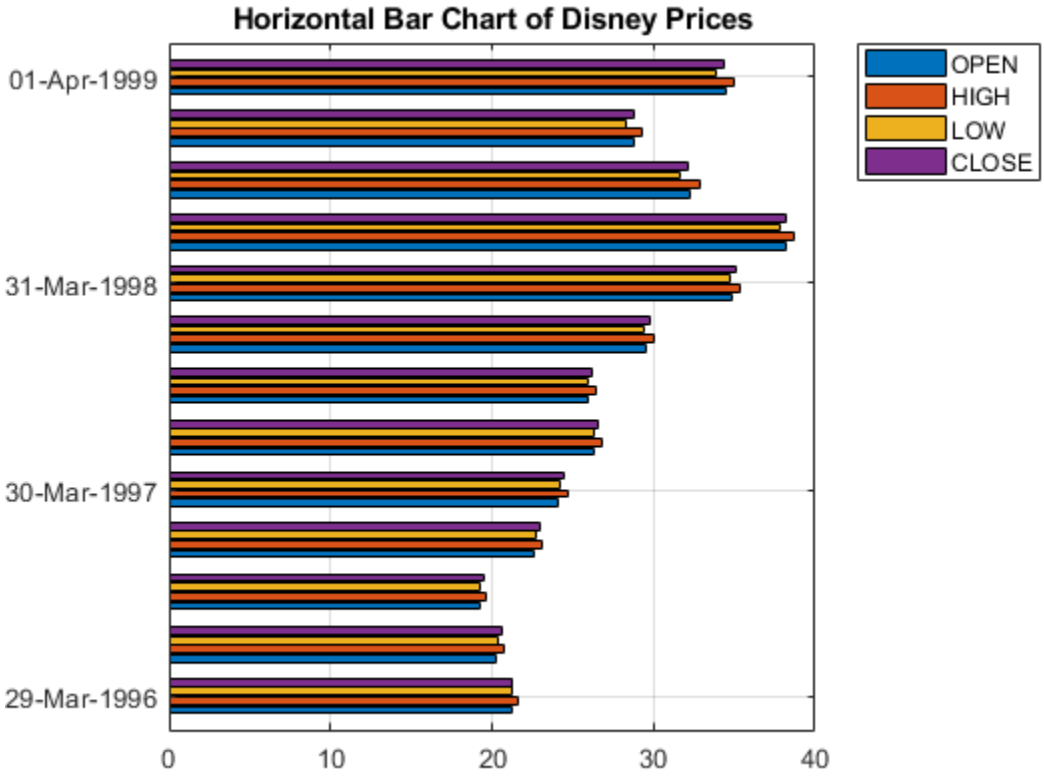
```
load disney
bar(q_dis)
title('Bar Chart of Disney Prices')
```



Create a Horizontal Bar Chart for a Stock

This example shows how to create a horizontal bar chart for Disney stock showing high, low, opening, and closing prices.

```
load disney
barh(q_dis)
title('Horizontal Bar Chart of Disney Prices')
```



- “Charting Financial Data” on page 2-14

See Also

`bar3`, `bar3h` | `candle` | `highlow`

Topics

“Charting Financial Data” on page 2-14

Introduced before R2006a

bar3, bar3h

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, bar3h` draw three-dimensional vertical and horizontal bar charts.

`bar3(tsobj)` draws the columns of data series of the object `tsobj`. The number of data series dictates the number of vertical bars per group. Each group is the data for one particular date.

`bar3(tsobj,width)` specifies the width of the bars.

`bar3(..., 'style')` changes the style of the bar chart.

`hbar3 = bar3(...)` returns a vector of bar handles.

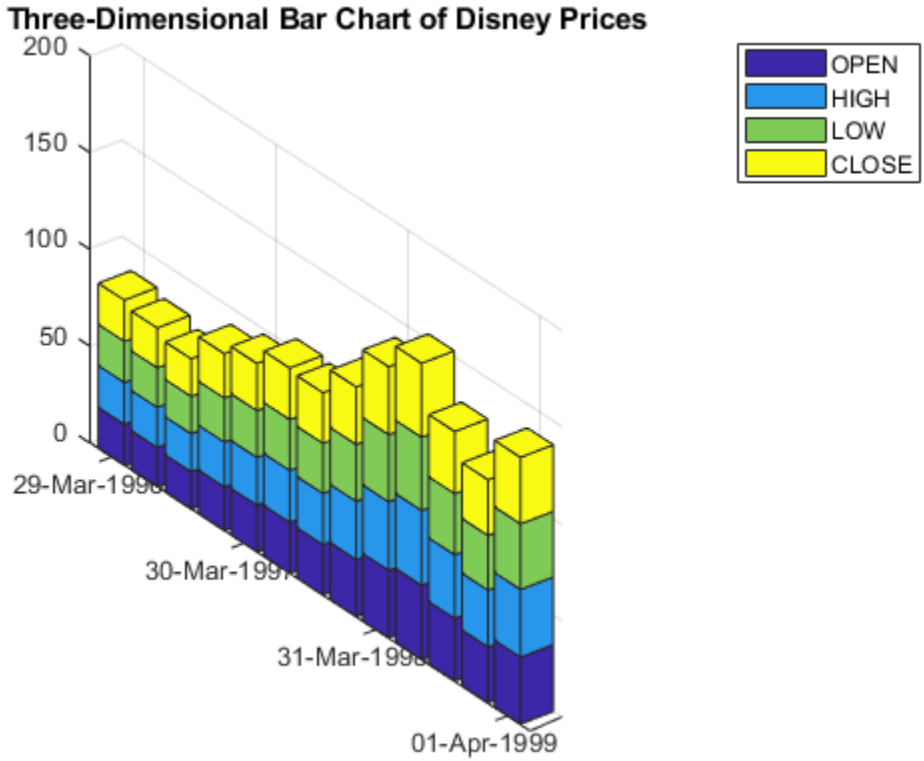
Use the MATLAB command `shading faceted` to put edges on the bars. Use `shading flat` to turn edges off.

Examples

Create a Three-Dimensional Bar Chart

This example shows how to create a three-dimensional bar chart for Disney stock showing high, low, opening, and closing prices.

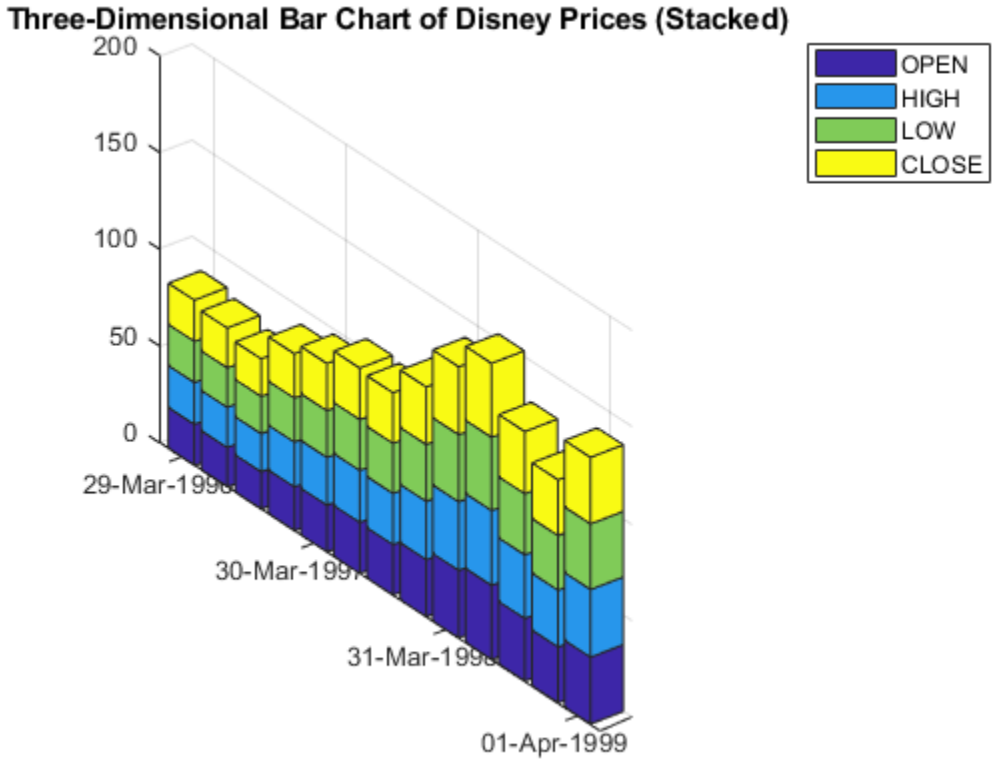
```
load disney
bar3(q_dis, 'stacked')
title('Three-Dimensional Bar Chart of Disney Prices')
```



Create a Three-Dimensional (Stacked) Bar Chart

This example shows how to create a three-dimensional, stacked bar chart for Disney stock showing high, low, opening, and closing prices.

```
load disney
bar3(q_dis, 'stacked')
title('Three-Dimensional Bar Chart of Disney Prices (Stacked)')
```



- “Charting Financial Data” on page 2-14

See Also

`bar`, `barh` | `candle` | `highlow`

Topics

“Charting Financial Data” on page 2-14

Introduced before R2006a

beytbill

Bond equivalent yield for Treasury bill

Syntax

```
Yield = beytbill(Settle,Maturity,Discount)
```

Description

`Yield = beytbill(Settle,Maturity,Discount)` returns the bond equivalent yield for a Treasury bill.

Examples

Find the Bond Equivalent Yield for a Treasury Bill

This example shows how to find the bond equivalent yield for a Treasury bill that has a settlement date of February 11, 2000, a maturity date of August 7, 2000, and a discount rate is 5.77.

```
Yield = beytbill('2/11/2000', '8/7/2000', 0.0577)
```

```
Yield = 0.0602
```

Find the Bond Equivalent Yield for a Treasury Bill Using datetime Inputs

This example shows how to use `datetime` inputs to find the bond equivalent yield for a Treasury bill that has a settlement date of February 11, 2000, a maturity date of August 7, 2000, and the discount rate is 5.77.

```
Yield = beytbill(datetime('11-Feb-2000','Locale','en_US'), datetime('7-Aug-2000','Local
0.0577)
```

```
Yield = 0.0602
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`Settle` — Settlement date of Treasury bill

serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill

serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Discount` — Discount rate of Treasury bill

decimal

Discount rate of the Treasury bill, specified as a scalar of a `NTBILLS`-by-`1` vector of decimal fraction values.

Data Types: `double`

Output Arguments

`Yield` — Treasury bill yield

decimal

Treasury bill yield, returned as a scalar or `NTBILLS`-by-`1` vector.

Note The number of days to maturity is typically quoted as: md - sd - 1. A `NaN` is returned for all cases in which negative prices are implied by the discount rate, `Discount`, and the number of days between `Settle` and `Maturity`.

See Also

`datenum` | `datetime` | `prtbill` | `yldtbill`

Topics

"Computing Treasury Bill Price and Yield" on page 2-41
"Treasury Bills Defined" on page 2-40

Introduced before R2006a

binprice

Binomial put and call American option pricing using Cox-Ross-Rubinstein model

Syntax

```
[AssetPrice,OptionValue] = binprice(Price,Strike,Rate,Time,
Increment,Volatility,Flag)
[AssetPrice,OptionValue] = binprice(___,DividendRate,Dividend,ExDiv)
```

Description

`[AssetPrice,OptionValue] = binprice(Price,Strike,Rate,Time, Increment,Volatility,Flag)` prices an American option using the Cox-Ross-Rubinstein binomial pricing model.

`[AssetPrice,OptionValue] = binprice(___,DividendRate,Dividend,ExDiv)` adds optional arguments for `DividendRate`,`Dividend`, and `ExDiv`.

Examples

Price an American Option Using the Cox-Ross-Rubinstein Binomial Pricing Model

This example shows how to price an American put option with an exercise price of $50 that matures in 5 months. The current asset price is $52, the risk-free interest rate is 10%, and the volatility is 40%. There is one dividend payment of $2.06 in 3-1/2 months. When specifying the input argument `ExDiv` in terms of number of periods, divide the ex-dividend date, specified in years, by the time `Increment`.

ExDiv = (3.5/12) / (1/12) = 3.5

```
[Price, Option] = binprice(52, 50, 0.1, 5/12, 1/12, 0.4, 0, 0, 2.06, 3.5)

Price =
```

```
   52.0000   58.1367   65.0226   72.7494   79.3515   89.0642
         0   46.5642   52.0336   58.1706   62.9882   70.6980
         0         0   41.7231   46.5981   49.9992   56.1192
         0         0         0   37.4120   39.6887   44.5467
         0         0         0         0   31.5044   35.3606
         0         0         0         0         0   28.0688


Option =

    4.4404    2.1627    0.6361         0         0         0
         0    6.8611    3.7715    1.3018         0         0
         0         0   10.1591    6.3785    2.6645         0
         0         0         0   14.2245   10.3113    5.4533
         0         0         0         0   18.4956   14.6394
         0         0         0         0         0   21.9312
```

The output returned is the asset price and American option value at each node of the binary tree.

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset

numeric

Current price of underlying asset, specified as a scalar numeric value.

Data Types: `double`

`Strike` — Exercise price of the option

numeric

Exercise price of the option, specified as a scalar numeric value.

Data Types: `double`

`Rate` — Risk-free interest rate

decimal

Risk-free interest rate, specified as scalar decimal fraction.

Data Types: `double`

`Time` — Option time until maturity

numeric

Option time until maturity, specified as a scalar for the number of years.

Data Types: `double`

`Increment` — Time increment

numeric

Time increment, specified as a scalar numeric. `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.)

Data Types: `double`

`Volatility` — Asset volatility

numeric

Asset volatility, specified as a scalar numeric.

Data Types: `double`

`Flag` — Flag indicating whether option is a call or put

integer with values `0` or `1`

Flag indicating whether option is a call or put, specified as a scalar `Flag = 1` for a call option, or `Flag = 0` for a put option.

Data Types: `logical`

`DividendRate` — Dividend rate

`0` (default) | decimal

(Optional) Dividend rate, specified as a scalar decimal. 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`

Data Types: `double`

`Dividend` — Dividend payment
`0` (default) | numeric

(Optional) Dividend payment at an ex-dividend date (`ExDiv`), specified as a `1`-by-`N` row vector. For each dividend payment, there must be a corresponding ex-dividend date. If you enter values for `Dividend` and `ExDiv`, set `DividendRate` = `0`.

Data Types: `double`

`ExDiv` — Ex-dividend date
`0` (default) | numeric

(Optional) Ex-dividend date, specified as a `1`-by-`N` vector row vector for the number of periods.

Data Types: `double`

Output Arguments

`AssetPrice` — Asset price
vector

Asset price, returned as a vector that represents each node of the Cox-Ross-Rubinstein (CRR) binary tree.

`OptionValue` — Option value
vector

Option value, returned as a vector that represents each node of the Cox-Ross-Rubinstein (CRR) binary tree.

References

[1] Cox, J., S. Ross, and M. Rubenstein. "Option Pricing: A Simplified Approach." *Journal of Financial Economics*. Vol. 7, Sept. 1979, pp. 229–263.

[2] Hull, John C. *Options, Futures, and Other Derivative Securities. 2nd edition*, Chapter 14.

See Also

`blkprice` | `blsprice`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blkimpv

Implied volatility for futures options from Black model

Syntax

```
Volatility = blkipmv(Price,Strike,Rate,Time,Value)
Volatility = blkimpv(___,Limit,Tolerance,Class)
```

Description

`Volatility = blkipmv(Price,Strike,Rate,Time,Value)` computes the implied volatility of a futures price from the market value of European futures options using Black's model.

Note Any input argument can be a scalar, vector, or matrix. When a value is a scalar, that value is used to compute the implied volatility of all the options. If more than one input is a vector or matrix, the dimensions of all nonscalar inputs must be identical.

Ensure that `Rate` and `Time` are expressed in consistent units of time.

`Volatility = blkimpv(___,Limit,Tolerance,Class)` adds optional arguments for `Limit`, `Tolerance`, and `Class`.

Examples

Find Implied Volatility for Futures Options from Black's Model

This example shows how to find the implied volatility for a European call futures option that expires in four months, trades at $1.1166, and has an exercise price of $20. Assume that the current underlying futures price is also $20 and that the risk-free rate is 9% per annum. Furthermore, assume that you are interested in implied volatilities no greater

than 0.5 (50% per annum). Under these conditions, the following commands all return an implied volatility of 0.25, or 25% per annum.

```
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5);
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5, [], {'Call'});
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5, [], true)

Volatility = 0.2500
```

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset
numeric

Current price of the underlying asset (that is, a futures contract), specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of the futures option
numeric

Exercise price of the futures option, specified as a numeric value.

Data Types: `double`

`Rate` — Annualized continuously compounded risk-free rate of return over life of the option
positive decimal

Annualized continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal number.

Data Types: `double`

`Time` — Time to expiration of the futures option
numeric

Time to expiration of the futures option, specified as the number of years.

Data Types: `double`

`Value` — Price of a European option from which implied volatility of underlying asset is derived
numeric

Price of a European futures option from which the implied volatility of the underlying asset is derived, specified as a numeric.

Data Types: `double`

`Limit` — Upper bound of implied volatility search interval
`10` (1000% per annum) (default) | positive scalar numeric

(Optional) Upper bound of the implied volatility search interval, specified as a positive scalar numeric. If `Limit` is empty or unspecified, the default is `10`, or 1000% per annum.

Data Types: `double`

`Tolerance` — Implied volatility termination tolerance
`1e-6` (default) | positive scalar numeric

(Optional) Implied volatility termination tolerance, specified as a positive scalar numeric. If empty or missing, the default is `1e-6`.

Data Types: `double`

`Class` — Option class from which implied volatility is derived
`true` (call option) (default) | logical | cell array of character vectors

(Optional) Option class indicating option type (call or put) from which implied volatility is derived, specified as a logical indicator or a cell array of character vectors.

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.

Data Types: `logical` | `cell`

Output Arguments

`Volatility` — Implied volatility of underlying asset derived from European futures option prices
decimal

Implied volatility of the underlying asset derived from European futures option prices, returned as a decimal number. If no solution is found, `blkimpv` returns `NaN`.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, , 2003, pp. 287–288.

[2] Black, Fischer. "The Pricing of Commodity Contracts." *Journal of Financial Economics*. March 3, 1976, pp. 167–79.

See Also

`blkprice` | `blsimpv` | `blsprice`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blkprice

Black model for pricing futures options

Syntax

```
[Call,Put] = blkprice(Price,Strike,Rate,Time,Volatility)
```

Description

`[Call,Put] = blkprice(Price,Strike,Rate,Time,Volatility)` computes European put and call futures option prices using Black's model.

Note Any input argument can be a scalar, vector, or matrix. If a scalar, then that value is used to price all options. If more than one input is a vector or matrix, then the dimensions of those non-scalar inputs must be the same.

Ensure that `Rate`, `Time`, and `Volatility` are expressed in consistent units of time.

Examples

Compute European Put and Call Futures Option Prices Using Black's Model

This example shows how to price European futures options with exercise prices of $20 that expire in four months. Assume that the current underlying futures price is also $20 with a volatility of 25% per annum. The risk-free rate is 9% per annum.

```
 [Call, Put] = blkprice(20, 20, 0.09, 4/12, 0.25)
```

```
Call = 1.1166
```

```
Put = 1.1166
```

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- “Greek-Neutral Portfolios of European Stock Options” on page 10-19
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Input Arguments

`Price` — Current price of underlying asset
numeric

Current price of the underlying asset (that is, a futures contract), specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of the futures option
numeric

Exercise price of the futures option, specified as a numeric value.

Data Types: `double`

`Rate` — Annualized continuously compounded risk-free rate of return over life of the option
positive decimal

Annualized continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal number.

Data Types: `double`

`Time` — Time to expiration of option
numeric

Time to expiration of the option, specified as the number of years. `Time` must be greater than `0`.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized futures price volatility, specified as a positive decimal number.

Data Types: `double`

Output Arguments

`Call` — Price of a European call futures option

matrix

Price of a European call futures option, returned as a matrix.

`Put` — Price of a European put futures option

matrix

Price of a European put futures option, returned as a matrix.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, , 2003, pp. 287–288.

[2] Black, Fischer. "The Pricing of Commodity Contracts." *Journal of Financial Economics*. March 3, 1976, pp. 167–79.

See Also

`binprice` | `blsprice`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blsdelta

Black-Scholes sensitivity to underlying price change

Syntax

```
[CallDelta,PutDelta] = blsdelta(Price,Strike,Rate,Time,Volatility)
[CallDelta,PutDelta] = blsdelta(___,Yield)
```

Description

`[CallDelta,PutDelta] = blsdelta(Price,Strike,Rate,Time,Volatility)` returns delta, the sensitivity in option value to change in the underlying asset price. Delta is also known as the hedge ratio. `blsdelta` uses `normcdf`, the normal cumulative distribution function in the Statistics and Machine Learning Toolbox.

Note `blsdelta` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

`[CallDelta,PutDelta] = blsdelta(___,Yield)` adds an optional argument for `Yield`.

Examples

Find the Sensitivity in Option Value to Change in the Underlying Asset Price

This example shows how to find the Black-Scholes delta sensitivity for an underlying asset price change.

```
[CallDelta, PutDelta] = blsdelta(50, 50, 0.1, 0.25, 0.3, 0)
```

```
CallDelta = 0.5955
```

```
PutDelta = -0.4045
```

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset

numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of option

numeric

Exercise price of the option, specified as a numeric valuie.

Data Types: `double`

`Rate` — Annualized, continuously compounded risk-free rate of return over life of option

positive decimal

Annualized, continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal value.

Data Types: `double`

`Time` — Time (in years) to expiration of the option
numeric

Time (in years) to expiration of the option, specified as a numeric value.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), specified as a positive decimal value.

Data Types: `double`

`Yield` — Annualized, continuously compounded yield of the underlying asset over life of option
`0` (default) | decimal

(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, specified as a decimal value. 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.

Data Types: `double`

Output Arguments

`CallDelta` — Delta of call option
numeric

Delta of the call option, returned as a numeric value.

`PutDelta` — Delta of put option
numeric

Delta of the put option, returned as a numeric.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, , 2003.

See Also

`blsgamma` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blsgamma

Black-Scholes sensitivity to underlying delta change

Syntax

```
Gamma = blsgamma(Price,Strike,Rate,Time,Volatility)
Gamma = blsgamma(___,Yield)
```

Description

`Gamma = blsgamma(Price,Strike,Rate,Time,Volatility)` returns gamma, the sensitivity of delta to change in the underlying asset price. `blsgamma` uses `normpdf`, the probability density function in the Statistics and Machine Learning Toolbox.

Note `blsgamma` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

`Gamma = blsgamma(___,Yield)` adds an optional argument for `Yield`.

Examples

Find Gamma for a Change in the Underlying Asset Price

This example shows how to find the gamma, the sensitivity of delta to a change in the underlying asset price.

```
Gamma = blsgamma(50, 50, 0.12, 0.25, 0.3, 0)
```

```
Gamma = 0.0512
```

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset
numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of option
numeric

Exercise price of the option, specified as a numeric valuie.

Data Types: `double`

`Rate` — Annualized, continuously compounded risk-free rate of return over life of option
positive decimal

Annualized, continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal value.

Data Types: `double`

`Time` — Time (in years) to expiration of the option
numeric

Time (in years) to expiration of the option, specified as a numeric value.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), specified as a positive decimal value.

Data Types: `double`

`Yield` — Annualized, continuously compounded yield of the underlying asset over life of option
`0` (default) | decimal

(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, specified as a decimal value. 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.

Data Types: `double`

Output Arguments

`Gamma` — Delta to change in underlying security price
numeric

Delta to change in underlying security price, returned as a numeric value.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, , 2003.

See Also

`blsdelta` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blsimpv

Black-Scholes implied volatility

Syntax

```
Volatility = blsimpv(Price,Strike,Rate,Time,Value)
Volatility = blsimpv(___,Limit,Yield,Tolerance,Class)
```

Description

`Volatility = blsimpv(Price,Strike,Rate,Time,Value)` using a Black-Scholes model computes the implied volatility of an underlying asset from the market value of European call and put options.

Note The input arguments `Price`, `Strike`, `Rate`, `Time`, `Value`, `Yield`, and `Class` can be scalars, vectors, or matrices. If scalars, then that value is used to compute the implied volatility from all options. If more than one of these inputs is a vector or matrix, then the dimensions of all non-scalar inputs must be the same.

Also, ensure that `Rate`, `Time`, and `Yield` are expressed in consistent units of time.

`Volatility = blsimpv(___,Limit,Yield,Tolerance,Class)` adds optional arguments for `Limit`, `Yield`,`Tolerance`, and `Class`.

Examples

Compute the Implied Volatility of an Underlying Asset Using a Black-Scholes Model

This example shows how to compute the implied volatility for a European call option trading at $10 with an exercise price of $95 and three months until expiration. Assume that the underlying stock pays no dividend and trades at $100. The risk-free rate is 7.5%

per annum. Furthermore, assume that you are interested in implied volatilities no greater than 0.5 (50% per annum). Under these conditions, the following statements all compute an implied volatility of 0.3130, or 31.30% per annum.

```
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5);
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5, 0, [], {'Call'});
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5, 0, [], true)

Volatility = 0.3130
```


- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset

numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of the option

numeric

Exercise price of the option, specified as a numeric value.

Data Types: `double`

`Rate` — Annualized continuously compounded risk-free rate of return over life of the option

positive decimal

Annualized continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal number.

Data Types: `double`

`Time` — Time to expiration of option

numeric

Time to expiration of the option, specified as the number of years.

Data Types: `double`

`Value` — Price of a European option from which implied volatility of underlying asset is derived

numeric

Price of a European option from which the implied volatility of the underlying asset is derived, specified as a numeric.

Data Types: `double`

`Limit` — Upper bound of implied volatility search interval

`10` (1000% per annum) (default) | positive scalar numeric

(Optional) Upper bound of the implied volatility search interval, specified as a positive scalar numeric. If `Limit` is empty or unspecified, the default is `10`, or 1000% per annum.

Data Types: `double`

`Yield` — Annualized continuously compounded yield of underlying asset over life of the option

`0` (default) | decimal

(Optional) Annualized continuously compounded yield of the underlying asset over the life of the option, specified as a decimal number. If `Yield` is empty or missing, the default value is `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.

Note `blsimpv` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

Data Types: `double`

`Tolerance` — Implied volatility termination tolerance
`1e-6` (default) | positive scalar numeric

(Optional) Implied volatility termination tolerance, specified as a positive scalar numeric. If empty or missing, the default is `1e-6`.

Data Types: `double`

`Class` — Option class from which implied volatility is derived
`true` (call option) (default) | logical | cell array of character vectors

(Optional) Option class indicating option type (call or put) from which implied volatility is derived, specified as a logical indicator or a cell array of character vectors.

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.

Data Types: `logical` | `cell`

Output Arguments

`Volatility` — Implied volatility of underlying asset derived from European option prices
decimal

Implied volatility of the underlying asset derived from European option prices, returned as a decimal number. If no solution is found, `blsimpv` returns `NaN`.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003.

[2] Luenberger, David G. *Investment Science.* Oxford University Press, 1998.

See Also

`blsdelta` | `blsgamma` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blslambda

Black-Scholes elasticity

Syntax

```
[CallEl,PutEl] = blslambda(Price,Strike,Rate,Time,Volatility)
[CallEl,PutEl] = blslambda(___,Yield)
```

Description

`[CallEl,PutEl] = blslambda(Price,Strike,Rate,Time,Volatility)` 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 1 percent change in the underlying asset price. `blslambda` uses `normcdf`, the normal cumulative distribution function in the Statistics and Machine Learning Toolbox.

Note `blslambda` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

`[CallEl,PutEl] = blslambda(___,Yield)` adds an optional argument for `Yield`.

Examples

Find the Black-Scholes Elasticity (Lambda) for an Option

This example shows how to find the Black-Scholes elasticity, or leverage, of an option position.

```
[CallEl, PutEl] = blslambda(50, 50, 0.12, 0.25, 0.3)
```

```
CallEl = 8.1274
```

```
PutEl = -8.6466
```

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset

numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of option

numeric

Exercise price of the option, specified as a numeric valuie.

Data Types: `double`

`Rate` — Annualized, continuously compounded risk-free rate of return over life of option

positive decimal

Annualized, continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal value.

Data Types: `double`

`Time` — Time (in years) to expiration of the option
numeric

Time (in years) to expiration of the option, specified as a numeric value.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), specified as a positive decimal value.

Data Types: `double`

`Yield` — Annualized, continuously compounded yield of the underlying asset over life of option
`0` (default) | decimal

(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, specified as a decimal value. 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.

Data Types: `double`

Output Arguments

`CallEl` — Call option elasticity
numeric

Call option elasticity or leverage factor, returned as a numeric value.

`PutEl` — Put option elasticity
numeric

Put option elasticity or leverage factor, returned as a numeric value.

References

[1] Daigler, R. *Advanced Options Trading*. McGraw-Hill, 1993.

See Also

`blsdelta` | `blsgamma` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blsprice

Black-Scholes put and call option pricing

Syntax

```
[Call,Put] = blsprice(Price,Strike,Rate,Time,Volatility)
[Call,Put] = blsprice(___,Yield)
```

Description

`[Call,Put] = blsprice(Price,Strike,Rate,Time,Volatility)` computes European put and call option prices using a Black-Scholes model.

Note Any input argument can be a scalar, vector, or matrix. If a scalar, then that value is used to price all options. If more than one input is a vector or matrix, then the dimensions of those non-scalar inputs must be the same.

Ensure that `Rate`, `Time`, `Volatility`, and `Yield` are expressed in consistent units of time.

`[Call,Put] = blsprice(___,Yield)` adds an optional an argument for `Yield`.

Examples

Compute European Put and Call Option Prices Using a Black-Scholes Model

This example shows how to price European stock options that expire in three months with an exercise price of $95. Assume that the underlying stock pays no dividend, trades at $100, and has a volatility of 50% per annum. The risk-free rate is 10% per annum.

```
[Call, Put] = blsprice(100, 95, 0.1, 0.25, 0.5)
```

```
Call = 13.6953

Put = 6.3497
```

Compute European Put and Call Option Prices on a Stock Index Using a Black-Scholes Model

The S&P 100 index is at 910 and has a volatility of 25% per annum. The risk-free rate of interest is 2% per annum and the index provides a dividend yield of 2.5% per annum. Calculate the value of a three-month European call and put with a strike price of 980.

```
 [Call,Put] = blsprice(910,980,.02,.25,.25,.025)

Call = 19.6863

Put = 90.4683
```

Price a European Call Option with the Garman-Kohlhagen Model

Price an FX option on buying GBP with USD.

```
S = 1.6;  % spot exchange rate
X = 1.6;  % strike
T = .3333;
r_d = .08;  % USD interest rate
r_f = .11;  % GBP interest rate
sigma = .2;

Price = blsprice(S,X,r_d,T,sigma,r_f)

Price = 0.0639
```

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- “Greek-Neutral Portfolios of European Stock Options” on page 10-19
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Input Arguments

`Price` — Current price of underlying asset
numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of the option
numeric

Exercise price of the option, specified as a numeric value.

Data Types: `double`

`Rate` — Annualized continuously compounded risk-free rate of return over life of the option
positive decimal

Annualized continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal number.

Data Types: `double`

`Time` — Time to expiration of option
numeric

Time to expiration of the option, specified as the number of years.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (that is, annualized standard deviation of the continuously compounded asset return), specified as a positive decimal number.

Data Types: `double`

`Yield` — Annualized continuously compounded yield of underlying asset over life of the option
`0` (default) | decimal

(Optional) Annualized continuously compounded yield of the underlying asset over the life of the option, specified as a decimal number. If `Yield` is empty or missing, the default value is `0`.

For example, `Yield` could represent the dividend yield (annual dividend rate expressed as a percentage of the price of the security) or foreign risk-free interest rate for options written on stock indices and currencies.

Note `blsprice` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

Data Types: `double`

Output Arguments

`Call` — Price of a European call option

matrix

Price of a European call option, returned as a matrix.

`Put` — Price of a European put option

matrix

Price of a European put option, returned as a matrix.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003.

[2] Luenberger, David G. *Investment Science.* Oxford University Press, 1998.

See Also

`blkprice` | `blsdelta` | `blsgamma` | `blsimpv` | `blslambda` | `blsrho` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blsrho

Black-Scholes sensitivity to interest-rate change

Syntax

```
[CallRho,PutRho] = blsrho(Price,Strike,Rate,Time,Volatility)
[CallRho,PutRho] = blsrho(___,Yield)
```

Description

`[CallRho,PutRho] = blsrho(Price,Strike,Rate,Time,Volatility)` returns the call option rho `CallRho`, and the put option rho `PutRho`. Rho is the rate of change in value of derivative securities with respect to interest rates. `blsrho` uses `normcdf`, the normal cumulative distribution function in the Statistics and Machine Learning Toolbox.

Note `blsrho` can also handle an underlying asset such as currencies. When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

`[CallRho,PutRho] = blsrho(___,Yield)` adds an optional argument for `Yield`.

Examples

Find the Black-Scholes Sensitivity (Rho) to Interest-Rate Change

This example shows how to find the Black-Scholes sensitivity, rho, to interest-rate change.

```
[CallRho, PutRho] = blsrho(50, 50, 0.12, 0.25, 0.3, 0)

CallRho = 6.6686

PutRho = -5.4619
```

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- “Greek-Neutral Portfolios of European Stock Options” on page 10-19
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Input Arguments

`Price` — Current price of underlying asset
numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of option
numeric

Exercise price of the option, specified as a numeric valuie.

Data Types: `double`

`Rate` — Annualized, continuously compounded risk-free rate of return over life of option
positive decimal

Annualized, continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal value.

Data Types: `double`

`Time` — Time (in years) to expiration of the option
numeric

Time (in years) to expiration of the option, specified as a numeric value.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), specified as a positive decimal value.

Data Types: `double`

`Yield` — Annualized, continuously compounded yield of the underlying asset over life of option
`0` (default) | decimal

(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, specified as a decimal value. 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.

Data Types: `double`

Output Arguments

`CallRho` — Call option rho
numeric

Call option rho, returned as a numeric value.

`PutRho` — Put option rho
numeric

Put option rho, returned as a numeric value.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003.

See Also

`blsdelta` | `blsgamma` | `blslambda` | `blsprice` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blstheta

Black-Scholes sensitivity to time-until-maturity change

Syntax

```
[CallTheta,PutTheta] = blstheta(Price,Strike,Rate,Time,Volatility)
[CallTheta,PutTheta] = blstheta(___,Yield)
```

Description

`[CallTheta,PutTheta] = blstheta(Price,Strike,Rate,Time,Volatility)` returns the call option theta `CallTheta`, and the put option theta `PutTheta`.

Theta is the sensitivity in option value with respect to time and is measured in years. `CallTheta` or `PutTheta` can be divided by 365 to get Theta per calendar day or by 252 to get Theta by trading day.

`blstheta` uses `normcdf`, the normal cumulative distribution function, and `normpdf`, the normal probability desnity function, in the Statistics and Machine Learning Toolbox.

Note `blstheta` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

`[CallTheta,PutTheta] = blstheta(___,Yield)` adds an optional argument for `Yield`.

Examples

Compute the Black-Scholes Sensitivity to Time-Until-Maturity Change (Theta)

This example shows how to compute theta, the sensitivity in option value with respect to time.

```
[CallTheta, PutTheta] = blstheta(50, 50, 0.12, 0.25, 0.3, 0)
```

```
CallTheta = -8.9630
```

```
PutTheta = -3.1404
```

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset

numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of option

numeric

Exercise price of the option, specified as a numeric valuie.

Data Types: `double`

`Rate` — Annualized, continuously compounded risk-free rate of return over life of option

positive decimal

Annualized, continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal value.

Data Types: `double`

`Time` — Time (in years) to expiration of the option
numeric

Time (in years) to expiration of the option, specified as a numeric value.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), specified as a positive decimal value.

Data Types: `double`

`Yield` — Annualized, continuously compounded yield of the underlying asset over life of option
`0` (default) | decimal

(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, specified as a decimal value. 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.

Data Types: `double`

Output Arguments

`CallTheta` — Call option theta
numeric

Call option theta, returned as a numeric value.

`PutTheta` — Put option theta
numeric

Put option theta, returned as a numeric value.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003.

See Also

`blsdelta` | `blsgamma` | `blslambda` | `blsprice` | `blsrho` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

blsvega

Black-Scholes sensitivity to underlying price volatility

Syntax

```
Vega = blsvega(Price,Strike,Rate,Time,Volatility)
Vega = blsvega(___,Yield)
```

Description

`Vega = blsvega(Price,Strike,Rate,Time,Volatility)` rate of change of the option value with respect to the volatility of the underlying asset. `blsvega` uses `normpdf`, the normal probability density function in the Statistics and Machine Learning Toolbox.

Note `blsvega` can handle other types of underlies like Futures and Currencies. When pricing Futures (Black model), enter the input argument `Yield` as:

```
Yield = Rate
```

When pricing currencies (Garman-Kohlhagen model), enter the input argument `Yield` as:

```
Yield = ForeignRate
```

where `ForeignRate` is the continuously compounded, annualized risk-free interest rate in the foreign country.

`Vega = blsvega(___,Yield)` adds an optional argument for `Yield`.

Examples

Compute Black-Scholes Sensitivity to Underlying Price Volatility (Vega)

This example shows how to compute vega, the rate of change of the option value with respect to the volatility of the underlying asset.

```
Vega = blsvega(50, 50, 0.12, 0.25, 0.3, 0)
```

```
Vega = 9.6035
```

- "Pricing and Analyzing Equity Derivatives" on page 2-48
- "Greek-Neutral Portfolios of European Stock Options" on page 10-19
- "Plotting Sensitivities of an Option" on page 10-31
- "Plotting Sensitivities of a Portfolio of Options" on page 10-34

Input Arguments

`Price` — Current price of underlying asset
numeric

Current price of the underlying asset, specified as a numeric value.

Data Types: `double`

`Strike` — Exercise price of option
numeric

Exercise price of the option, specified as a numeric valuie.

Data Types: `double`

`Rate` — Annualized, continuously compounded risk-free rate of return over life of option
positive decimal

Annualized, continuously compounded risk-free rate of return over the life of the option, specified as a positive decimal value.

Data Types: `double`

`Time` — Time (in years) to expiration of the option
numeric

Time (in years) to expiration of the option, specified as a numeric value.

Data Types: `double`

`Volatility` — Annualized asset price volatility
positive decimal

Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), specified as a positive decimal value.

Data Types: `double`

`Yield` — Annualized, continuously compounded yield of the underlying asset over life of option
`0` (default) | decimal

(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, specified as a decimal value. 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.

Data Types: `double`

Output Arguments

`Vega` — Rate of change of option value with respect to volatility of underlying asset
numeric

Rate of change of the option value with respect to the volatility of the underlying asset, returned as a numeric value.

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003.

See Also

`blsdelta` | `blsgamma` | `blslambda` | `blsprice` | `blsrho` | `blstheta`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

bm class

Brownian motion models

Description

The `bm` constructor creates and displays Brownian motion (sometimes called *arithmetic Brownian motion* or *generalized Wiener process*) `bm` objects that derive from the `sdeld` (SDE with drift rate expressed in linear form) class. Use `bm` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Brownian motion stochastic processes. This enables you to transform a vector of `NBROWNS` uncorrelated, zero-drift, unit-variance rate Brownian components into a vector of `NVARS` Brownian components with arbitrary drift, variance rate, and correlation structure.

The `bm` constructor allows you to simulate any vector-valued BM process of the form:

$$dX_t = \mu(t)dt + V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`1` drift-rate vector.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` vector of (possibly) correlated zero-drift/unit-variance rate Brownian components.

Construction

`BM = bm(Mu,Sigma)` constructs a default `bm` object.

`BM = bm(Mu,Sigma,Name,Value)` constructs a `bm` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `bm` object, see `bm`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Mu` — `Mu` represents the parameter μ

array or deterministic function of time or deterministic function of time and state

`Mu` represents the parameter μ, specified as an array or deterministic function of time.

If you specify `Mu` as an array, it must be an `NVARS`-by-`1` column vector representing the drift rate (the expected instantaneous rate of drift, or time trend).

As a deterministic function of time, when `Mu` is called with a real-valued scalar time `t` as its only input, `Mu` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Mu` as a function of time and state, it calculates the expected instantaneous rate of drift. This function must generate an `NVARS`-by-`1` column vector when invoked with two inputs:

- A real-valued scalar observation time *t*.

- An NVARS-by-1 state vector X_t.

Data Types: double | function_handle

Sigma — Sigma represents the parameter *V*
array or deterministic function of time or deterministic function of time and state

Sigma represents the parameter *V*, specified as an array or a deterministic function of time.

If you specify Sigma as an array, it must be an NVARS-by-NBROWNS matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of Sigma corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when Sigma is called with a real-valued scalar time t as its only input, Sigma must produce an NVARS-by-NBROWNS matrix. If you specify Sigma as a function of time and state, it must return an NVARS-by-NBROWNS matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An NVARS-by-1 state vector X_t.

Although the gbm constructor enforces no restrictions on the sign of Sigma volatilities, they are usually specified as positive values.

Data Types: double | function_handle

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

For more information on using optional name-value arguments, see bm.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the `gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`Simulation` — User-defined simulation function or SDE simulation method
if you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `function_handle`

Methods

Inherited Methods

The following methods are inherited from the`sde` class.

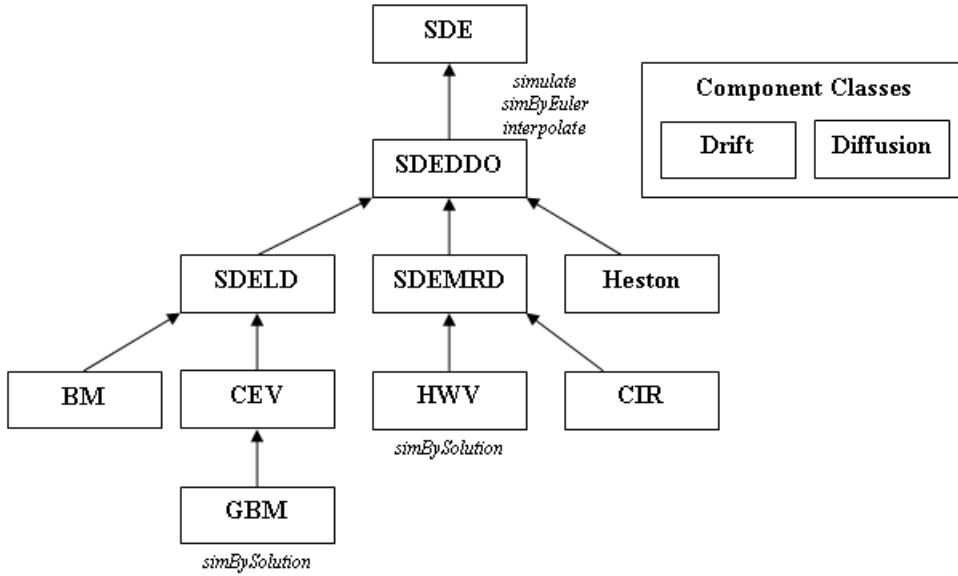
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a bm Object

Create a univariate Brownian motion (`bm`) object to represent the model: $dX_t = 0.3dW_t$.

```
obj = bm(0, 0.3) % (A = Mu, Sigma)

obj =
   Class BM: Brownian Motion
   -------------------------------------
```

```
  Dimensions: State = 1, Brownian = 1
---------------------------------------
   StartTime: 0
  StartState: 0
 Correlation: 1
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
          Mu: 0
       Sigma: 0.3
```

`bm` objects display the parameter `A` as the more familiar `Mu`.

The `bm` class also provides an overloaded Euler simulation method that improves run-time performance in certain common situations. This specialized method is invoked automatically only if *all* the following conditions are met:

- The expected drift, or trend, rate `Mu` is a column vector.
- The volatility rate, `Sigma`, is a matrix.
- No end-of-period adjustments and/or processes are made.
- If specified, the random noise process `Z` is a three-dimensional array.
- If `Z` is unspecified, the assumed Gaussian correlation structure is a double matrix.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `bm` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering.* New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeld` | `simByEuler` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

bm

Construct Brownian motion models

Syntax

```
BM = bm(Mu, Sigma)

BM = bm(Mu, Sigma, 'Name1', Value1, 'Name2', Value2, ...)
```

Class

`bm`

Description

This constructor creates and displays Brownian motion (sometimes called *arithmetic Brownian motion* or *generalized Wiener process*) objects that derive from the`sdeld` (SDE with drift rate expressed in linear form) class. Use `bm` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Brownian motion stochastic processes. This enables you to transform a vector of `NBROWNS` uncorrelated, zero-drift, unit-variance rate Brownian components into a vector of `NVARS` Brownian components with arbitrary drift, variance rate, and correlation structure.

The `bm` method allows you to simulate any vector-valued BM process of the form:
$dX_t = \mu(t)dt + V(t)dW_t$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`1` drift-rate vector.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` vector of (possibly) correlated zero-drift/unit-variance rate Brownian components.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Mu`	`Mu` represents μ. If you specify `Mu` as an array, it must be an `NVARS`-by-`1` column vector representing the drift rate (the expected instantaneous rate of drift, or time trend). As a deterministic function of time, when `Mu` is called with a real-valued scalar time `t` as its only input, `Mu` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Mu` as a function of time and state, it calculates the expected instantaneous rate of drift. This function must generate an `NVARS`-by-`1` column vector when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t. Although the constructor does not enforce restrictions on the sign of this argument, `Sigma` is specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.

`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `bm` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `bm` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `bm` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`BM`	Object of class `BM` with the following displayed parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method • `Mu`: Access function for the input argument `Mu`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Algorithm

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `bm` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

Examples

"Creating Brownian Motion (BM) Models" on page 17-25

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeld`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2

"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

bondDefaultBootstrap

Bootstrap default probability curve from bond prices

Syntax

```
[ProbabilityData,HazardData] = bondDefaultBootstrap(ZeroData,
MarketData,Settle)
[ProbabilityData,HazardData] = bondDefaultBootstrap(___,Name,Value)
```

Description

`[ProbabilityData,HazardData] = bondDefaultBootstrap(ZeroData, MarketData,Settle)` bootstraps the default probability curve from bond prices.

Using `bondDefaultBootstrap`, you can:

- Extract discrete default probabilities for a certain period from market bond data.
- Interpolate these default probabilities to get the default probability curve for pricing and risk management purposes.

`[ProbabilityData,HazardData] = bondDefaultBootstrap(___,Name,Value)` adds optional name-value pair arguments.

Examples

Determine the Default Probability and Hazard Rate Values for Treasury Bonds

Use the following bond data.

```
Settle = datenum('08-Jul-2016');
MarketDate = datenum({'06/15/2018', '01/08/2019', '02/01/2021', '03/18/2021', '08/04/2
CouponRate = [2.240 2.943 5.750 3.336 4.134]'/100;
MarketPrice = [101.300 103.020 115.423 104.683 108.642]';
MarketData = [MarketDate,MarketPrice,CouponRate];
```

Calculate the `ProbabilityData` and `HazardData`.

```
TreasuryParYield = [0.26 0.28 0.36 0.48 0.61 0.71 0.95 1.19 1.37 1.69 2.11]'/100;
TreasuryDates = datemnth(Settle, [[1 3 6], 12 * [1 2 3 5 7 10 20 30]]');
[ZeroRates, CurveDates] = pyld2zero(TreasuryParYield, TreasuryDates, Settle);
ZeroData = [CurveDates, ZeroRates];
format longg
[ProbabilityData,HazardData]=bondDefaultBootstrap(ZeroData,MarketData,Settle)
format
```

```
ProbabilityData =

                    737226        0.0299675399937611
                    737433        0.0418832295824677
                    738188        0.0905183328842623
                    738233         0.101248065083714
                    739833         0.233002708031915


HazardData =

                    737226        0.0157077745460244
                    737433        0.0217939816590409
                    738188         0.025184912824721
                    738233        0.0962608718640789
                    739833        0.0361632398787917
```

Reprice a Bond Listed on the Default Probability Curve

Reprice one of the bonds from bonds list based on the default probability curve. The expected result of this repricing is a perfect match with the market quote.

Use the following Treasury data from US Department of the Treasury.

```
Settle = datetime('08-Jul-2016','Locale','en_US');
TreasuryParYield = [0.26 0.28 0.36 0.48 0.61 0.71 0.95 1.19 1.37 1.69 2.11]'/100;
TreasuryDates = datemnth(Settle, [[1 3 6], 12 * [1 2 3 5 7 10 20 30]]');
```

Preview the bond date using semiannual coupon bonds with market quotes, coupon rates, and a settle date of July-08-2016.

```
MarketDate = datenum({'06/01/2017','06/01/2019','06/01/2020','06/01/2022'}','mm/dd/yyyy
CouponRate = [7 8 9 10]'/100;
MarketPrice = [101.300 109.020 114.42 118.62]';
MarketData = [MarketDate, MarketPrice, CouponRate];

BondList = array2table(MarketData, 'VariableNames', {'Maturity', 'Price','Coupon'});
BondList.Maturity = datetime(BondList.Maturity,'Locale','en_US','ConvertFrom','datenum'
BondList.Maturity.Format = 'MMM-dd-yyyy'
```

```
BondList =

  4x3 table

     Maturity      Price     Coupon
    ___________    ______    ______

    Jun-01-2017     101.3    0.07
    Jun-01-2019    109.02    0.08
    Jun-01-2020    114.42    0.09
    Jun-01-2022    118.62     0.1
```

Choose the second coupon bond as the one to be priced.

```
number = 2;
TestCase = BondList(number, :);
```

Preview the risk-free rate data provided here that is based on a continuous compound rate.

```
[ZeroRates, CurveDates] = pyld2zero(TreasuryParYield, TreasuryDates, Settle);
ZeroData = [datenum(CurveDates), ZeroRates];
RiskFreeRate = array2table(ZeroData, 'VariableNames', {'Date', 'Rate'});
RiskFreeRate.Date = datetime(RiskFreeRate.Date,'Locale','en_US','ConvertFrom','datenum'
RiskFreeRate.Date.Format = 'MMM-dd-yyyy'
```

```
RiskFreeRate =

  11x2 table

       Date          Rate
    ___________    _________
```

```
    Aug-08-2016    0.0026057
    Oct-08-2016    0.0027914
    Jan-08-2017    0.0035706
    Jul-08-2017    0.0048014
    Jul-08-2018    0.0061053
    Jul-08-2019    0.0071115
    Jul-08-2021    0.0095416
    Jul-08-2023     0.012014
    Jul-08-2026     0.013883
    Jul-08-2036     0.017359
    Jul-08-2046     0.022704
```

Bootstrap the probability of default (PD) curve from the bonds.

```
format longg
[defaultProb1, hazard1] = bondDefaultBootstrap(ZeroData, MarketData, Settle)
format
```

```
defaultProb1 =

                    736847        0.0704863142317494
                    737577         0.162569420050034
                    737943         0.217308133826187
                    738673           0.38956773145021


hazard1 =

                    736847        0.0813390794774647
                    737577        0.0521615800986284
                    737943        0.0674145844133175
                    738673          0.12428587278862
```

Reformat the default probability and hazard rate for a better representation.

```
DefProbHazard = [defaultProb1, hazard1(:,2)];
DefProbHazardTable = array2table(DefProbHazard, 'VariableNames', {'Date', 'DefaultProba
DefProbHazardTable.Date = datetime(DefProbHazardTable.Date,'Locale','en_US','ConvertFrc
DefProbHazardTable.Date.Format = 'MMM-dd-yyyy'
```

```
DefProbHazardTable =
```

```
  4x3 table

      Date        DefaultProbability    HazardRate
    ___________    __________________    __________

    Jun-01-2017    0.070486              0.081339
    Jun-01-2019     0.16257              0.052162
    Jun-01-2020     0.21731              0.067415
    Jun-01-2022     0.38957               0.12429
```

Preview the selected bond to reprice based on the PD curve.

```
TestCase
```

```
TestCase =

  1x3 table

     Maturity      Price     Coupon
    ___________    ______    ______

    Jun-01-2019    109.02    0.08
```

To reprice the bond, first generate cash flows and payment dates.

```
[Payments, PaymentDates] = cfamounts(TestCase.Coupon, Settle, TestCase.Maturity);
AccInt=-Payments(1);
    % Truncate the payments as well as payment dates for calculation
    % PaymentDates(1) is the settle date, no need for following calculations
PaymentDates = PaymentDates(2:end)
Payments = Payments(2:end)
```

```
PaymentDates =

  1x6 datetime array

Columns 1 through 5

   01-Dec-2016   01-Jun-2017   01-Dec-2017   01-Jun-2018   01-Dec-2018
```

```
Column 6

   01-Jun-2019


Payments =

     4     4     4     4     4   104
```

Calculate the discount factors on the payment dates.

```
DF = zero2disc(interp1(RiskFreeRate.Date, RiskFreeRate.Rate, PaymentDates, 'linear', 'e

DF =

    0.9987    0.9959    0.9926    0.9887    0.9845    0.9799
```

Assume that the recovery amount is a fixed proportion of bond's face value. The bond's face value is `100`, and the recovery ratio is set to 40% as assumed in `bondDefaultBootstrap`.

```
Num = length(Payments);
RecoveryAmount = repmat(100*0.4, 1, Num)

RecoveryAmount =

    40    40    40    40    40    40
```

Calculate the probability of default based on the default curve.

```
DefaultProb1 = bondDefaultBootstrap(ZeroData, MarketData, Settle, 'ZeroCompounding', -1
SurvivalProb = 1 - DefaultProb1(:,2)

SurvivalProb =

    0.9680
    0.9295
    0.9055
    0.8823
```

```
    0.8595
    0.8375
```

Calculate the model-based clean bond price.

```
DirtyPrice = DF * (SurvivalProb.*Payments') + (RecoveryAmount.*DF) * (-diff([1;Survival
ModelPrice = DirtyPrice - AccInt
```

```
ModelPrice =

  109.0200
```

Compare the repriced bond to the market quote.

```
ResultTable = TestCase;
ResultTable.ModelPrice = ModelPrice;
ResultTable.Difference = ModelPrice - TestCase.Price
```

```
ResultTable =

  1x5 table

     Maturity      Price     Coupon    ModelPrice    Difference
    ____________    ______    ______    __________    __________

    Jun-01-2019    109.02    0.08      109.02        0
```

- “Creating an IRDataCurve Object” (Financial Instruments Toolbox)

Input Arguments

`ZeroData` — Zero rate data

matrix | `IRDataCurve` object

Zero rate data, specified as an `M`-by-`2` matrix of dates and zero rates or an `IRDataCurve` object of zero rates. For array input, the dates must be entered as serial date numbers, and discount rate must be in decimal form.

When ZeroData is an IRDataCurve object, ZeroCompounding and ZeroBasis are implicit in ZeroData and are redundant inside this function. In this case, specify these optional parameters when constructing the IRDataCurve object before using this bondDefaultBootstrap function.

For more information on an IRDataCurve object, see "Creating an IRDataCurve Object" (Financial Instruments Toolbox).

Data Types: double

MarketData — Bond market data
matrix

Bond market data, specified as an N-by-3 matrix of maturity dates, market prices, and coupon rates for bonds. The dates must be entered as serial date numbers, market prices must be numeric values, and coupon rate must be in decimal form.

Note A warning is displayed when MarketData is not sorted in ascending order by time.

Data Types: double

Settle — Settlement date
serial date number | date character vector | datetime object | date string object

Settlement date, specified as a serial date number, a date character vector, a datetime object, or a date string object. Settle must be earlier than or equal to the maturity dates in MarketData.

Data Types: double | char | datetime | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Note Any optional input of size N-by-1 is also acceptable as an array of size 1-by-N, or as a single value applicable to all contracts.

Example: `[ProbabilityData,HazardData] = bondDefaultBootstrap(ZeroData,MarketData,Settle,'RecoveryRate',Recovery,'ZeroCompounding',-1)`

`RecoveryRate` — Recovery rate
`0.4` (default) | decimal

Recovery rate, specified as the comma-separated pair consisting of `'RecoveryRate'` and a `N`-by-`1` vector of recovery rates, expressed as a decimal from `0` through `1`.

Data Types: `double`

`ProbabilityDates` — Dates for output of default probability data
column of dates in `MarketData` (default) | serial date number | date character vector | datetime object | date string object

Dates for the output of default probability data, specified as the comma-separated pair consisting of `'ProbabilityDates'` and a `P`-by-`1` vector, given as serial date numbers, datetime objects, date character vectors, or date string objects.

Data Types: `double` | `char` | `datetime` | `string`

`ZeroCompounding` — Compounding frequency of the zero curve
`2` (semiannual) (default) | integer with value of `1`,`2`,`3`,`4`,`6`,`12`, or `-1`

Compounding frequency of the zero curve, specified as the comma-separated pair consisting of `'ZeroCompounding'` and a `N`-by-`1` vector. Values are:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding
- `-1` — Continuous compounding

Data Types: `double`

`ZeroBasis` — Basis of the zero curve
`0` (actual/actual) (default) | integer with value of `0` to `13`

Basis of the zero curve, specified as the comma-separated pair consisting of `'ZeroBasis'` and the same values listed for `Basis`.

Data Types: `double`

`RecoveryMethod` — Recovery method
`'facevalue'` (default) | character vector with value of `'presentvalue'` or `'facevalue'` | string object with value of `'presentvalue'` or `'facevalue'`

Recovery method, specified as the comma-separated pair consisting of `'RecoveryMethod'` and a character vector or a string with a value of `'presentvalue'` or `'facevalue'`.

- `'presentvalue'` assumes that upon default, a bond is valued at a given fraction to the hypothetical present value of its remaining cash flows, discounted at risk-free rate.
- `'facevalue'` assumes that a bond recovers a given fraction of its face value upon recovery.

Data Types: `char` | `string`

`Face` — Face or par value
`100` (default) | numeric

Face or par value, specified as the comma-separated pair consisting of `'Face'` and a `NINST`-by-`1` vector of bonds.

Data Types: `double`

`Period` — Payment frequency
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Payment frequency, specified as the comma-separated pair consisting of `'Period'` and a `N`-by-`1` vector with values of `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Day-count basis of the instrument, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag

`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as the comma-separated pair consisting of `'EndMonthRule'` and a nonnegative integer, `0` or `1`, using a `NINST`-by-`1` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `double`

`IssueDate` — Bond issue date
if `IssueDate` not specified, cash flow payment dates determined from other inputs (default) | serial date number | date character vector | datetime object | date string object

Bond issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a `N`-by-`1` vector, given as serial date numbers, datetime objects, date character vectors, or date string objects.

Data Types: `double` | `char` | `datetime` | `string`

`FirstCouponDate` — First actual coupon date
if you do not specify a `FirstCouponDate`, cash flow payment dates are determined from other inputs (default) | serial date number

First actual coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a serial date number. `FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

Data Types: `double`

`LastCouponDate` — Last actual coupon date
if you do not specify a `LastCouponDate`, cash flow payment dates are determined from other inputs (default) | scalar for serial date number

Last actual coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a serial date number. `LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Data Types: `double`

`StartDate` — Forward starting date of payments
if you do not specify `StartDate`, effective start date is `Settle` date (default) | serial date number

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a serial date number. `StartDate` is when a bond actually starts (the

date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

Data Types: `double`

`BusinessDayConvention` — Business day conventions
`'actual'` (default) | character vector or string object with with values`'actual'`, `'follow'`, `'modifiedfollow'`, `'previous'`or `'modifiedprevious'`

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and a character vector or a string object. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `'actual'` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell` | `string`

Output Arguments

`ProbabilityData` — Default probability values
matrix

Default probability values, returned as a `P`-by-`2` matrix with dates and corresponding cumulative default probability values. The dates match those in `MarketData`, unless the optional input parameter `ProbabilityDates` is provided.

`HazardData` — Hazard rate values
matrix

Hazard rate values, returned as an `N`-by-`2` matrix with dates and corresponding hazard rate values for the survival probability model. The dates match those in `MarketData`.

Note A warning is displayed when nonmonotone default probabilities (that is, negative hazard rates) are found.

References

[1] Jarrow, Robert A., and Stuart Turnbull. "Pricing Derivatives on Financial Securities Subject to Credit Risk." *Journal of Finance*. 50.1, 1995, pp. 53–85.

[2] Berd, A., Mashal, R. and Peili Wang. *"Defining, Estimating and Using Credit Term Structures."* Research report, Lehman Brothers, 2004.

See Also

`IRDataCurve` | `cdsbootstrap`

Topics

"Creating an IRDataCurve Object" (Financial Instruments Toolbox)

Introduced in R2017a

bndconvp

Bond convexity given price

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
[YearConvexity,PerConvexity] = bndconvp(Price,CouponRate,Settle,
Maturity)
[YearConvexity,PerConvexity] = bndconvp(___,Name,Value)
```

Description

`[YearConvexity,PerConvexity] = bndconvp(Price,CouponRate,Settle, Maturity)` computes the convexity of `NUMBONDS` fixed income securities given a clean price for each bond.

`bndconvp` determines the convexity for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). `bndconvp` also determines the convexity of a zero coupon bond.

`[YearConvexity,PerConvexity] = bndconvp(___,Name,Value)` adds optional name-value pair arguments.

Examples

Find Bond Convexity Given Price

This example shows how to compute the convexity of three bonds given their prices.

```
Price = [106; 100; 98];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[YearConvexity, PerConvexity] = bndconvp(Price,...
CouponRate,Settle, Maturity, Period, Basis)

YearConvexity =

   21.4447
   21.0363
   20.8951


PerConvexity =

   85.7788
   84.1454
   83.5803
```

Find Bond Convexity Given Price Using datetime Inputs

This example shows how to compute the convexity of three bonds given their prices using datetime inputs.

```
Price = [106; 100; 98];
CouponRate = 0.055;
Period = 2;
Basis = 0;
Settle = datetime('02-Aug-1999','Locale','en_US');
Maturity = datetime('15-Jun-2004','Locale','en_US');
[YearConvexity, PerConvexity] = bndconvp(Price,...
CouponRate, Settle, Maturity, Period, Basis)
```

```
YearConvexity =

   21.4447
   21.0363
   20.8951


PerConvexity =

   85.7788
   84.1454
   83.5803
```

- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7

Input Arguments

`Price` — Clean price (excludes accrued interest)

numeric

Clean price (excludes accrued interest), specified as numeric value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond

decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`Settle` — Settlement date for certificate of deposit

serial date number | date character vector | datetime

Settlement date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date for certificate of deposit
serial date number | date character vector | datetime

Maturity date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[YearConvexity,PerConvexity] = bndconvp(Price,CouponRate,Settle, Maturity,'Period',4,'Basis',7)`

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

Data Types: double | char | datetime

LastCouponDate — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

Data Types: double | char | datetime

StartDate — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays. The StartDate is when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

If you do not specify a StartDate, the effective start date is the Settle date.

Data Types: double | char | datetime

Face — Face value of bond
100 (default) | numeric

Face value of the bond, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector.

Data Types: double

CompoundingFrequency — Compounding frequency for yield calculation
SIA bases uses2, ICMA bases uses 1 (default) | integer with value of 1, 2, 3, 4, 6, or 12

Compounding frequency for yield calculation, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the actual/actual day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the`Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period
`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.
`LastCouponInterest` is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`
- `compound`

Data Types: `char` | `cell`

Output Arguments

`YearConvexity` — Yearly (annualized) convexity
numeric

Yearly (annualized) convexity, returned as a `NUMBONDS`-by-`1` vector.

`PerConvexity` — Periodic convexity reported on semiannual bond basis
numeric

Periodic convexity reported on a semiannual bond basis (in accordance with SIA convention), returned as a `NUMBONDS`-by-`1` vector.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`bndconvy` | `bnddurp` | `bnddury` | `cfconv` | `cfdur` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7
“Yield Conventions” on page 2-34

Introduced before R2006a

bndconvy

Bond convexity given yield

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
[YearConvexity,PerConvexity] = bndconvy(Yield,CouponRate,Settle,
Maturity)
[YearConvexity,PerConvexity] = bndconvy(___,Name,Value)
```

Description

`[YearConvexity,PerConvexity] = bndconvy(Yield,CouponRate,Settle, Maturity)` computes the convexity of `NUMBONDS` fixed income securities given a clean price for each bond.

`bndconvy` determines the convexity for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). `bndconvy` also determines the convexity of a zero coupon bond.

`[YearConvexity,PerConvexity] = bndconvy(___,Name,Value)` adds optional name-value pair arguments.

Examples

Find Bond Convexity Given Yield

This example shows how to compute the convexity of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[YearConvexity, PerConvexity]=bndconvy(Yield, CouponRate,...
Settle, Maturity, Period, Basis)
```

```
YearConvexity =

   21.4825
   21.0358
   20.8885


PerConvexity =

   85.9298
   84.1434
   83.5541
```

Find Bond Convexity Given Yield Using datetime Inputs

This example shows how to use `datetime` inputs to compute the convexity of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];
CouponRate = 0.055;
Settle = datetime('02-Aug-1999','Locale','en_US');
Maturity = datetime('15-Jun-2004','Locale','en_US');
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
```

- `on page 10-7

Input Arguments

`Yield` — Yield to maturity on semiannual basis
numeric

Yield to maturity on a semiannual basis, specified as numeric value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond
decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`Settle` — Settlement date for certificate of deposit
serial date number | date character vector | datetime

Settlement date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date for certificate of deposit
serial date number | date character vector | datetime

Maturity date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[YearConvexity,PerConvexity] = bndconvy(Yield,CouponRate,Settle, Maturity,'Period',4,'Basis',7)`

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `StartDate` is when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency for yield calculation
SIA bases uses`2`, ICMA bases uses `1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the actual/actual day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the`Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period
`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.
`LastCouponInterest` is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`
- `compound`

Data Types: `char` | `cell`

Output Arguments

`YearConvexity` — Yearly (annualized) convexity
numeric

Yearly (annualized) convexity, returned as a `NUMBONDS`-by-`1` vector.

`PerConvexity` — Periodic convexity reported on semiannual bond basis
numeric

Periodic convexity reported on a semiannual bond basis (in accordance with SIA convention), returned as a `NUMBONDS`-by-`1` vector.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`bndconvp` | `bnddurp` | `bnddury` | `cfconv` | `cfdur` | `datetime`

Topics

` on page 10-7
“Yield Conventions” on page 2-34

Introduced before R2006a

bnddurp

Bond duration given price

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
[ModDuration,YearDuration,PerDuration] = bnddurp(Price,CouponRate,
Settle,Maturity)
[ModDuration,YearDuration,PerDuration] = bnddurp(___,Name,Value)
```

Description

`[ModDuration,YearDuration,PerDuration] = bnddurp(Price,CouponRate, Settle,Maturity)` computes the Macaulay and modified duration of `NUMBONDS` fixed-income securities given a clean price for each bond.

`bnddurp` determines the Macaulay and modified duration for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). `bnddurp` also determines the Macaulay and modified duration for a zero coupon bond.

`[ModDuration,YearDuration,PerDuration] = bnddurp(___,Name,Value)` adds optional name-value pair arguments.

Examples

Find Bond Duration Given Price

This example shows how to compute the duration of three bonds given their prices.

```
Price = [106; 100; 98];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[ModDuration, YearDuration, PerDuration] = bnddurp(Price,...
CouponRate, Settle, Maturity, Period, Basis)

ModDuration =

    4.2400
    4.1925
    4.1759


YearDuration =

    4.3275
    4.3077
    4.3007


PerDuration =

    8.6549
    8.6154
    8.6014
```

Find Bond Duration Given Price Using datetime Inputs

This example shows how to use `datetime` inputs to compute the duration of three bonds given their prices.

```
Price = [106; 100; 98];
CouponRate = 0.055;
```

```
Settle = datetime('02-Aug-1999','Locale','en_US');
Maturity = datetime('15-Jun-2004','Locale','en_US');
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
```

- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7

Input Arguments

`Price` — Clean price (excludes accrued interest)

numeric

Clean price (excludes accrued interest), specified as numeric value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond

decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal value using a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector.

Data Types: double

Settle — Settlement date for certificate of deposit
serial date number | date character vector | datetime

Settlement date for the certificate of deposit, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays. The Settle date must be before the Maturity date.

Data Types: double | char | datetime

Maturity — Maturity date for certificate of deposit
serial date number | date character vector | datetime

Maturity date for the certificate of deposit, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: double | char | datetime

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: [ModDuration,YearDuration,PerDuration] = bnddurp(Price,CouponRate,Settle,Maturity,'Period',4,'Basis',7)

Period — Number of coupon payments per year
2 (default) | numeric with values 0, 1, 2, 3, 4, 6 or 12

Number of coupon payments per year, specified as scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using the values: 0, 1, 2, 3, 4, 6, or 12.

Data Types: double

Basis — Day-count basis of instrument
0 (default) | numeric values: 0,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag

`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `StartDate` is when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: double | char | datetime

Face — Face value of bond
100 (default) | numeric

Face value of the bond, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector.

Data Types: double

CompoundingFrequency — Compounding frequency for yield calculation
SIA bases uses2, ICMA bases uses 1 (default) | integer with value of 1, 2, 3, 4, 6, or 12

Compounding frequency for yield calculation, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector.

- 1 — Annual compounding
- 2 — Semiannual compounding
- 3 — Compounding three times per year
- 4 — Quarterly compounding
- 6 — Bimonthly compounding
- 12 — Monthly compounding

Note By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ICMA bases (8-12) use an annual compounding convention.

Data Types: double

DiscountBasis — Basis used to compute the discount factors for computing the yield
SIA uses 0 (default) | integers of the set [0...13] | vector of integers of the set [0...13]

Basis used to compute the discount factors for computing the yield, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)

- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the actual/actual day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the`Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period

`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. `LastCouponInterest` is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`

- `compound`

Data Types: `char` | `cell`

Output Arguments

`ModDuration` — Modified duration in years
numeric

Modified duration in years reported on a semiannual bond basis (in accordance with SIA convention), returned as a `NUMBONDS`-by-`1` vector.

`YearDuration` — Macaulay duration in years
numeric

Macaulay duration in years, returned as a `NUMBONDS`-by-`1` vector.

`PerDuration` — Periodic Macaulay duration
numeric

Periodic Macaulay duration reported on a semiannual bond basis (in accordance with SIA convention), returned as a `NUMBONDS`-by-`1` vector.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations.* Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation.* McGraw-Hill, 1996.

See Also

`bndconvp` | `bndconvy` | `bnddury` | `bndkrdur` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7

“Yield Conventions” on page 2-34

Introduced before R2006a

bnddury

Bond duration given yield

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
[ModDuration,YearDuration,PerDuration] = bnddury(Yield,CouponRate,
Settle,Maturity)
[ModDuration,YearDuration,PerDuration] = bnddury(___,Name,Value)
```

Description

`[ModDuration,YearDuration,PerDuration] = bnddury(Yield,CouponRate, Settle,Maturity)` computes the Macaulay and modified duration of `NUMBONDS` fixed income securities given yield to maturity for each bond.

`bnddury` determines the Macaulay and modified duration for a bond whether the first or last coupon periods in the coupon structure are short or long (that is, whether the coupon structure is synchronized to maturity). `bnddury` also determines the Macaulay and modified duration for a zero coupon bond.

`[ModDuration,YearDuration,PerDuration] = bnddury(___,Name,Value)` adds optional name-value pair arguments.

Examples

Find Bond Duration Given Yield

This example shows how to compute the duration of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[ModDuration,YearDuration,PerDuration]=bnddury(Yield,...
CouponRate, Settle, Maturity, Period, Basis)
```

```
ModDuration =

    4.2444
    4.1924
    4.1751


YearDuration =

    4.3292
    4.3077
    4.3004


PerDuration =

    8.6585
    8.6154
    8.6007
```

Find Bond Duration Given Yield Using datetime Inputs

This example shows how to use `datetime` inputs to compute the duration of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];
CouponRate = 0.055;
```

```
Settle = datetime('02-Aug-1999','Locale','en_US');
Maturity = datetime('15-Jun-2004','Locale','en_US');
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
```

- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7

Input Arguments

`Yield` — Yield to maturity on a semiannual basis

numeric

Yield to maturity on a semiannual basis, specified as numeric value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond

decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal value using a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector.

Data Types: double

Settle — Settlement date for certificate of deposit
serial date number | date character vector | datetime

Settlement date for the certificate of deposit, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays. The Settle date must be before the Maturity date.

Data Types: double | char | datetime

Maturity — Maturity date for certificate of deposit
serial date number | date character vector | datetime

Maturity date for the certificate of deposit, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: double | char | datetime

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: [ModDuration,YearDuration,PerDuration] = bnddury(Yield,CouponRate,Settle,Maturity,'Period',4,'Basis',7)

Period — Number of coupon payments per year
2 (default) | numeric with values 0, 1, 2, 3, 4, 6 or 12

Number of coupon payments per year, specified as scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using the values: 0, 1, 2, 3, 4, 6, or 12.

Data Types: double

Basis — Day-count basis of instrument
0 (default) | numeric values: 0,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag

`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `StartDate` is when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency for yield calculation
SIA bases uses`2`, ICMA bases uses `1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)

- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the actual/actual day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the`Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period

`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. `LastCouponInterest` is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`

- `compound`

Data Types: `char` | `cell`

Output Arguments

`ModDuration` — Modified duration in years
numeric

Modified duration in years reported on a semiannual bond basis (in accordance with SIA convention), returned as a `NUMBONDS`-by-`1` vector.

`YearDuration` — Macaulay duration in years
numeric

Macaulay duration in years, returned as a `NUMBONDS`-by-`1` vector.

`PerDuration` — Periodic Macaulay duration
numeric

Periodic Macaulay duration reported on a semiannual bond basis (in accordance with SIA convention), returned as a `NUMBONDS`-by-`1` vector.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations.* Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation.* McGraw-Hill, 1996.

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bndkrdur` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7

“Yield Conventions” on page 2-34

Introduced before R2006a

bndkrdur

Bond key rate duration given zero curve

Syntax

```
KeyRateDuration = bndkrdur(ZeroData,CouponRate,Settle,Maturity)
KeyRateDuration = bndkrdur(___,Name,Value)
```

Description

`KeyRateDuration = bndkrdur(ZeroData,CouponRate,Settle,Maturity)` computes the key rate durations for one or more bonds given a zero curve and a set of key rates.

`KeyRateDuration = bndkrdur(___,Name,Value)` adds optional name-value pair arguments.

Examples

Find the Bond Key Rate Duration Given the Zero Curve

This example shows how to compute the key rate duration of a bond for key rate times of 2, 5, 10, and 30 years.

```
ZeroRates = [0.0476 .0466 .0465 .0468 .0473 .0478 ...
.0493 .0539 .0572 .0553 .0530]';

ZeroDates = daysadd('31-Dec-1998',[30 360 360*2 360*3 360*5 ...
360*7 360*10 360*15 360*20 360*25 360*30],1);

ZeroData = [ZeroDates ZeroRates];

krdur = bndkrdur(ZeroData,.0525,'12/31/1998',...
'11/15/2028','KeyRates',[2 5 10 30])
```

```
krdur =

    0.2986    0.8791    4.1353    9.5814
```

Find the Bond Key Rate Duration Given the Zero Curve Using datetime Inputs

This example shows how to use `datetime` inputs for `Settle` and `Maturity` and also use a table for `ZeroData` to compute the key rate duration of a bond for key rate times of 2, 5, 10, and 30 years.

```
ZeroRates = [0.0476 .0466 .0465 .0468 .0473 .0478 ...
.0493 .0539 .0572 .0553 .0530]';

ZeroDates = daysadd('31-Dec-1998',[30 360 360*2 360*3 360*5 ...
360*7 360*10 360*15 360*20 360*25 360*30],1);

ZeroData = table(datetime(ZeroDates,'ConvertFrom','datenum','Locale','en_US'), ZeroRate

krdur = bndkrdur(ZeroData,.0525,datetime('12/31/1998','Locale','en_US'),...
datetime('11/15/2028','Locale','en_US'),'KeyRates',[2 5 10 30])

krdur =

    0.2986    0.8791    4.1353    9.5814
```

- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7

Input Arguments

`ZeroData` — Zero curve

matrix | table

Zero Curve, specified as a `numRates`-by-`2` matrix or a `numRates`-by-`2` table.

If `ZeroData` is represented as a `numRates`-by-`2` matrix, the first column is a MATLAB serial date number and the second column is the accompanying zero rates.

If `ZeroData` is a table, the first column can be serial date numbers, date character vectors, or datetime arrays. The second column must be numeric data corresponding to the zero rates.

Data Types: `double` | `table`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond
decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal value using a scalar or a `NUMBONDS`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date for all bonds and zero curve
serial date number | date character vector | datetime

Settlement date for all bonds and zero curve, specified as a scalar using a serial date number, date character vector, or datetime array. `Settle` must be the same settlement date for all the bonds and the zero curve.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date for bonds
serial date number | date character vector | datetime

Maturity date for bonds, specified as a scalar or a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `KeyRateDuration = bndkrdur(ZeroData,.`
`0525,'12/31/1998','11/15/2028','KeyRates',[2 5 10 30])`

`InterpMethod` — Interpolation method used to obtain points from zero curve
`'linear'` (default) | `'cubic'`, `'pchip'`

Interpolation method used to obtain points from the zero curve, specified as a character vector using one of the following values:

- `'linear'` (default)
- `'cubic'`
- `'pchip'`

Data Types: `char`

`ShiftValue` — Value that zero curve is shifted up and down to compute duration
`.01` (100 basis points) (default) | numeric

Value that zero curve is shifted up and down to compute duration, specified as scalar numeric value.

Data Types: `double`

`KeyRates` — Rates to perform the duration calculation
set to each of the zero dates (default) | numeric

Rates to perform the duration calculation, specified as a time to maturity using a scalar or a `NUMBONDS`-by-`1` vector.

Data Types: `double`

`CurveCompounding` — Compounding frequency of curve
2 (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency of the curve, specified as a scalar using one of the following values:.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

.

Data Types: `double`

`CurveBasis` — Basis of the curve

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Basis of the curve, specified as a scalar using one of the following values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`Period` — Number of coupon payments per year

`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NUMBONDS`-by-`1` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays. The `StartDate` is when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NUMBONDS`-by-`1` vector.

Data Types: `double`

Output Arguments

`KeyRateDuration` — Key rate durations
matrix

Key rate durations, returned as a `numBonds`-by-`numRates` matrix.

Algorithms

`bndkrdur` computes the key rate durations for one or more bonds given a zero curve and a set of key rates. By default, the key rates are each of the zero curve rates. For each key rate, the duration is computed by shifting the zero curve up and down by a specified amount (`ShiftValue`) at that particular key rate, computing the present value of the bond in each case with the new zero curves, and then evaluating the following:

$$krdur_i = \frac{(PV_{down} - PV_{up})}{(PV \times ShiftValue \times 2)}$$

Note The shift to the curve is computed by shifting the particular key rate by the `ShiftValue` and then interpolating the values of the curve in the interval between the previous and next key rates. For the first key rate, any curve values before the date are equal to the `ShiftValue`; likewise, for the last key rate, any curve values after the date are equal to the `ShiftValue`.

References

[1] Golub, B., Tilman, L. *Risk Management: Approaches for Fixed Income Markets*. Wiley, 2000.

[2] Tuckman, B. *Fixed Income Securities: Tools for Today's Markets*. Wiley, 2002.

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bnddury` | `datetime`

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-7
"Yield Conventions" on page 2-34

Introduced before R2006a

bndprice

Price fixed-income security from yield to maturity

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
[Price,AccruedInt] = bndprice(Yield,CouponRate,Settle,Maturity)
[Price,AccruedInt] = bndprice(___,Name,Value)
```

Description

`[Price,AccruedInt] = bndprice(Yield,CouponRate,Settle,Maturity)` given bonds with SIA date parameters and yields to maturity, returns the clean prices and accrued interest due.

`[Price,AccruedInt] = bndprice(___,Name,Value)` adds optional name-value pair arguments.

Examples

Price a Treasury Bond from Yield to Maturity

This example shows how to price a treasury bond at three different yield values.

```
Yield = [0.04; 0.05; 0.06];
CouponRate = 0.05;
Settle = '20-Jan-1997';
Maturity = '15-Jun-2002';
```

```
Period = 2;
Basis = 0;

[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,...
Maturity, Period, Basis)

Price =

  104.8106
   99.9951
   95.4384


AccruedInt =

    0.4945
    0.4945
    0.4945
```

Price a Treasury Bond from Yield to Maturity Using datetime Inputs

This example shows how to use `datetime` inputs to price a treasury bond at three different yield values.

```
Yield = [0.04; 0.05; 0.06];
CouponRate = 0.05;
Settle = datetime('20-Jan-1997','Locale','en_US');
Maturity = datetime('15-Jun-2002','Locale','en_US');
Period = 2;
Basis = 0;
[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,...
Maturity, Period, Basis)

Price =

  104.8106
   99.9951
   95.4384


AccruedInt =
```

```
    0.4945
    0.4945
    0.4945
```

Price a Treasury Bond with Different Yield Values

This example shows how to price a Treasury bond at two different yield values that include parameter/value pairs for `CompoundingFrequency`, `DiscountBasis`, and `LastCouponPeriodInterest`.

```
bndprice(.04,0.08,'5/25/2004','4/21/2005','Period',1,'Basis',8, ...
'LastCouponInterest','simple')

ans = 103.4743
```

- "Bond Portfolio for Hedging Duration and Convexity" on page 10-7
- "Pricing Functions" on page 2-35
- "Sensitivity of Bond Prices to Interest Rates" on page 10-3

Input Arguments

`Yield` — Bond yield to maturity

numeric

Bond yield to maturity is specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. `Yield` is on a semiannual basis for `Basis` values `0` through `7` and `13` and an annual basis for `Basis` values `8` through `12`.

Data Types: `double`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond

decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`Settle` — Settlement date of bond
serial date number | date character vector | datetime

Settlement date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of bond
serial date number | date character vector | datetime

Maturity date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
[Price,AccruedInt] =
bndprice(Yield,CouponRate,Settle,Maturity, 'Period',4,'Basis',9)
```

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of instrument
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual

- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency for yield calculation
SIA bases uses`2`, ICMA bases uses `1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA bases uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)

- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the actual/actual day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the `Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period
`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. `LastCouponInterest` is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`
- `compound`

Data Types: `char` | `cell`

Output Arguments

`Price` — Clean price of bond
numeric

Clean price of bond, returned as a `NUMBONDS`-by-`1` vector. The dirty price of the bond is the clean price plus the accrued interest. It equals the present value of the bond cash flows of the yield to maturity with semiannual compounding.

`AccruedInt` — Accrued interest payable at settlement
numeric

accrued interest payable at settlement, returned as a `NUMBONDS`-by-`1` vector.

Definitions

Price and Yield Conventions

The `Price` and `Yield` are related to different formulae for SIA and ICMA conventions.

For SIA conventions, `Price` and `Yield` are related by the formula:

```
Price + Accrued Interest = sum(Cash_Flow*(1+Yield/2)^(-Time))
```

where the sum is over the bond's cash flows and corresponding times in units of semiannual coupon periods.

For ICMA conventions, the `Price` and `Yield` are related by the formula:

```
Price + Accrued Interest = sum(Cash_Flow*(1+Yield)^(-Time))
```

Algorithms

For SIA conventions, the following formula defines bond price and yield:

$$PV = \sum_{i=1}^{n}\left(\frac{CF}{(1+\frac{z}{f})^{TF}}\right),$$

where:

PV =	Present value of a cash flow.

CF =	Cash flow amount.
z =	Risk-adjusted annualized rate or yield corresponding to a given cash flow. The yield is quoted on a semiannual basis.
f =	Frequency of quotes for the yield. Default is `2` for `Basis` values `0` to `7` and `13` and `1` for `Basis` values `8` to `12`. The default can be overridden by specifying the `CompoundingFrequency` name-value pair.
TF =	Time factor for a given cash flow. The time factor is computed using the compounding frequency and the discount basis. If these values are not specified, then the defaults are as follows: `CompoundingFrequency` default is `2` for `Basis` values `0` to `7` and `13` and `1` for `Basis` values `8` to `12`. `DiscountBasis` is `0` for `Basis` values `0` to `7` and `13` and the input `Basis` for `Basis` values `8` to `12`.

Note The `Basis` is always used to compute accrued interest.

For ICMA conventions, the frequency of annual coupon payments determines bond price and yield.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`bndyield` | `cfamounts` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7
“Pricing Functions” on page 2-35
“Sensitivity of Bond Prices to Interest Rates” on page 10-3
“Yield Conventions” on page 2-34

Introduced before R2006a

bndspread

Static spread over spot curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
Spread = bndspread(SpotInfo,Price,Coupon,Settle,Maturity)
Spread = bndspread(___,Name,Value)
```

Description

`Spread = bndspread(SpotInfo,Price,Coupon,Settle,Maturity)` computes the static spread (Z-spread) to benchmark in basis points.

`Spread = bndspread(___,Name,Value)` adds optional name-value pair arguments.

Examples

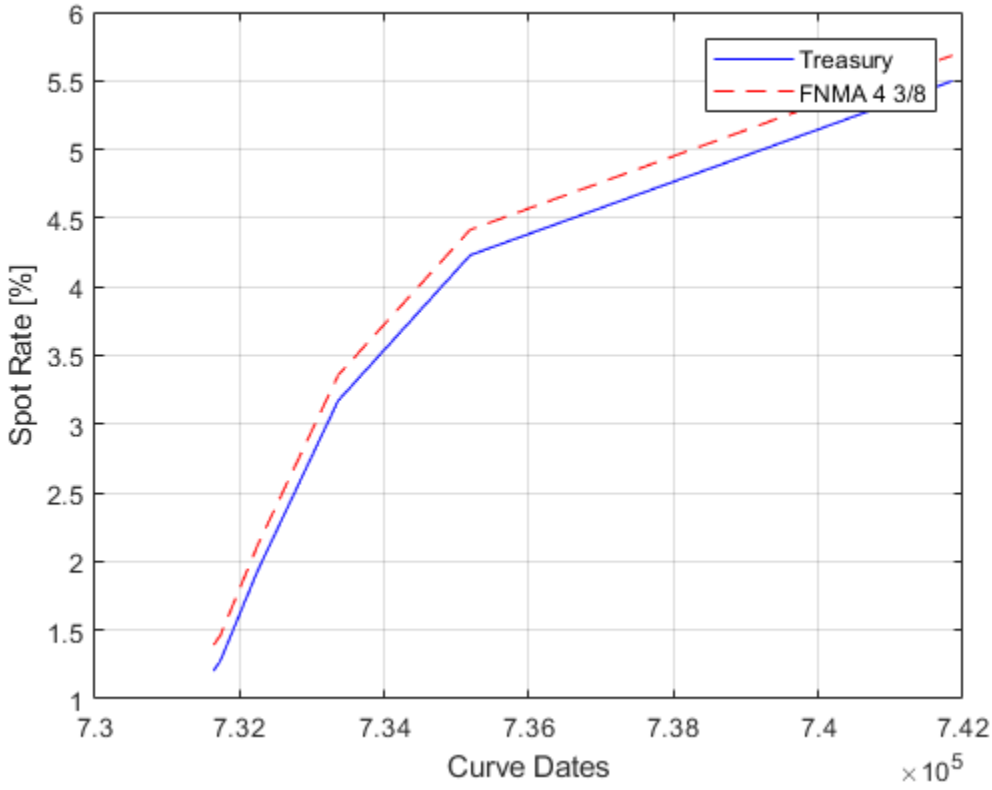
Compute the Static Spread Over a Spot Curve

This example shows how to compute a Federal National Mortgage Association (FNMA) 4 3/8 spread over a Treasury spot curve and plot the results.

```
RefMaturity = [datenum('02/27/2003');
               datenum('05/29/2003');
               datenum('10/31/2004');
               datenum('11/15/2007');
               datenum('11/15/2012');
```

```
                datenum('02/15/2031')];

RefCpn = [0;
          0;
          2.125;
          3;
          4;
          5.375] / 100;

RefPrices =  [99.6964;
              99.3572;
             100.3662;
              99.4511;
              99.4299;
             106.5756];

RefBonds = [RefPrices, RefMaturity, RefCpn];
Settle   = datenum('26-Nov-2002');
[ZeroRates, CurveDates] = zbtprice(RefBonds(:, 2:end), ...
RefPrices, Settle)

ZeroRates =

    0.0121
    0.0127
    0.0194
    0.0317
    0.0423
    0.0550


CurveDates =

      731639
      731730
      732251
      733361
      735188
      741854



% FNMA 4 3/8 maturing 10/06 at 4.30 pm Tuesday
Price    = 105.484;
Coupon   = 0.04375;
```

```
Maturity = datenum('15-Oct-2006');

% All optional inputs are supposed to be accounted by default,
% except the accrued interest under 30/360 (SIA), so:
Period = 2;
Basis  = 1;
SpotInfo = [CurveDates, ZeroRates];

% Compute static spread over treasury curve, taking into account
% the shape of curve as derived by bootstrapping method embedded
% within bndspread.

SpreadInBP = bndspread(SpotInfo, Price, Coupon, Settle, ...
Maturity, Period, Basis)

SpreadInBP = 18.5669


plot(CurveDates, ZeroRates*100, 'b', CurveDates, ...
ZeroRates*100+SpreadInBP/100, 'r--')
legend({'Treasury'; 'FNMA 4 3/8'})
xlabel('Curve Dates')
ylabel('Spot Rate [%]')
grid;
```



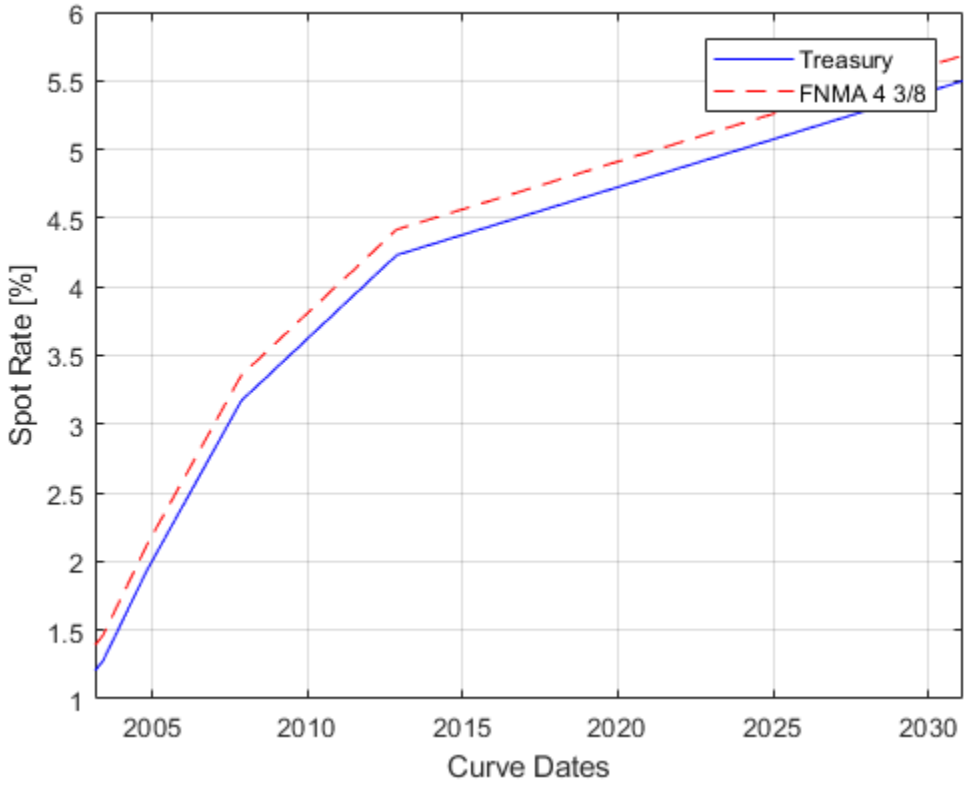
Compute the Static Spread Over a Spot Curve Using datetime Inputs

This example shows how to compute a Federal National Mortgage Association (FNMA) 4 3/8 spread over a Treasury spot curve using `datetime` inputs for `Settle` and `Maturity` and a table for `SpotInfo` and plot the results.

```
RefMaturity = [datenum('02/27/2003');
               datenum('05/29/2003');
               datenum('10/31/2004');
               datenum('11/15/2007');
               datenum('11/15/2012');
```

```
               datenum('02/15/2031')];

RefCpn = [0;
          0;
          2.125;
          3;
          4;
          5.375] / 100;

RefPrices =  [99.6964;
              99.3572;
             100.3662;
              99.4511;
              99.4299;
             106.5756];

RefBonds = [RefPrices, RefMaturity, RefCpn];
Settle   = datetime('26-Nov-2002','Locale','en_US');
[ZeroRates, CurveDates] = zbtprice(RefBonds(:, 2:end), ...
RefPrices, Settle)

ZeroRates =

    0.0121
    0.0127
    0.0194
    0.0317
    0.0423
    0.0550


CurveDates = 6x1 datetime array
   27-Feb-2003
   29-May-2003
   31-Oct-2004
   15-Nov-2007
   15-Nov-2012
   15-Feb-2031



% FNMA 4 3/8 maturing 10/06 at 4.30 pm Tuesday
Price    = 105.484;
Coupon   = 0.04375;
Maturity = datetime('15-Oct-2006','Locale','en_US');
```

```
% All optional inputs are accounted by default,
% except the accrued interest under 30/360 (SIA), so:
Period = 2;
Basis  = 1;

SpotInfo = table(datetime(CurveDates,'ConvertFrom','datenum','Locale','en_US'), ZeroRat

% 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)

SpreadInBP = 18.5669


plot(CurveDates, ZeroRates*100, 'b', CurveDates, ...
ZeroRates*100+SpreadInBP/100, 'r--')
legend({'Treasury'; 'FNMA 4 3/8'})
xlabel('Curve Dates')
ylabel('Spot Rate [%]')
grid;
```



- "Bond Portfolio for Hedging Duration and Convexity" on page 10-7
- "Pricing Functions" on page 2-35
- "Sensitivity of Bond Prices to Interest Rates" on page 10-3

Input Arguments

`SpotInfo` — Spot-rates information

matrix | table | term structure

Spot-rates information, specified as matrix of two columns, an annualized term structure created by `intenvset`, or a table.

- Matrix of two columns— The first column is the `SpotDate`, and the second column, `ZeroRates`, is the zero-rate corresponding to maturities on the `SpotDate`. It is recommended that the spot-rates are spaced as evenly apart as possible, perhaps one that is built from 3-months deposit rates. For example, using the 3-month deposit rates:

```
SpotInfo = ...
[datenum('2-Jan-2004') ,  0.03840;
datenum('2-Jan-2005') ,  0.04512;
datenum('2-Jan-2006') ,  0.05086];
```

- Annualized term structure — Refer to `intenvset` to create an annualized term structure. For example:

```
Settle = datenum('1-Jan-2004');
Rates = [0.03840; 0.04512; 0.05086];
EndDates = [datenum('2-Jan-2004'); datenum('2-Jan-2005');...
            datenum('2-Jan-2006')];
SpotInfo = intenvset('StartDates' , Settle  ,...
                     'Rates'      , Rates   ,...
                     'EndDates'   , EndDates,...
                     'Compounding', 2       ,...
                     'Basis'      , 0);
```

- Table — If `SpotInfo` is a table, the first column can be either a serial date number, date character vector, or datetime array. The second column is numerical data representing zero rates. For example:

```
ZeroRates = … [0.012067955808764;0.012730933424479;0.019360902068703;0.031704525214251;0.042306085224510;0.05498741534
CurveDates = [731639;731730;732251;733361;735188;741854];
Settle   = datenum('26-Nov-2002');
Price    = 105.484;
Coupon   = 0.04375;
Maturity = datenum('15-Oct-2006');
Period = 2;
Basis  = 1;
SpotInfo = table(datestr(CurveDates), ZeroRates);
```

Data Types: `double` | `table` | `struct`

`Price` — Price for every $100 notional amount of bonds whose spreads are computed

numeric

Price for every $100 notional amount of bonds whose spreads are computed, specified as numeric value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`Coupon` — Annual coupon rate of bonds whose spreads are computed
decimal

Annual coupon rate of bonds whose spreads are computed, specified as decimal value using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`Settle` — Settlement date of bond
serial date number | date character vector | datetime

Settlement date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of bond
serial date number | date character vector | datetime

Maturity date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Spread = bndspread(SpotInfo,Price,Coupon,Settle,Maturity,'Period',4,'Basis', 7)`

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag

`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `StartDate` is when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency for yield calculation
SIA bases uses`2`, ICMA bases uses `1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- 0 = actual/actual

- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the actual/actual day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the `Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period

`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. `LastCouponInterest` is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`
- `compound`

Data Types: `char` | `cell`

Output Arguments

`Spread` — Static spread to benchmark in basis points

numeric

Static spread to benchmark, returned in basis points as a scalar or a `NUMBONDS`-by-`1` vector.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`bndprice` | `bndyield` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7
“Pricing Functions” on page 2-35
“Sensitivity of Bond Prices to Interest Rates” on page 10-3
“Yield Conventions” on page 2-34

Introduced before R2006a

bndtotalreturn

Total return of fixed-coupon bond

Syntax

```
[BondEquiv,EffectiveRate] = bndtotalreturn(Price,CouponRate,Settle,
Maturity,ReinvestRate)
[BondEquiv,EffectiveRate] = bndtotalreturn(___,Name,Value)
```

Description

`[BondEquiv,EffectiveRate] = bndtotalreturn(Price,CouponRate,Settle,Maturity,ReinvestRate)` calculates the total return for fixed-coupon bonds to maturity or to a specific investment horizon.

`[BondEquiv,EffectiveRate] = bndtotalreturn(___,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Total Return of a Fixed-Coupon Bond

Use `bndtotalreturn` to compute the total return for a fixed-coupon bond, given an investment horizon date.

Define fixed-coupon bond.

```
Price = 101;
CouponRate = 0.05;
Settle = '15-Nov-2011';
Maturity = '15-Nov-2031';
ReinvestRate = 0.04;
```

Calculate the total return to maturity.

```
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...
Settle, Maturity, ReinvestRate)

BondEquiv = 0.0460

EffectiveRate = 0.0466
```

Specify an investment horizon.

```
HorizonDate = '15-Nov-2021';
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...
Settle, Maturity, ReinvestRate, 'HorizonDate', HorizonDate)

BondEquiv = 0.0521

EffectiveRate = 0.0528
```

Perform scenario analysis on the reinvestment rate.

```
ReinvestRate = [0.03; 0.035; 0.04; 0.045; 0.05];
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...
Settle, Maturity, ReinvestRate, 'HorizonDate', HorizonDate)

BondEquiv =

    0.0557
    0.0538
    0.0521
    0.0505
    0.0490


EffectiveRate =

    0.0565
    0.0546
    0.0528
    0.0511
    0.0496
```

Compute the Total Return of a Fixed-Coupon Bond Using datetime Inputs

Use `bndtotalreturn` with `datetime` inputs to compute the total return for a fixed-coupon bond, given an investment horizon date.

```
Price = 101;
CouponRate = 0.05;
Settle = datetime('15-Nov-2011','Locale','en_US');
Maturity = datetime('15-Nov-2031','Locale','en_US');
HorizonDate = datetime('15-Nov-2021','Locale','en_US');
ReinvestRate = 0.04;
[BondEquiv, EffectiveRate] = bndtotalreturn(Price, CouponRate, ...
Settle, Maturity, ReinvestRate, 'HorizonDate', HorizonDate)
```

```
BondEquiv = 0.0521
```

```
EffectiveRate = 0.0528
```


- "Bond Portfolio for Hedging Duration and Convexity" on page 10-7
- "Pricing Functions" on page 2-35
- "Sensitivity of Bond Prices to Interest Rates" on page 10-3

Input Arguments

`Price` — Clean price at settlement date

matrix

Clean price at the settlement date, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`CouponRate` — Coupon rate

decimal

Coupon rate, specified as a scalar or a `NINST`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date of fixed-coupon bond

serial date number | date character vector | datetime

Settlement date of the fixed-coupon bond, specified as scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. If supplied as a `NINST`-by-`1` vector of dates, settlement dates can be different, as long as they are before the `Maturity` date and `HorizonDate`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of fixed-coupon bond
serial date number | date character vector | datetime

Maturity date of the fixed-coupon bond, specified as scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`ReinvestRate` — Reinvestment rate
decimal

Reinvestment rate (the rate earned by reinvesting the coupons), specified as scalar or a `NINST`-by-`2` vector of decimal values.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
[BondEquiv,EffectiveRate] =
bndtotalreturn(Price,CouponRate,Settle,Maturity,ReinvestRate,'Horizo
nDate','15-Nov-2021')
```

`HorizonDate` — Investment horizon date
`Maturity` date (default) | serial date number | date character vector | datetime

Investment horizon date, specified as a scalar or a `NINST`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If `HorizonDate` is unspecified, the total return is calculated to `Maturity`.

Data Types: `double` | `char` | `datetime`

`HorizonPrice` — Price at investment horizon date
calculated based on `ReinvestRate` (default) | numeric

Price at investment horizon date, specified as a scalar or a `NINST`-by-`1` vector.

If `HorizonPrice` is unspecified, the price at the `HorizonDate` is calculated based on the `ReinvestRate`. If the `HorizonDate` equals the `Maturity` date, the `HorizonPrice` is ignored and the total return to maturity is calculated based on the `Face` value.

Data Types: `double`

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NINST`-by-`1` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Day-count basis, specified as a scalar or a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NINST`-by-`1` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NINST`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NINST`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NINST`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NINST`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency for yield calculation
integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NINST`-by-`1` vector.

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (0-7) and BUS/252 use a semiannual compounding convention and ICMA bases (8-12) use an annual compounding convention.

Data Types: double

DiscountBasis — Basis used to compute the discount factors for computing the yield
integers of the set [0...13] | vector of integers of the set [0...13]

Basis used to compute the discount factors for computing the yield, specified as a scalar or a NINST-by-1 vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note The default behavior is for SIA bases (0-7) to use the actual/actual day count to compute discount factors, and for ICMA day counts (8 – 12) and BUS/252 to use the specified DiscountBasis.

Data Types: double

Output Arguments

`BondEquiv` — Total return in bond equivalent basis
numeric

Total return in bond equivalent basis, returned as a `NUMBONDS`-by-`1` vector.

`EffectiveRate` — Total return in effective rate basis
numeric

Total return in effective rate basis, returned as a `NUMBONDS`-by-`1` vector.

References

[1] Fabozzi, Frank J., Mann, Steven V. *Introduction to Fixed Income Analytics: Relative Value Analysis, Risk Measures and Valuation*. John Wiley and Sons, New York, 2010.

See Also

`bndprice` | `bndyield` | `cfamounts` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7
“Pricing Functions” on page 2-35
“Sensitivity of Bond Prices to Interest Rates” on page 10-3
“Yield Conventions” on page 2-34

Introduced in R2012b

bndyield

Yield to maturity for fixed-income security

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `CompoundingFrequency`, `DiscountBasis`, and `LastCouponInterest`.

Syntax

```
Yield = bndyield(Price,CouponRate,Settle,Maturity)
Yield = bndyield(___,Name,Value)
```

Description

`Yield = bndyield(Price,CouponRate,Settle,Maturity)` given `NUMBONDS` bonds with SIA date parameters and clean prices (excludes accrued interest), returns the bond equivalent yields to maturity.

`Yield = bndyield(___,Name,Value)` adds optional name-value arguments.

Examples

Compute Yield to Maturity for a Treasury Bond

This example shows how to compute the yield of a Treasury bond at three different price values.

```
Price = [95; 100; 105];
CouponRate = 0.05;
Settle = '20-Jan-1997';
Maturity = '15-Jun-2002';
```

```
Period = 2;
Basis = 0;

Yield = bndyield(Price, CouponRate, Settle,...
Maturity, Period, Basis)

Yield =

    0.0610
    0.0500
    0.0396
```

Compute Yield to Maturity for a Treasury Bond Using datetime Inputs

This example shows how to use `datetime` inputs to compute the yield of a Treasury bond at three different price values.

```
Price = [95; 100; 105];
CouponRate = 0.05;
Settle = datetime('20-Jan-1997','Locale','en_US');
Maturity = datetime('15-Jun-2002','Locale','en_US');
Period = 2;
Basis = 0;
Yield = bndyield(Price, CouponRate, Settle,...
Maturity, Period, Basis)

Yield =

    0.0610
    0.0500
    0.0396
```

Compute the Yield of a Treasury Bond Using the Same Basis for Discounting and Generating the Cash Flows

Compute the yield of a Treasury bond.

```
Price = [95; 100; 105];
CouponRate = 0.0345;
Settle = '15-May-2016';
Maturity = '02-Feb-2026';
Period = 2;
Basis = 1;
format long

Yield = bndyield(Price,CouponRate,Settle,Maturity,Period,Basis)

Yield =
   0.040764403932618
   0.034482347625316
   0.028554719853118
```

Using the same data, compute the yield of a Treasury bond using the same basis for discounting and generating the cash flows.

```
DiscountBasis = 1;

Yield = bndyield(Price,CouponRate,Settle,Maturity,'Period',Period,'Basis',Basis, ...
'DiscountBasis',DiscountBasis)

Yield =
   0.040780176658036
   0.034495592361619
   0.028565614029497
```

- “Bond Portfolio for Hedging Duration and Convexity” on page 10-7
- “Pricing Functions” on page 2-35
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3

Input Arguments

`Price` — Clean price of the bond

numeric

Clean price of the bond (current price without accrued interest), specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond
decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal using a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`Settle` — Settlement date of bond
serial date number | date character vector | datetime

Settlement date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of bond
serial date number | date character vector | datetime

Maturity date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Yield = bndyield(Price,CouponRate,Settle,Maturity,'Period',4,'Basis',9)`

`Period` — Number of coupon payments per year
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count of the instrument, specified as scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag

`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.

- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond Issue date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of payments
serial date number | date character vector | datetime

Forward starting date of payments, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency for yield calculation
SIA uses `2`, ICMA uses `1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. Values are:

- 0 = actual/actual

- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the `actual/actual` day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the`Basis` input argument are used.

Data Types: `double`

`LastCouponInterest` — Compounding convention for computing yield of a bond in last coupon period

`compound` (default) | values are `simple` or `compound`

Compounding convention for computing the yield of a bond in the last coupon period, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector. This is based on only the last coupon and the face value to be repaid. Acceptable values are:

- `simple`

- `compound`

Data Types: `char` | `cell`

Output Arguments

`Yield` — Yield to maturity with semiannual compounding
numeric

Yield to maturity with semiannual compounding, returned as a `NUMBONDS`-by-`1` vector.

Definitions

Price and Yield Conventions

The `Price` and `Yield` are related to different formulae for SIA and ICMA conventions.

For SIA conventions, `Price` and `Yield` are related by the formula:

```
Price + Accrued Interest = sum(Cash_Flow*(1+Yield/2)^(-Time))
```

where the sum is over the bond's cash flows and corresponding times in units of semiannual coupon periods.

For ICMA conventions, the `Price` and `Yield` are related by the formula:

```
Price + Accrued Interest = sum(Cash_Flow*(1+Yield)^(-Time))
```

Algorithms

For SIA conventions, the following formula defines bond price and yield:

$$PV = \frac{CF}{(1+\frac{z}{f})^{TF}},$$

where:

PV =	Present value of a cash flow.
CF =	The cash flow amount.
z =	The risk-adjusted annualized rate or yield corresponding to a given cash flow. The yield is quoted on a semiannual basis.
f =	The frequency of quotes for the yield.
TF =	Time factor for a given cash flow. Time is measured in semiannual periods from the settlement date to the cash flow date. In computing time factors, use SIA `actual/actual` day count conventions for all time factor calculations.

For ICMA conventions, the frequency of annual coupon payments determines bond price and yield.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`bndprice` | `cfamounts` | `datetime`

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-7
“Pricing Functions” on page 2-35
“Sensitivity of Bond Prices to Interest Rates” on page 10-3
“Yield Conventions” on page 2-34

Introduced before R2006a

bolling

Bollinger band chart

Syntax

```
bolling(Asset,Samples,Alpha,Width)

[Movavgv,UpperBand,LowerBand] = bolling(Asset,Samples,Alpha,Width)
```

Arguments

`Asset`	Vector of asset data.
`Samples`	Number of samples to use in computing the moving average.
`Alpha`	(Optional) Exponent used to compute the element weights of the moving average. Default = `0` (simple moving average).
`Width`	(Optional) Number of standard deviations to include in the envelope. A multiplicative factor specifying how tight the bands should be around the simple moving average. Default = `2`.

Description

`bolling(Asset,Samples,Alpha,Width)` plots Bollinger bands for given `Asset` data. This form of the function does not return any data.

`[Movavgv,UpperBand,LowerBand] = bolling(Asset,Samples,Alpha,Width)` returns `Movavgv` with the moving average of the `Asset` data, `UpperBand` with the upper band data, and `LowerBand` with the lower band data. This form of the function does not plot any data.

Note The standard deviations are normalized by `N-1`, where `N` = the sequence length.

Examples

If `Asset` is a column vector of closing stock prices

```
bolling(Asset, 20, 1)
```

plots linear 20-day moving average Bollinger bands based on the stock prices.

```
[Movavgv, UpperBand, LowerBand] = bolling(Asset, 20, 1)
```

returns `Movavgv`, `UpperBand`, and `LowerBand` as vectors containing the moving average, upper band, and lower band data, without plotting the data.

See Also

`candle` | `dateaxis` | `highlow` | `movavg` | `movavg`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

bollinger

Time series Bollinger band

Syntax

```
[mid,uppr,lowr] = bollinger(data,wsize,wts,nstd)
[midfts,upprfts,lowrfts] = bollinger(tsobj,wsize,wts,nstd)
```

Arguments

`data`	Data vector. **Note** The input time series `data` must be ordered from newest to oldest.
`wsize`	(Optional) Window size. Default = `20`.
`wts`	(Optional) Weight factor. Determines the type of moving average used. Default = `0` (box). `1` = linear.
`nstd`	(Optional) Number of standard deviations for upper and lower bands. Default = `2`.
`tsobj`	Financial time series object.

Description

`[mid,uppr,lowr] = bollinger(data,wsize,wts,nstd)` calculates the middle (`mid`), upper (`uppr`), and lower (`lowr`) bands that make up the Bollinger bands from the vector `data`.

`mid` is the vector that represents the middle band, a simple moving average with a window size of `wsize`. `uppr` and `lowr` are vectors that represent the upper and lower bands. `uppr` is a vector representing the upper band that is +`nstd` times. `lowr` is a vector representing the lower band that is -`nstd` times.

`[midfts,upprfts,lowrfts] = bollinger(tsobj,wsize,wts,nstd)` calculates the middle, upper, and lower bands that make up the Bollinger bands from a financial time series object `tsobj`.

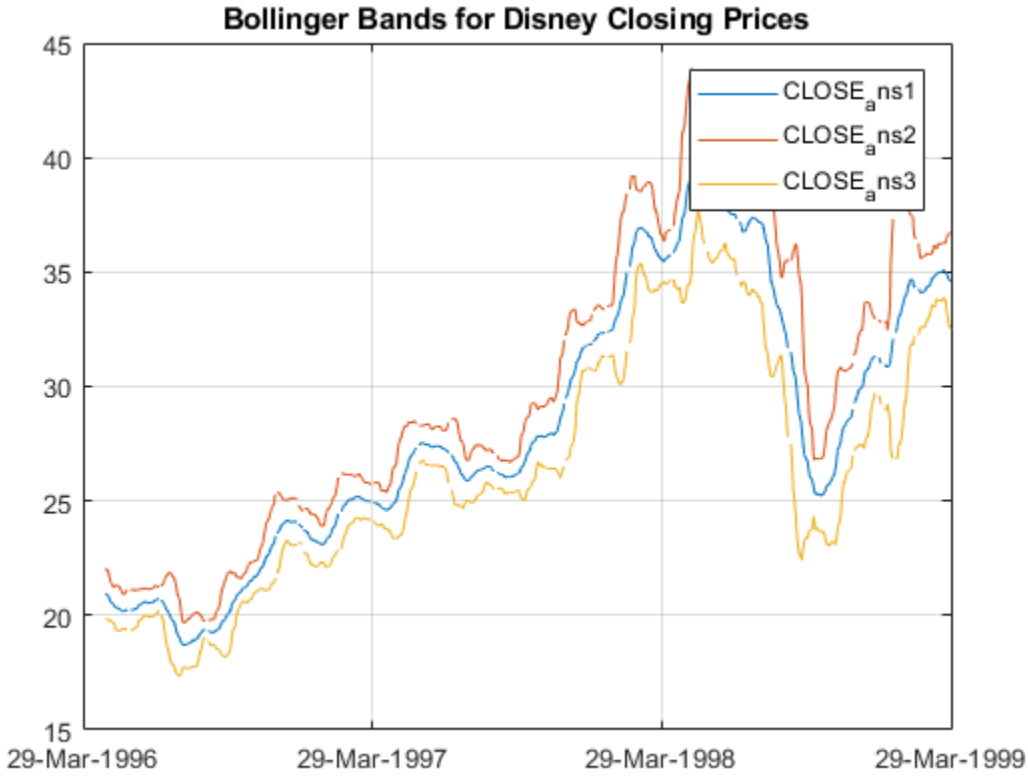
`midfts` is a financial time series object that represents the middle band for all series in `tsobj`. Both `upprfts` and `lowrfts` are financial time series objects that represent the upper and lower bands of all series, which are `+nstd` times and `-nstd` times moving standard deviations away from the middle band.

Examples

Create a Bollinger Bands Plot

This example shows how to compute the Bollinger bands for Disney stock closing prices and plot the results.

```
load disney.mat
[dis_Mid,dis_Uppr,dis_Lowr]= bollinger(dis);
dis_CloseBolling = [dis_Mid.CLOSE, dis_Uppr.CLOSE,...
dis_Lowr.CLOSE];
plot(dis_CloseBolling)
title('Bollinger Bands for Disney Closing Prices')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 72–74.

See Also

`tsmovavg`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

boxcox

Box-Cox transformation

Syntax

```
[transdat,lambda] = boxcox(data)

[transfts,lambda] = boxcox(tsobj)

transdat = boxcox(lambda,data)

transfts = boxcox(lambda,tsobj)
```

Arguments

`data`	Data vector. Must be positive and specified as a column data vector.
`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.

If λ is not = `0`, then

$$data(\lambda) = \frac{data^{\lambda} - 1}{\lambda}$$

If λ is = `0`, then

$$data(\lambda) = \log(data)$$

The logarithm is the natural logarithm (log base e). The algorithm calls for finding the λ value that maximizes the Log-Likelihood Function (LLF). The search is conducted using `fminsearch`.

`[transdat,lambda] = boxcox(data)` transforms the data vector `data` using the Box-Cox transformation method into `transdat`. It also estimates the transformation parameter λ.

`[transfts,lambda] = boxcox(tsojb)` transforms the financial time series object `tsobj` using the Box-Cox transformation method into `transfts`. It also estimates the transformation parameter λ.

If the input data is a vector, `lambda` is a scalar. If the input is a financial time series object, `lambda` is a structure with fields similar to the components of the object; for example, if the object contains series names `Open` and `Close`, `lambda` has fields `lambda.Open` and `lambda.Close`.

`transdat = boxcox(lambda, data)` and `transfts = boxcox(lambda, 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.

Examples

Transform a Data Series Contained in a Financial Times Series Object

Use `boxcox` to transform the data series contained in a financial time series object into another set of data series with relatively normal distributions.

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 `NaN`'s in whrl with values calculated using the linear method.

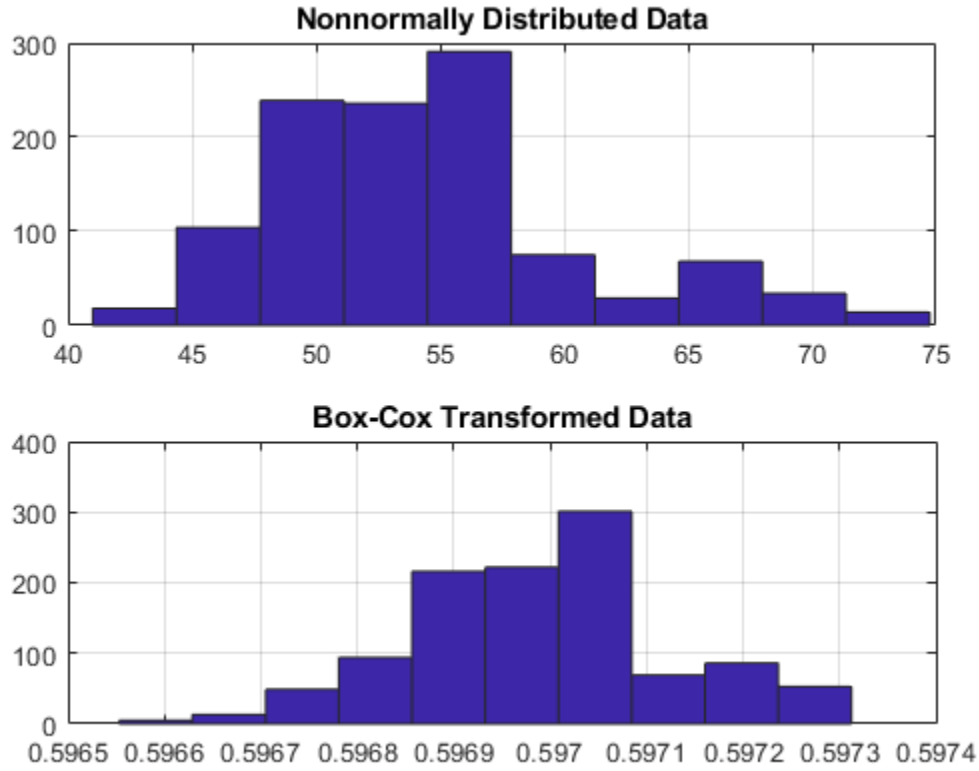
```
f_whrl = fillts(whrl);
```

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 and 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');
```



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 toward the left (not normally distributed). The bar chart on the bottom is less skewed to the left. If you plot a Gaussian probability distribution function (PDF) with similar mean and standard deviation, the distribution of the transformed data is very close to normal (Gaussian).

When you examine the contents of the resulting object `bc_whrl`, you find an identical object to the original object `whrl` but the contents are the transformed data series.

- "Data Transformation and Frequency Conversion" on page 12-12
- "Using Time Series to Predict Equity Return" on page 12-25

See Also

`fminsearch`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

busdate

Next or previous business day

Syntax

```
Busday = busdate(Date)
Busday = busdate( ___ ,DirFlag,Holiday,Weekend)
```

Description

`Busday = busdate(Date)` returns the scalar, vector, or matrix of the next or previous business days, depending on the definition for `Holiday`.

`Busday = busdate( ___ ,DirFlag,Holiday,Weekend)`. returns the scalar, vector, or matrix of the next or previous business days, depending on the optional input arguments, including `Holiday`.

If both `Date` and `Holiday` are either serial date numbers or date character vectors, `Busday` is returned as a serial date number.

However, if either `Date` or `Holiday` are datetime arrays, `Busday` is returned as a datetime array.

Use the function `datestr` to convert serial date numbers to formatted date character vectors.

Examples

Determine Business Days

Determine the next business day when `Date` is a character vector.

```
Busday = busdate('3-Jul-2001', 1)
```

```
Busday = 731037

datestr(Busday)

ans =
'05-Jul-2001'
```

Indicate that Saturday is a business day by appropriately setting the `Weekend` argument. July 4, 2003 falls on a Friday. Use `busdate` to verify that Saturday, July 5, is actually a business day.

```
Weekend = [1 0 0 0 0 0 0];
Date = datestr(busdate('3-Jul-2003', 1, [], Weekend))

Date =
'05-Jul-2003'
```

If either `Date` or `Holiday` are datetime arrays, `Busday` is returned as a datetime array.

```
Busday = busdate(datetime('3-Jul-2001','Locale','en_US'), 1)

Busday = datetime
   05-Jul-2001
```

- "Handle and Convert Dates" on page 2-2

Input Arguments

`Date` — Reference business date

serial date number | date character vector | datetime object

Reference business date, specified as a scalar, vector, or matrix using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`DirFlag` — Business day convention

`follow` (default) | date character vector with values of `follow`, `modifiedfollow`, `previous`, or `modifiedprevious` | cell array of date character vectors with values of `follow`, `modifiedfollow`, `previous`, or `modifiedprevious`

Business day convention, specified date character vector or cell array of date character vectors with values of `follow`, `modifiedfollow`, `previous`, or `modifiedprevious`.

Also, `DirFlag` can be a scalar, vector, or matrix of search directions, where Next is `DIREC = 1` (default) or Previous is `DIREC = -1`.

Data Types: `double` | `char` | `datetime`

`Holiday` — Holidays and nontrading-day dates
non-trading day vector is determined by the routine `holidays` (default) | serial date number | date character vector | datetime object

Holidays and nontrading-day dates, specified as vector.

All dates in `Holiday` must be the same format: either serial date numbers, or date character vectors, or datetime arrays. (Using serial date numbers improves performance.)

Data Types: `double` | `char` | `datetime`

`Weekend` — Weekend days
`[1 0 0 0 0 0 1]` (Saturday and Sunday form the weekend) (default) | vector of length 7, containing `0` and `1`, where `1` indicates weekend days

Weekend days, specified as a vector of length 7, containing `0` and `1`, where `1` indicates weekend days and the first element of this vector corresponds to Sunday.

Data Types: `double`

Output Arguments

`Busday` — Next or previous business day
scalar | vector | matrix

Next or previous business day, returned as a scalar, vector, or matrix depending on the definition for `Holiday`. If `Date` is a datetime array, `Busday` returns a datetime array. Otherwise, `Busday` returns a serial date numbers.

See Also

`datetime` | `holidays`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2

Introduced before R2006a

busdays

Business days for given period

Syntax

```
bdates = busdays(sdate,edate)
bdates = busdays(___,bdmode,holvec)
```

Description

`bdates = busdays(sdate,edate)` generates a vector of business days between the last business date of the period that contains the start date (`sdate`), and the last business date of period that contains the end date (`edate`).

`bdates = busdays(___,bdmode,holvec)` generates a vector of business days between the last business date of the period that contains the start date (`sdate`), and the last business date of period that contains the end date (`edate`) using optional input arguments. If `holvec` is not supplied, the dates are generated based on United States holidays. If you do not supply `bdmode`, `bdates` generates a daily vector.

Examples

Determine Business Days for a Given Period

Determine the business days for a weekly period.

```
 bdates = datestr(busdays('1/2/01','1/9/01','weekly'))
```

```
bdates = 2x11 char array
    '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 the following week, the following Friday (1/12/01) is also reported.

Determine the business days for a weekly period using a datetime input for `sdate`.

```
bdates = busdays(datetime('2-Jan-2001','Locale','en_US'),'9-Jan-2001','weekly')
```

```
bdates = 2x1 datetime array
   05-Jan-2001
   12-Jan-2001
```

Determine the business days for a monthly period.

```
vec = datestr(busdays('1/8/16','3/1/16','monthly'))
```

```
vec = 3x11 char array
    '29-Jan-2016'
    '29-Feb-2016'
    '31-Mar-2016'
```

The start date (1/8/16) is in the month of January, 2016. The last business day for the month of January is 1/29/16 (Friday). The end date (3/1/16) is in the month of March, 2016. The last business day for the month of March is 3/31/16 (Thursday). The month of February, 2016 lies between the start date and the end date. The last business day for the month of February is 2/29/16 (Monday).

- “Handle and Convert Dates” on page 2-2

Input Arguments

`sdate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`edate` — End date

serial date number | date character vector | datetime object

End date, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`bdmode` — Frequency of business days

`DAILY` (`1`) (default) | nonnegative numeric with values `1` through `5` | date character vector with values `DAILY`, `WEEKLY`, `MONTHLY`, `QUARTERLY`, `SEMIANNUAL` or `ANNUAL`

Frequency of business days, specified as a nonnegative numeric with values `1` through `5` or date character vector with values of `DAILY`, `WEEKLY`, `MONTHLY`, `QUARTERLY`, `SEMIANNUAL`, or `ANNUAL`

Valid periodicities include:

- `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`

  Character vectors must be enclosed in single quotation marks.

For example, if `bdmode` is set to `monthly`, `busdays` returns end-of-month business dates for all full or partial months between the start date and end date inclusive.

Data Types: `double` | `char`

`holvec` — Holiday dates

if `holvec` is `[ ]` holiday dates used are based on United States holidays (default) | serial date number | date character vector | datetime object

Holiday dates, specified as a vector in date character vector, serial date, or datetime array format. If you specify `holvec`, you must also supply the frequency `bdmode`. Using a `holvec` value of `NaN` uses a holiday list that has no dates.

Data Types: `double` | `char` | `datetime`

Output Arguments

`bdates` — Business days

column vector

Business days, returned as a column vector of business dates, in serial date format (default) or datetime format (if `sdate`, `edate`, or `holvec` are in datetime format). Business dates can exist before and/or after the specified `sdate` and `edate`.

See Also

`datetime` | `holidays`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2

Introduced before R2006a

candle

Candlestick chart

Syntax

```
candle(HighPrices,LowPrices,ClosePrices,OpenPrices)

candle(HighPrices,LowPrices,ClosePrices,OpenPrices,Color,Dates,Dateform)
```

Arguments

`HighPrices`	High prices for a security. A column vector.
`LowPrices`	Low prices for a security. A column vector.
`ClosePrices`	Closing prices for a security. A column vector.
`OpenPrices`	Opening prices for a security. A column vector.
`Color`	(Optional) Candlestick color is specified as a character vector. MATLAB software supplies a default color if none is specified. The default color differs depending on the background color of the figure window. See `ColorSpec` for color names.
`Dates`	(Optional) Column vector of dates for user specified X-axis tick labels. Date is specified as a serial date number or datetime array.
`Dateform`	(Optional) Date character vector format used as the *x*-axis tick labels. (See `datetick`.) 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(HighPrices,LowPrices,ClosePrices,OpenPrices)` plots a candlestick chart given column vectors with the high, low, closing, and opening prices of a security.

If the closing price is greater than the opening price, the body (the region between the opening and closing price) is unfilled.

If the opening price is greater than the closing price, the body is filled.

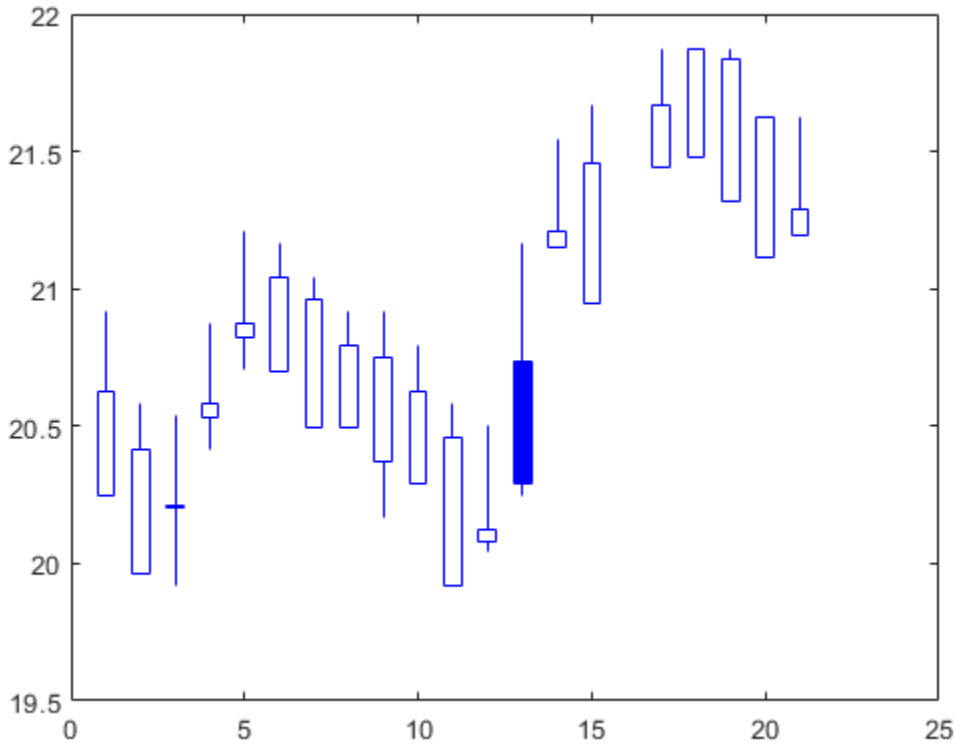
`candle(HighPrices,LowPrices,ClosePrices,OpenPrices,Color,Dates,Dateform)` plots a candlestick chart given column vectors with the high, low, closing, and opening prices of a security. In addition, the optional arguments `Color`, `Dates`, and `Dateform` specify the color of the candle box and the date character vector format used as the *x*-axis tick labels.

Examples

Create a Candlestick Chart

This example shows how to create a candlestick chart, with blue candles, for the most recent 21 days in disney.mat.

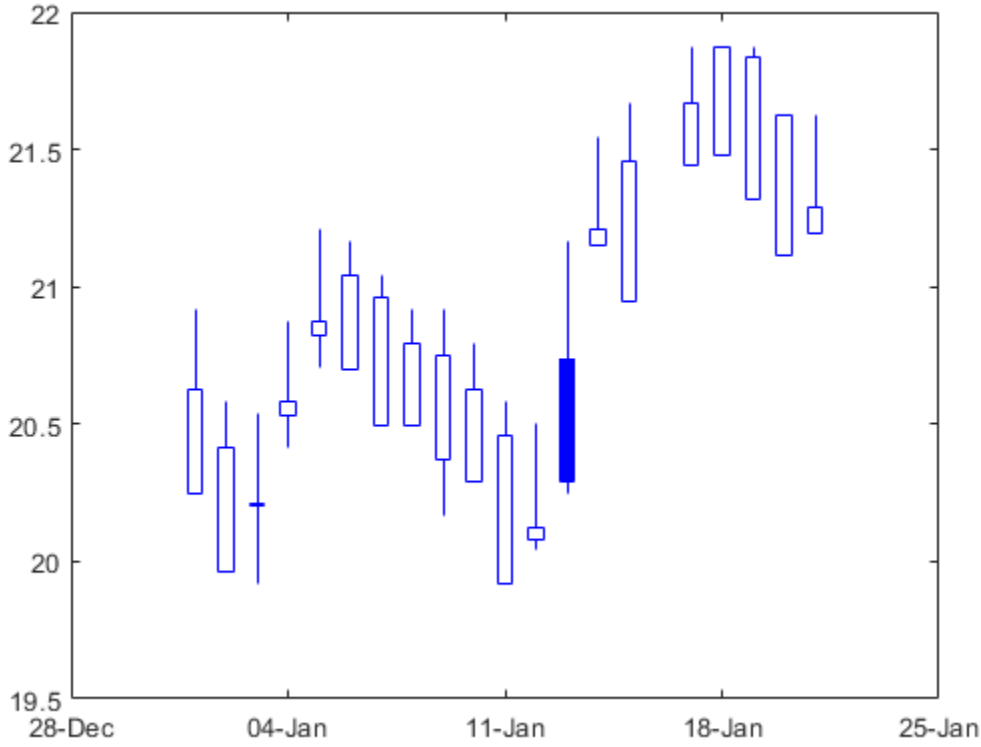
```
load disney;
candle(dis_HIGH(end-20:end), dis_LOW(end-20:end), dis_CLOSE(end-20:end),...
dis_OPEN(end-20:end), 'b');
```



Create a Candlestick Chart Using datetime Input

This example shows how to create a candlestick chart using `datetime` input, with blue candles, for the most recent 21 days in `disney.mat`.

```
load disney;
dates=datetime(2015,1,1:21);
candle(dis_HIGH(end-20:end), dis_LOW(end-20:end), dis_CLOSE(end-20:end),...
dis_OPEN(end-20:end), 'b',dates','dd-mmm');
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `dateaxis` | `datetime` | `highlow` | `movavg` | `pointfig`

Topics

“Charting Financial Data” on page 2-14

Introduced before R2006a

candle (fts)

Time series candle plot

Syntax

```
candle(tsobj)

candle(tsobj,color)

candle(tsobj,color,dateform)

candle(tsobj,color,dateform,'ParameterName',ParameterValue, ...)

hcdl = candle(tsobj,color,dateform,'ParameterName',ParameterValue, ...)
```

Arguments

`tsobj`	Financial time series object
`color`	(Optional) A three-element row vector representing RGB or a color identifier. (See `plot`.)
`dateform`	(Optional) Date character vector format used as the *x*-axis tick labels. (See `datetick`.) 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 character vector format used as the *x*-axis tick labels. See `datestr` for a list of date character vector formats.

`candle(tsobj,color,dateform,'ParameterName',ParameterValue, ...)` indicates the actual names 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

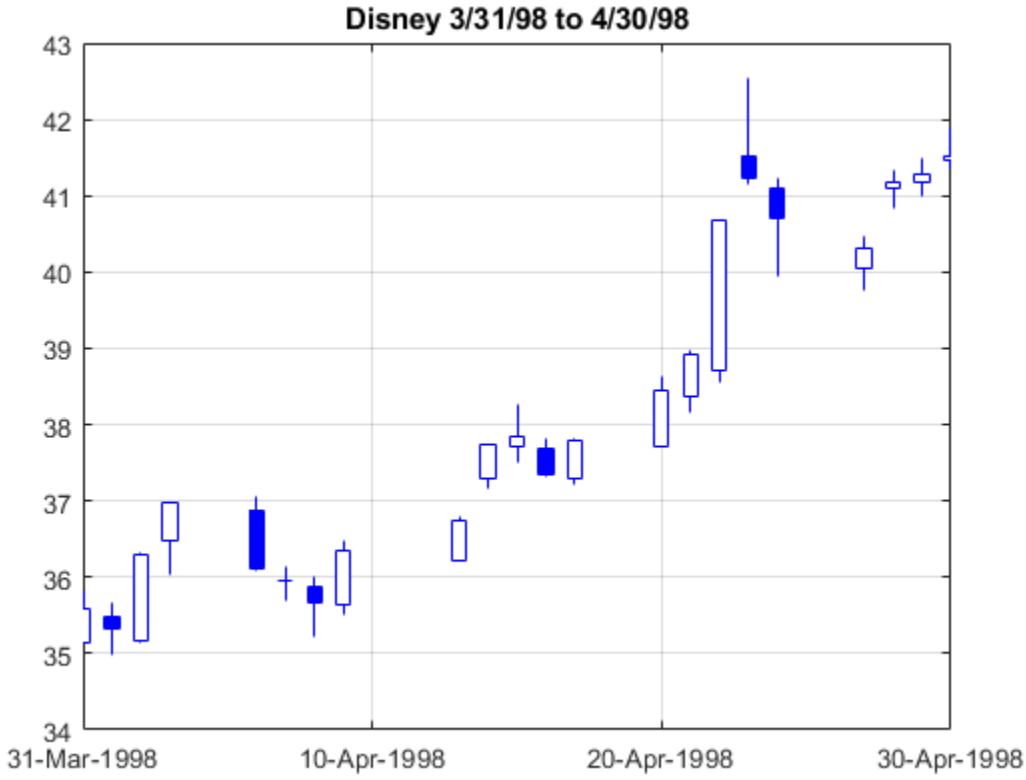
`hcdl = candle(tsobj, color, dateform, 'ParameterName', ParameterValue, ...)` returns the handle to the patch objects and the line object that make up the candle plot. `hdcl` is a three-element column vector representing the handles to the two patches and one line that forms the candle plot.

Examples

Create a Candle Plot for a Financial Time Series Object

This example shows how to create a candle plot for Disney stock for the dates March 31, 1998 through April 30, 1998.

```
load disney.mat
candle(dis('3/31/98::4/30/98'))
title('Disney 3/31/98 to 4/30/98')
```



- “Technical Analysis Examples” on page 16-4

See Also

`candle` | `chartfts` | `highlow` | `plot`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

cdai

Accrued interest on certificate of deposit

Syntax

```
AccrInt = cdai(CouponRate,Settle,Maturity,IssueDate)
AccrInt = cdai(___,Basis)
```

Description

`AccrInt = cdai(CouponRate,Settle,Maturity,IssueDate)` computes the accrued interest on a certificate of deposit.

`cdai` assumes that the certificates of deposit pay interest at maturity. Because of the simple interest treatment of these securities, this function is best used for short-term maturities (less than 1 year). The default simple interest calculation uses the `Basis` for the actual/360 convention (`2`).

`AccrInt = cdai(___,Basis)` adds an optional argument for `Basis`.

Examples

Find the Accrued Interest on a Certificate of Deposit

This example shows how to compute the accrued interest due, given a certificate of deposit with the following characteristics.

```
CouponRate      =  0.05;
Settle          =  '02-Jan-02';
Maturity        =  '31-Mar-02';
IssueDate       =  '1-Oct-01';

AccrInt = cdai(CouponRate, Settle, Maturity, IssueDate)
```

```
AccrInt = 1.2917
```

Find the Accrued Interest on a Certificate of Deposit Using datetime Inputs

This example shows how to use `datetime` inputs to compute the accrued interest due, given a certificate of deposit with the following characteristics.

```
CouponRate =  0.05;
Settle =  datetime('02-Jan-02','Locale','en_US');
Maturity =  datetime('31-Mar-02','Locale','en_US');
IssueDate =  datetime('1-Oct-01','Locale','en_US');
AccrInt = cdai(CouponRate, Settle, Maturity, IssueDate)
```

```
AccrInt = 1.2917
```

- “Coupon Date Calculations” on page 2-34

Input Arguments

`CouponRate` — Annual interest rate

decimal

Annual interest rate, specified as decimal using a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

Data Types: `double`

`Settle` — Settlement date for certificate of deposit

serial date number | date character vector | datetime

Settlement date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date for certificate of deposit

serial date number | date character vector | datetime

Maturity date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`IssueDate` — Issue date for certificate of deposit
serial date number | date character vector | datetime

Issue date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Basis` — Day-count basis for certificate of deposit
`2` (actual/360) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis for the certificate of deposit, specified as a scalar or a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`AccrInt` — Accrued interest per $100 of face value
numeric

Accrued interest per $100 of face value, returned as a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

See Also

`accrfrac` | `bndyield` | `cdprice` | `cdyield` | `datetime` | `stepcpnyield` | `tbillyield` | `zeroyield`

Topics

“Coupon Date Calculations” on page 2-34
“Yield Conventions” on page 2-34

Introduced before R2006a

cdprice

Price of certificate of deposit

Syntax

```
[Price,AccrInt] = cdprice(Yield,CouponRate,Settle,Maturity,
IssueDate)
[PriceAccrInt] = cdprice( ___ ,Basis)
```

Description

`[Price,AccrInt] = cdprice(Yield,CouponRate,Settle,Maturity, IssueDate)` computes the price of a certificate of deposit given its yield.

`cdprice` assumes that the certificates of deposit pay interest at maturity. Because of the simple interest treatment of these securities, this function is best used for short-term maturities (less than 1 year). The default simple interest calculation uses the `Basis` for the actual/360 convention (`2`).

`[PriceAccrInt] = cdprice( ___ ,Basis)` adds an optional argument for `Basis`.

Examples

Compute the Price and Accrued Interest for a Certificate of Deposit

This example shows how to compute the price and the accrued interest due on the settlement date, given a certificate of deposit with the following characteristics.

```
Yield           =  0.0525;
CouponRate      =  0.05;
Settle          =  '02-Jan-02';
Maturity        =  '31-Mar-02';
IssueDate       =  '1-Oct-01';
```

```
[Price, AccruedInt] = cdprice(Yield, CouponRate, Settle, ...
Maturity, IssueDate)

Price = 99.9233

AccruedInt = 1.2917
```

Compute the Price and Accrued Interest for a Certificate of Deposit Using datetime Inputs

This example shows how to use `datetime` inputs to compute the price and the accrued interest due on the settlement date, given a certificate of deposit with the following characteristics.

```
Yield =  0.0525;
CouponRate =  0.05;
Settle =  datetime('02-Jan-02','Locale','en_US');
Maturity =  datetime('31-Mar-02','Locale','en_US');
IssueDate =  datetime('1-Oct-01','Locale','en_US');

[Price, AccruedInt] = cdprice(Yield, CouponRate, Settle, ...
Maturity, IssueDate)

Price = 99.9233

AccruedInt = 1.2917
```

- “Coupon Date Calculations” on page 2-34

Input Arguments

`Yield` — Simple yield to maturity over basis denominator

numeric

Simple yield to maturity over the basis denominator, specified as a numeric value using a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

Data Types: `double`

`CouponRate` — Coupon annual interest rate

decimal

Coupon annual interest rate, specified as decimal using a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

Data Types: `double`

`Settle` — Settlement date for certificate of deposit
serial date number | date character vector | datetime

Settlement date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date for certificate of deposit
serial date number | date character vector | datetime

Maturity date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`IssueDate` — Issue date for certificate of deposit
serial date number | date character vector | datetime

Issue date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Basis` — Day-count basis for certificate of deposit
`2` (actual/360) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis for the certificate of deposit, specified as a scalar or a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`Price` — Clean price of certificate of deposit per $100

numeric

Clean price of the certificate of deposit per $100, returned as a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

`AccrInt` — Accrued interest payable at settlement per unit of face value

numeric

Accrued interest payable at settlement per unit of face value, returned as a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

See Also

`bndprice` | `cdai` | `cdyield` | `datetime` | `stepcpnprice` | `tbillprice`

Topics

“Coupon Date Calculations” on page 2-34
“Yield Conventions” on page 2-34

Introduced before R2006a

cdsbootstrap

Bootstrap default probability curve from credit default swap market quotes

Syntax

```
[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle)
[ProbData,HazData] = cdsbootstrap(___,Name,Value)
```

Description

`[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle)` bootstraps the default probability curve using credit default swap (CDS) market quotes. The market quotes can be expressed as a list of maturity dates and corresponding CDS market spreads, or as a list of maturities and corresponding upfronts and standard spreads for standard CDS contracts. The estimation uses the standard model of the survival probability.

`[ProbData,HazData] = cdsbootstrap(___,Name,Value)` adds optional name-value pair arguments.

Examples

Bootstrap Default Probability Curve from Credit Default Swap Market Quotes

This example shows how to use `cdsbootstrap` with market quotes for CDS contracts to generate `ProbData` and `HazData` values.

```
Settle = '17-Jul-2009'; % valuation date for the CDS
Spread_Time = [1 2 3 5 7]';
Spread = [140 175 210 265 310]';
Market_Dates = daysadd(datenum(Settle),360*Spread_Time,1);
MarketData = [Market_Dates Spread];
Zero_Time = [.5 1 2 3 4 5]';
Zero_Rate = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
```

```
Zero_Dates = daysadd(datenum(Settle),360*Zero_Time,1);
ZeroData = [Zero_Dates Zero_Rate];

format long
[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle)

ProbData =
   1.0e+05 *

   7.343360000000000   0.000000233427859
   7.347010000000000   0.000000575839968
   7.350670000000000   0.000001021397017
   7.357970000000000   0.000002064539982
   7.365280000000000   0.000003234110940


HazData =
   1.0e+05 *

   7.343360000000000   0.000000232959886
   7.347010000000000   0.000000352000512
   7.350670000000000   0.000000476383354
   7.357970000000000   0.000000609055766
   7.365280000000000   0.000000785241515
```

- “Bootstrapping a Default Probability Curve” on page 8-108
- “Bootstrapping from Inverted Market Curves” on page 8-120

Input Arguments

`ZeroData` — Zero rate data

vector | `IRDataCurve` object

Zero rate data, specified as a `M`-by-`2` vector of dates and zero rates or an `IRDataCurve` object of zero rates.

When `ZeroData` is an `IRDataCurve` object, `ZeroCompounding` and `ZeroBasis` are implicit in `ZeroData` and are redundant inside this function. In this case, specify these optional parameters when constructing the `IRDataCurve` object before using the `cdsbootstrap` function.

For more information on an `IRDataCurve` object, see "Creating an IRDataCurve Object" (Financial Instruments Toolbox).

Data Types: `double`

`MarketData` — Bond market data
matrix

Bond market data, specified as a `N`-by-`2` matrix of dates and corresponding market spreads or `N`-by-`3` matrix of dates, upfronts, and standard spreads of CDS contracts. The dates must be entered as serial date numbers, upfronts must be numeric values between `0` and `1`, and spreads must be in basis points.

Data Types: `double`

`Settle` — Settlement date
serial date number | date character vector

Settlement date, specified as a serial date number or a date character vector. The `Settle` date must be earlier than or equal to the dates in `MarketData`

Data Types: `double` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note Any optional input of size `N`-by-`1` is also acceptable as an array of size `1`-by-`N`, or as a single value applicable to all contracts. Single values are internally expanded to an array of size `N`-by-`1`.

Example: `[ProbData,HazData] = cdsbootstrap(ZeroData,MarketData,Settle,'RecoveryRate',Recovery,'ZeroCompounding',-1)`

`RecoveryRate` — Recovery rate
`0.4` (default) | decimal

Recovery rate, specified as a `N`-by-`1` vector of recovery rates, specified as a decimal from `0` to `1`.

Data Types: `double`

`Period` — Premium payment frequency
`4` (default) | numeric with values `1`, `2`, `3`, `4`, `6` or `12`

Premium payment frequency, specified as a `N`-by-`1` vector with values of `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of contract
`2` (actual/360) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Day-count basis of the contract, specified as a positive integer using a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`BusDayConvention` — Business day conventions
`'actual'` (default) | character vector

Business day conventions, specified by a character vector. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `'actual'` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char`

`PayAccruedPremium` — Flag for accrued premiums paid upon default
`true` (default) | integer with value `1` or `0`

Flag for accrued premiums paid upon default, specified as a `N`-by-`1` vector of Boolean flags that is `true` (default) if accrued premiums are paid upon default, `false` otherwise.

Data Types: `logical`

`TimeStep` — Number of days as time step for numerical integration
`10` (days) (default) | nonnegative integer

Number of days to take as time step for the numerical integration, specified as a nonnegative integer.

Data Types: `double`

`ZeroCompounding` — Compounding frequency of the zero curve
`2` (semiannual) (default) | integer with value of `1`,`2`,`3`,`4`,`6`,`12`, or `-1`

Compounding frequency of the zero curve, specified using values:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding
- `-1` — Continuous compounding

Data Types: `double`

`ZeroBasis` — Basis of the zero curve
`0` (actual/actual) (default) | integer with value of `0` to `13`

Basis of the zero curve, where the choices are identical to `Basis`.

Data Types: `double`

`ProbDates` — Dates for probability data
column of dates in `MarketData` (default) | serial date number | date character vector

Dates for probability data, specified as a `P`-by-`1` vector of dates, given as serial date numbers or date character vectors.

Data Types: `double` | `char`

Output Arguments

`ProbData` — Default probability values
matrix

Default probability values, returned as a `P`-by-`2` matrix with dates and corresponding cumulative default probability values. The dates match those in `MarketData`, unless the optional input parameter `ProbDates` is provided.

`HazData` — Hazard rate values
matrix

Hazard rate values, returned as a `N`-by-`2` matrix with dates and corresponding hazard rate values for the survival probability model. The dates match those in `MarketData`.

Note A warning is displayed when non-monotone default probabilities (that is, negative hazard rates) are found.

Algorithms

If the time to default is denoted by τ, the default probability curve, or function, *PD(t)*, and its complement, the survival function *Q(t)*, are given by:

$$PD(t) = P[\tau \le t] = 1 - P[\tau > t] = 1 - Q(t)$$

In the standard model, the survival probability is defined in terms of a piecewise constant hazard rate *h(t)*. For example, if *h(t)* =

λ_1, for $0 \le t \le t_1$

λ_2, for $t_1 < t \le t_2$

λ_3, for $t_2 < t$

then the survival function is given by *Q(t)* =

$e^{-\lambda_1 t}$, for $0 \le t \le t_1$

$e^{-\lambda_1 t - \lambda_2 (t - t_1)}$, for $t_1 < t \le t_2$

$e^{-\lambda_1 t_1 - \lambda_2 (t_2 - t_1) - \lambda_3 (t - t_2)}$, for $t_2 < t$

Given n market dates $t_1,...,t_n$ and corresponding market CDS spreads $S_1,...,S_n$, `cdsbootstrap` calibrates the parameters $\lambda_1,...,\lambda_n$ and evaluates *PD(t)* on the market dates, or an optional user-defined set of dates.

References

[1] Beumee, J., D. Brigo, D. Schiemert, and G. Stoyle. *"Charting a Course Through the CDS Big Bang."* Fitch Solutions, Quantitative Research, Global Special Report. April 7, 2009.

[2] Hull, J., and A. White. "Valuing Credit Default Swaps I: No Counterparty Default Risk." *Journal of Derivatives.* Vol. 8, pp. 29–40.

[3] O'Kane, D. and S. Turnbull. *"Valuation of Credit Default Swaps."* Lehman Brothers, Fixed Income Quantitative Credit Research, April 2003.

See Also

`IRDataCurve` | `cdsprice` | `cdsrpv01` | `cdsspread`

Topics

"Bootstrapping a Default Probability Curve" on page 8-108
"Bootstrapping from Inverted Market Curves" on page 8-120
"Credit Default Swap (CDS)" on page 8-107

Introduced in R2010b

cdsprice

Determine price for credit default swap

Syntax

```
[Price,AccPrem,PaymentDates,PaymentTimes,PaymentCF] = cdsprice(
ZeroData,ProbData,Settle,Maturity,ContractSpread)
[Price,AccPrem,PaymentDates,PaymentTimes,PaymentCF] = cdsprice(___,
Name,Value)
```

Description

`[Price,AccPrem,PaymentDates,PaymentTimes,PaymentCF] = cdsprice(ZeroData,ProbData,Settle,Maturity,ContractSpread)` computes the price, or the mark-to-market value for CDS instruments.

`[Price,AccPrem,PaymentDates,PaymentTimes,PaymentCF] = cdsprice(___,Name,Value)` adds optional name-value pair arguments.

Examples

Determine the Price for a Credit Default Swap

This example shows how to use `cdsprice` to compute the clean price for a CDS contract using the following data.

```
Settle = '17-Jul-2009'; % valuation date for the CDS
Zero_Time = [.5 1 2 3 4 5]';
Zero_Rate = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
Zero_Dates = daysadd(Settle,360*Zero_Time,1);
ZeroData = [Zero_Dates Zero_Rate];

ProbData = [daysadd(datenum(Settle),360,1), 0.0247];
Maturity = '20-Sep-2010';
```

```
ContractSpread = 135;

[Price,AccPrem] = cdsprice(ZeroData,ProbData,Settle,Maturity,ContractSpread)

Price = 1.5461e+04

AccPrem = 10500
```

- “Finding Breakeven Spread for New CDS Contract” on page 8-111
- “Valuing an Existing CDS Contract” on page 8-114
- “Converting from Running to Upfront” on page 8-117

Input Arguments

`ZeroData` — Zero rate data

vector | `IRDataCurve` object

Zero rate data, specified as a `M`-by-`2` vector of dates and zero rates or an `IRDataCurve` object of zero rates.

When `ZeroData` is an `IRDataCurve` object, `ZeroCompounding` and `ZeroBasis` are implicit in `ZeroData` and are redundant inside this function. In this case, specify these optional parameters when constructing the `IRDataCurve` object before using the `cdsprice` function.

For more information on an `IRDataCurve` object, see “Creating an IRDataCurve Object” (Financial Instruments Toolbox).

Data Types: `double` | `struct`

`ProbData` — Default probability values

matrix

Default probability values, specified as a `P`-by-`2` matrix with dates and corresponding cumulative default probability values.

Data Types: `double` | `char`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a `N`-by-`1` vector of serial date numbers or date character vectors. The `Settle` date must be earlier than or equal to the dates in `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date
serial date number | date character vector

Maturity date, specified as a `N`-by-`1` vector of serial date numbers or date character vectors.

Data Types: `double` | `char`

`ContractSpread` — Contract spreads
numeric

Contract spreads, specified as a `N`-by-`1` vector of spreads, expressed in basis points.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note Any optional input of size `N`-by-`1` is also acceptable as an array of size `1`-by-`N`, or as a single value applicable to all contracts. Single values are internally expanded to an array of size `N`-by-`1`.

Example: `[Price,AccPrem] = cdsprice(ZeroData,ProbData,Settle,Maturity,ContractSpread,'Basis', 7,'BusDayConvention','previous')`

`RecoveryRate` — Recovery rate
`0.4` (default) | decimal

Recovery rate, specified as a `N`-by-`1` vector of recovery rates, specified as a decimal from `0` to `1`.

Data Types: `double`

`Period` — Premium payment frequency
`4` (default) | numeric with values `1`, `2`, `3`, `4`, `6` or `12`

Premium payment frequency, specified as a `N`-by-`1` vector with values of `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of contract
`2` (actual/360) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the contract, specified as a positive integer using a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`BusDayConvention` — Business day conventions
`actual` (default) | character vector

Business day conventions, specified by a character vector. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char`

`PayAccruedPremium` — Flag for accrued premiums paid upon default
`true` (default) | integer with value `1` or `0`

Flag for accrued premiums paid upon default, specified as a `N`-by-`1` vector of Boolean flags that is `true` (default) if accrued premiums are paid upon default, `false` otherwise.

Data Types: `logical`

`Notional` — Contract notional values
`10MM` (default) | positive or negative integer

Contract notional values, specified as a `N`-by-`1`vector of integers. Use positive integer values for long positions and negative integer values for short positions.

Data Types: `double`

`TimeStep` — Number of days as time step for numerical integration
`10` (days) (default) | nonnegative integer

Number of days to take as time step for the numerical integration, specified as a nonnegative integer.

Data Types: `double`

`ZeroCompounding` — Compounding frequency of the zero curve
`2` (semiannual) (default) | integer with value of `1`,`2`,`3`,`4`,`6`,`12`, or `-1`

Compounding frequency of the zero curve, specified using values:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding
- `-1` — Continuous compounding

Data Types: `double`

`ZeroBasis` — Basis of the zero curve
`0` (actual/actual) (default) | integer with value of `0` to `13`

Basis of the zero curve, where the choices are identical to `Basis`.

Data Types: `double`

Output Arguments

`Price` — CDS clean prices
vector

CDS clean prices, returned as a `N`-by-`1` vector.

`AccPrem` — Acrued premiums
vector

Acrued premiums, returned as a `N`-by-`1` vector.

`PaymentDates` — Payment dates
matrix

Payment dates, returned as a `N`-by-`numCF` matrix.

`PaymentTimes` — Payment times
matrix

Payment times, returned as a `N`-by-`numCF` matrix of accrual fractions.

`PaymentCF` — Payments
matrix

Payments, returned as a `N`-by-`numCF` matrix.

Definitions

CDS Price

The price or mark-to-market (MtM) value of an existing CDS contract.

The CDS price is computed using the following formula:

```
CDS price = Notional * (Current Spread - Contract Spread) * RPV01
```

`Current Spread` is the current breakeven spread for a similar contract, according to current market conditions. `RPV01` is the 'risky present value of a basis point,' the present value of the premium payments, considering the default probability. This formula assumes a long position, and the right side is multiplied by -1 for short positions.

Algorithms

The premium leg is computed as the product of a spread S and the risky present value of a basis point (`RPV01`). The `RPV01` is given by:

$$RPV01 = \sum_{j=1}^{N} Z(t_j)\Delta(t_{j-1}, t_j, B)Q(t_j)$$

when no accrued premiums are paid upon default, and it can be approximated by

$$RPV01 \approx \frac{1}{2}\sum_{j=1}^{N} Z(t_j)\Delta(t_{j-1}, t_j, B)(Q(t_{j-1}) + Q(t_j))$$

when accrued premiums are paid upon default. Here, $t_0 = 0$ is the valuation date, and $t_1,...,t_n = T$ are the premium payment dates over the life of the contract, T is the maturity of the contract, $Z(t)$ is the discount factor for a payment received at time t, and $\Delta(t_{j-1}, t_j, B)$ is a day count between dates t_{j-1} and t_j corresponding to a basis B.

The protection leg of a CDS contract is given by the following formula:

$$ProtectionLeg = \int_0^T Z(\tau)(1-R)dPD(\tau)$$

$$\approx (1-R)\sum_{i=1}^{M} Z(\tau_i)(PD(\tau_i) - PD(\tau_{i-1}))$$

$$= (1-R)\sum_{i=1}^{M} Z(\tau_i)(Q(\tau_{i-1}) - Q(\tau_i))$$

where the integral is approximated with a finite sum over the discretization $\tau_0 = 0, \tau_1,...,\tau_M = T$.

If the spread of an existing CDS contract is S_C, and the current breakeven spread for a comparable contract is S_0, the current price, or mark-to-market value of the contract is given by:

$$\texttt{MtM} = \texttt{Notional}\ (S_0 - S_C)\texttt{RPV01}$$

This assumes a long position from the protection standpoint (protection was bought). For short positions, the sign is reversed.

References

[1] Beumee, J., D. Brigo, D. Schiemert, and G. Stoyle. *"Charting a Course Through the CDS Big Bang."* Fitch Solutions, Quantitative Research, Global Special Report. April 7, 2009.

[2] Hull, J., and A. White. "Valuing Credit Default Swaps I: No Counterparty Default Risk." *Journal of Derivatives.* Vol. 8, pp. 29–40.

[3] O'Kane, D. and S. Turnbull. *"Valuation of Credit Default Swaps."* Lehman Brothers, Fixed Income Quantitative Credit Research, April 2003.

See Also

`IRDataCurve` | `cdsbootstrap` | `cdsoptprice` | `cdsspread`

Topics

"Finding Breakeven Spread for New CDS Contract" on page 8-111
"Valuing an Existing CDS Contract" on page 8-114
"Converting from Running to Upfront" on page 8-117
"Credit Default Swap (CDS)" on page 8-107

External Websites

Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Introduced in R2010b

cdsspread

Determine spread of credit default swap

Syntax

```
[Spread,PaymentDates,PaymentTimes,] = cdsspread(ZeroData,ProbData,
Settle,Maturity,)
[Spread,PaymentDates,PaymentTimes,] = cdsspread(___,Name,Value)
```

Description

`[Spread,PaymentDates,PaymentTimes,] = cdsspread(ZeroData,ProbData, Settle,Maturity,)` computes the spread of the CDS.

`[Spread,PaymentDates,PaymentTimes,] = cdsspread(___,Name,Value)` adds optional name-value pair arguments.

Examples

Determine the Spread of a Credit Default Swap

This example shows how to use `cdsspread` to compute the spread (in basis points) for a CDS contract with the following data.

```
Settle = '17-Jul-2009'; % valuation date for the CDS
Zero_Time = [.5 1 2 3 4 5]';
Zero_Rate = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
Zero_Dates = daysadd(Settle,360*Zero_Time,1);
ZeroData = [Zero_Dates Zero_Rate];
ProbData = [daysadd(datenum(Settle),360,1), 0.0247];
Maturity = '20-Sep-2010';

Spread = cdsspread(ZeroData,ProbData,Settle,Maturity)

Spread = 148.2705
```

- "Finding Breakeven Spread for New CDS Contract" on page 8-111
- "Valuing an Existing CDS Contract" on page 8-114
- "Converting from Running to Upfront" on page 8-117
- "First-to-Default Swaps" (Financial Instruments Toolbox)
- "Pricing a CDS Index Option" (Financial Instruments Toolbox)

Input Arguments

`ZeroData` — Zero rate data

vector | `IRDataCurve` object

Zero rate data, specified as a `M`-by-`2` vector of dates and zero rates or an `IRDataCurve` object of zero rates.

When `ZeroData` is an `IRDataCurve` object, `ZeroCompounding` and `ZeroBasis` are implicit in `ZeroData` and are redundant inside this function. In this case, specify these optional parameters when constructing the `IRDataCurve` object before using the `cdsspread` function.

For more information on an `IRDataCurve` object, see "Creating an IRDataCurve Object" (Financial Instruments Toolbox).

Data Types: `double` | `struct`

`ProbData` — Default probability values

matrix

Default probability values, specified as a `P`-by-`2` matrix with dates and corresponding cumulative default probability values.

Data Types: `double` | `char`

`Settle` — Settlement date

serial date number | date character vector

Settlement date, specified as a `N`-by-`1` vector of serial date numbers or date character vectors. The `Settle` date must be earlier than or equal to the dates in `Maturity`.

Data Types: `double` | `char`

`Maturity` — Maturity date
serial date number | date character vector

Maturity date, specified as a `N`-by-`1` vector of serial date numbers or date character vectors.

Data Types: `double` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note Any optional input of size `N`-by-`1` is also acceptable as an array of size `1`-by-`N`, or as a single value applicable to all contracts. Single values are internally expanded to an array of size `N`-by-`1`.

Example: `Spread = cdsspread(ZeroData,ProbData,Settle,Maturity,'Basis',7,'BusDayConvention','previous')`

`RecoveryRate` — Recovery rate
`0.4` (default) | decimal

Recovery rate, specified as a `N`-by-`1` vector of recovery rates, specified as a decimal from `0` to `1`.

Data Types: `double`

`Period` — Premium payment frequency
`4` (default) | numeric with values `1`, `2`, `3`, `4`, `6` or `12`

Premium payment frequency, specified as a `N`-by-`1` vector with values of `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of contract

`2` (actual/360) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the contract, specified as a positive integer using a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`BusDayConvention` — Business day conventions

`actual` (default) | character vector

Business day conventions, specified by a character vector. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.

- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char`

`PayAccruedPremium` — Flag for accrued premiums paid upon default
`true` (default) | integer with value `1` or `0`

Flag for accrued premiums paid upon default, specified as a `N`-by-`1` vector of Boolean flags that is `true` (default) if accrued premiums are paid upon default, `false` otherwise.

Data Types: `logical`

`TimeStep` — Number of days as time step for numerical integration
`10` (days) (default) | nonnegative integer

Number of days to take as time step for the numerical integration, specified as a nonnegative integer.

Data Types: `double`

`ZeroCompounding` — Compounding frequency of the zero curve
`2` (semiannual) (default) | integer with value of `1`,`2`,`3`,`4`,`6`,`12`, or `-1`

Compounding frequency of the zero curve, specified using values:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

- `-1` — Continuous compounding

Data Types: `double`

`ZeroBasis` — Basis of the zero curve
`0` (actual/actual) (default) | integer with value of `0` to `13`

Basis of the zero curve, where the choices are identical to `Basis`.

Data Types: `double`

Output Arguments

`Spread` — Spreads (in basis points)
vector

Spreads (in basis points), returned as a `N`-by-`1` vector.

`PaymentDates` — Payment dates
matrix

Payment dates, returned as a `N`-by-`numCF` matrix.

`PaymentTimes` — Payment times
matrix

Payment times, returned as a `N`-by-`numCF` matrix of accrual fractions.

Definitions

CDS Spread

The market, or breakeven, spread value of a CDS.

The CDS spread can be computed by equating the value of the protection leg with the value of the premium leg:

```
Market Spread * RPV01 = Value of Protection Leg
```

The left side corresponds to the value of the premium leg, and this has been decomposed as the product of the market or breakeven spread times the `RPV01` or 'risky present value of a basis point' of the contract. The latter is the present value of the premium payments, considering the default probability. The `Market Spread` can be computed as the ratio of the value of the protection leg, to the `RPV01` of the contract. `cdsspread` returns the resulting spread in basis points.

Algorithms

The premium leg is computed as the product of a spread S and the risky present value of a basis point (`RPV01`). The `RPV01` is given by:

$$RPV01 = \sum_{j=1}^{N} Z(t_j)\Delta(t_{j-1}, t_j, B)Q(t_j)$$

when no accrued premiums are paid upon default, and it can be approximated by

$$RPV01 \approx \frac{1}{2}\sum_{j=1}^{N} Z(t_j)\Delta(t_{j-1}, t_j, B)(Q(t_{j-1}) + Q(t_j))$$

when accrued premiums are paid upon default. Here, $t_0 = 0$ is the valuation date, and $t_1,...,t_n = T$ are the premium payment dates over the life of the contract, T is the maturity of the contract, $Z(t)$ is the discount factor for a payment received at time t, and $\Delta(t_{j-1}, t_j, B)$ is a day count between dates t_{j-1} and t_j corresponding to a basis B.

The protection leg of a CDS contract is given by the following formula:

$$ProtectionLeg = \int_0^T Z(\tau)(1-R)dPD(\tau)$$

$$\approx (1-R)\sum_{i=1}^{M} Z(\tau_i)(PD(\tau_i) - PD(\tau_{i-1}))$$

$$= (1-R)\sum_{i=1}^{M} Z(\tau_i)(Q(\tau_{i-1}) - Q(\tau_i))$$

where the integral is approximated with a finite sum over the discretization $\tau_0 = 0, \tau_1,...,\tau_M = T$.

A breakeven spread S_0 makes the value of the premium and protection legs equal. It follows that:

$$S_0 = \frac{ProtectionLeg}{RPV01}$$

References

[1] Beumee, J., D. Brigo, D. Schiemert, and G. Stoyle. *"Charting a Course Through the CDS Big Bang."* Fitch Solutions, Quantitative Research, Global Special Report. April 7, 2009.

[2] Hull, J., and A. White. "Valuing Credit Default Swaps I: No Counterparty Default Risk." *Journal of Derivatives.* Vol. 8, pp. 29–40.

[3] O'Kane, D. and S. Turnbull. *"Valuation of Credit Default Swaps."* Lehman Brothers, Fixed Income Quantitative Credit Research, April 2003.

See Also

`IRDataCurve` | `cdsbootstrap` | `cdsprice`

Topics

"Finding Breakeven Spread for New CDS Contract" on page 8-111
"Valuing an Existing CDS Contract" on page 8-114
"Converting from Running to Upfront" on page 8-117
"First-to-Default Swaps" (Financial Instruments Toolbox)
"Pricing a CDS Index Option" (Financial Instruments Toolbox)
"Credit Default Swap (CDS)" on page 8-107

External Websites

Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Introduced in R2010b

cdsrpv01

Compute risky present value of a basis point for credit default swap

Syntax

```
RPV01 = cdsrpv01(ZeroData,ProbData,Settle,Maturity)
RPV01 = cdsrpv01(___,Name,Value)

[RPV01,PaymentDates,PaymentTimes] = cdsrpv01(ZeroData,ProbData,
Settle,Maturity)
[RPV01,PaymentDates,PaymentTimes] = cdsrpv01(___,Name,Value)
```

Description

`RPV01 = cdsrpv01(ZeroData,ProbData,Settle,Maturity)` computes the risky present value of a basis point (RPV01) for a credit default swap (CDS).

`RPV01 = cdsrpv01(___,Name,Value)` adds optional name-value arguments.

`[RPV01,PaymentDates,PaymentTimes] = cdsrpv01(ZeroData,ProbData, Settle,Maturity)` computes the risky present value of a basis point (RPV01), `PaymentDates`, and `PaymentTimes` for a credit default swap (CDS).

`[RPV01,PaymentDates,PaymentTimes] = cdsrpv01(___,Name,Value)` computes the risky present value of a basis point (RPV01), `PaymentDates`, and `PaymentTimes` for a credit default swap (CDS) using optional name-value pair arguments.

Examples

Calculate the `RPV01` Value for a CDS

Calculate the `RPV01` value, given the following specification for a CDS.

```
Settle = '17-Jul-2009'; % valuation date for the CDS
Zero_Time = [.5 1 2 3 4 5]';
Zero_Rate = [1.35 1.43 1.9 2.47 2.936 3.311]'/100;
Zero_Dates = daysadd(Settle,360*Zero_Time,1);
ZeroData = [Zero_Dates Zero_Rate];
ProbData = [daysadd(datenum(Settle),360,1), 0.0247];
Maturity = '20-Sep-2010';

RPV01 = cdsrpv01(ZeroData,ProbData,Settle,Maturity)

RPV01 = 1.1651
```

- "Pricing a CDS Index Option" (Financial Instruments Toolbox)

Input Arguments

`ZeroData` — Dates and zero rates

object from `IRDataCurve` or vector of dates and zero rates

Dates and zero rates, specified by an `M`-by-`2` vector of dates and zero rates or the object `IRDataCurve` for zero rates. For more information on an `IRDataCurve` object, see "Creating an IRDataCurve Object" (Financial Instruments Toolbox).

Data Types: `struct` | `double`

`ProbData` — Dates and default probabilities

vector of dates and default probabilities

Dates and default probabilities, specified by a `P`-by-`2` array.

Data Types: `double`

`Settle` — Settlement date

serial date number | character vector | cell array of character vectors

Settlement date, specified by a serial date number or date character vector. This must be earlier than or equal to the dates in `Maturity`.

Data Types: `char` | `cell` | `double`

`Maturity` — CDS maturity date

serial date number | character vector | cell array of character vectors

CDS maturity date, specified by an `N`-by-`1` vector of serial date numbers or date character vectors containing the maturity dates. The CDS premium payment dates occur at regular intervals, and the last payment occurs on these maturity dates.

Data Types: `char` | `cell` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `RPV01 = cdsrpv01(ZeroData,ProbData,Settle,Maturity,'Period',1,'StartDate','20-Sep-2010','Basis',1,'BusDayConvention',actual,'CleanRPV01',true,'PayAccruedPremium',true,'ZeroCompounding',1,'ZeroBasis',1)`

`Period` — Number of premium payments per year

`4` (default) | positive integer from the set `[1,2,3,4,6,12]` | vector of positive integers from the set `[1,2,3,4,6,12]`

Number of premium payments per year, specified by an `N`-by-`1` vector. Values are `1`, `2`, `3`, `4`, `6`, and `12`.

Data Types: `double`

`StartDate` — Dates the CDS premium leg starts

`Settle` date (default) | serial date number | character vector | cell array of character vectors

Dates when the CDS premium leg actually starts, specified by an `N`-by-`1` vector of serial date numbers or date character vectors. Must be on or between the `Settle` and `Maturity` dates. For a forward-starting CDS, specify this date as a future date after `Settle`.

Data Types: `double` | `char` | `cell`

`Basis` — Day-count basis of contract

`2` (actual/360) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis of the contract, specified as a positive integer using a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`BusDayConvention` — Business day conventions

`actual` (default) | character vector | cell array of character vectors

Business day conventions, specified by a character vector or `N`-by-`1` cell array of character vectors of business day conventions. The selection for business day convention determines how non-business days are treated. Non-business days are defined as weekends plus any other date that businesses are not open (e.g. statutory holidays). Values are:

- `actual` — Non-business days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `follow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day.
- `modifiedfollow` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.

- `previous` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day.
- `modifiedprevious` — Cash flows that fall on a non-business day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`CleanRPV01` — Flag for premium accrual
`true` (default) | boolean flag with value `true` or `false`

Flag for premium accrual, specified as an `N`-by-`1` vector of Boolean flags, which is `true` if the premium accrued at `StartDate` is excluded in the RPV01, and `false` otherwise.

Data Types: `logical`

`PayAccruedPremium` — Flag for accrued premium payment
`true` (default) | boolean flag with value `true` or `false`

Flag for accrued premium payment, specified as a `N`-by-`1` vector of Boolean flags, `true` if accrued premiums are paid upon default, `false` otherwise.

Data Types: `logical`

`ZeroCompounding` — Compounding frequency of zero curve
`2` semiannual compounding (default) | integer with acceptable value [`1,2,3,4,6,12, -1`]

Compounding frequency of the zero curve, specified with integer values:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding
- `-1` — Continuous compounding

Note When `ZeroData` is an `IRDataCurve` object, the arguments `ZeroCompounding` and `ZeroBasis` are implicit in `ZeroData` and are redundant inside this function. In that case, specify these optional arguments when constructing the `IRDataCurve` object before calling this function.

Data Types: `double`

`ZeroBasis` — Basis of zero curve

`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Basis of the zero curve, specified as a positive integer using a `NINST`-by-`1` vector.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`RPV01` — RPV01 value
scalar | vector

RPV01 value, returned as an `N`-by-`1` vector.

`PaymentDates` — Payment dates
scalar | vector

Payment dates, returned as an `N`-by-`numCF` matrix of dates.

`PaymentTimes` — Payment times
scalar | vector

Payment times, returned as an `N`-by-`numCF` matrix of accrual fractions.

Definitions

RPV01

RPV01, associated with a CDS, is the value of a stream of 1-basis-point premiums according to the payment structure of the CDS contract, and considering the default probability over time.

For more information, see [3] and [4] for details.

References

[1] [1] Beumee, J., D. Brigo, D. Schiemert, and G. Stoyle. “Charting a Course Through the CDS Big Bang.” *Fitch Solutions, Quantitative Research.* Global Special Report. April 7, 2009.

[2] [2] Hull, J., and A. White. “Valuing Credit Default Swaps I: No Counterparty Default Risk.” *Journal of Derivatives.* Vol. 8, pp. 29–40.

[3] [3] O'Kane, D. and S. Turnbull. “Valuation of Credit Default Swaps.” *Lehman Brothers, Fixed Income Quantitative Credit Research.* April, 2003.

[4] [4] O'Kane, D. *Modelling Single-name and Multi-name Credit Derivatives*. Wiley Finance, 2008.

See Also

`IRDataCurve` | `cdsbootstrap` | `cdsoptprice` | `cdsprice` | `cdsspread`

Topics

"Pricing a CDS Index Option" (Financial Instruments Toolbox)
"Credit Default Swap Option" (Financial Instruments Toolbox)

External Websites

www.cdsmodel.com/cdsmodel/
Pricing and Valuation of Credit Default Swaps (4 min 22 sec)

Introduced in R2013b

creditexposures

Compute credit exposures from contract values

Syntax

```
[exposures,exposurecpty] = creditexposures(values,counterparties)
[exposures,exposurecpty] = creditexposures(___,Name,Value)

[exposures,exposurecpty,collateral] = creditexposures(___,
Name,Value)
```

Description

`[exposures,exposurecpty] = creditexposures(values,counterparties)` computes the counterparty credit exposures from an array of mark-to-market OTC contract values. These exposures are used when calculating the CVA (credit value adjustment) for a portfolio.

`[exposures,exposurecpty] = creditexposures(___,Name,Value)` adds optional name-value arguments.

`[exposures,exposurecpty,collateral] = creditexposures(___,Name,Value)` computes the counterparty credit exposures from an array of mark-to-market OTC contract values using optional name-value pair arguments for `CollateralTable` and `Dates`, the `collateral` output is returned for the simulated collateral amounts available to counterparties at each simulation date and over each scenario.

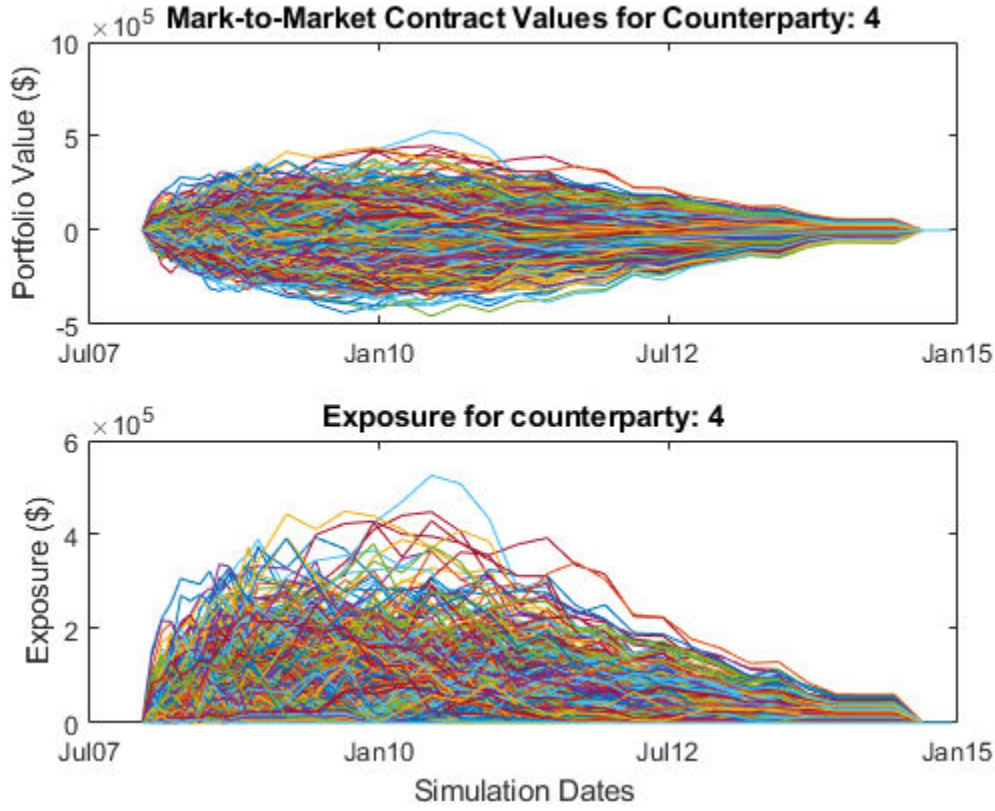
Examples

View Contract Values and Exposures Over Time for a Particular Counterparty

After computing the mark-to-market contract values for a portfolio of swaps over many scenarios, compute the credit exposure for a particular counterparty. View the contract

values and credit exposure over time. First, load data (`ccr.mat`) containing the mark-to-market contract values for a portfolio of swaps over many scenarios.

```
load ccr.mat
% Look at one counterparty.
cpID = 4;
cpValues = squeeze(sum(values(:,swaps.Counterparty == cpID,:),2));
subplot(2,1,1)
plot(simulationDates,cpValues);
title(sprintf('Mark-to-Market Contract Values for Counterparty: %d',cpID));
datetick('x','mmmyy')
ylabel('Portfolio Value ($)')
% Compute the exposure by counterparty.
[exposures, expcpty] = creditexposures(values,swaps.Counterparty,...
'NettingID',swaps.NettingID);
% View the credit exposure over time for the counterparty.
subplot(2,1,2)
cpIdx = find(expcpty == cpID);
plot(simulationDates,squeeze(exposures(:,cpIdx,:)));
title(sprintf('Exposure for counterparty: %d',cpIdx));
datetick('x','mmmyy')
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Compute the Credit Exposure and Determine the Incremental Exposure for a New Trade

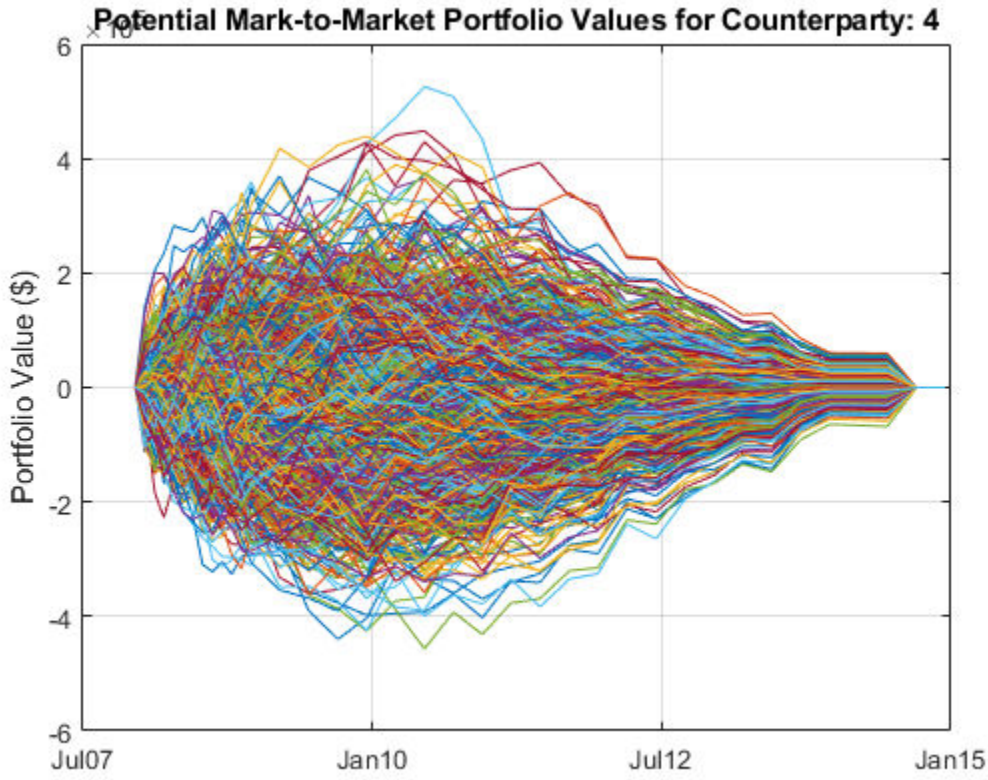
Load the data (`ccr.mat`) containing the mark-to-market contract values for a portfolio of swaps over many scenarios.

```
load ccr.mat
```

Look at one counterparty.

```
cpID = 4;
cpIdx = swaps.Counterparty == cpID;
```

```
cpValues = values(:,cpIdx,:);
plot(simulationDates,squeeze(sum(cpValues,2)));
grid on;
title(sprintf('Potential Mark-to-Market Portfolio Values for Counterparty: %d',cpID));
datetick('x','mmmyy')
ylabel('Portfolio Value ($)')
```

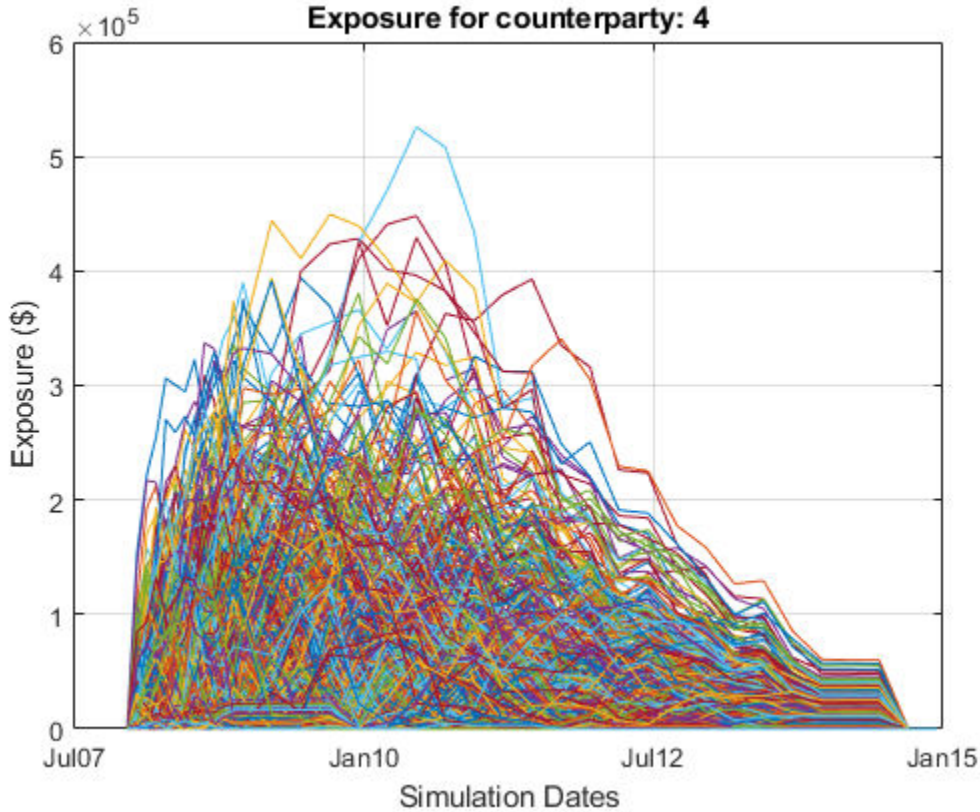


Compute the exposures.

```
netting = swaps.NettingID(cpIdx);
exposures = creditexposures(cpValues,cpID,'NettingID',netting);
```

View the credit exposure over time for the counterparty.

```
figure;
plot(simulationDates,squeeze(exposures));
grid on
title(sprintf('Exposure for counterparty: %d',cpID));
datetick('x','mmmyy')
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Compute the credit exposure profiles.

```
profilesBefore = exposureprofiles(simulationDates,exposures)
```

```
profilesBefore = struct with fields:
     Dates: [37x1 double]
        EE: [37x1 double]
```

```
       PFE: [37x1 double]
      MPFE: 2.1580e+05
     EffEE: [37x1 double]
       EPE: 2.8602e+04
    EffEPE: 4.9579e+04
```

Consider a new trade with a counterparty. For this example, take another trade from the original swap portfolio and "copy" it for a new counterparty. This example is only for illustrative purposes.

```
newTradeIdx = 3;
newTradeValues = values(:,newTradeIdx,:);

% Append a new trade to your existing portfolio.
cpValues = [cpValues newTradeValues];
netting = [netting; cpID];
exposures = creditexposures(cpValues,cpID,'NettingID',netting);
```

Compute the new credit exposure profiles.

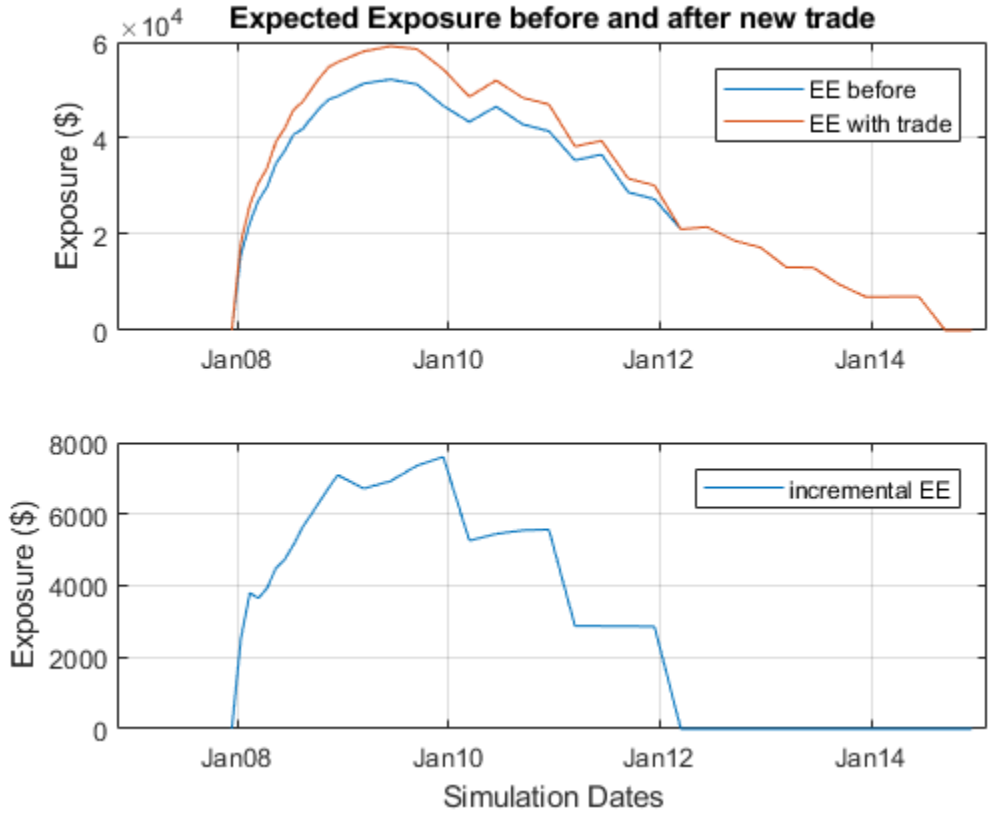
```
profilesAfter = exposureprofiles(simulationDates,exposures)
```

```
profilesAfter = struct with fields:
     Dates: [37x1 double]
        EE: [37x1 double]
       PFE: [37x1 double]
      MPFE: 2.4689e+05
     EffEE: [37x1 double]
       EPE: 3.1609e+04
    EffEPE: 5.6178e+04
```

Visualize the expected exposures and the new trade's incremental exposure. The incremental exposure is used to compute the incremental credit value adjustment (CVA) charge.

```
figure;
subplot(2,1,1)
plot(simulationDates,profilesBefore.EE,...
    simulationDates,profilesAfter.EE);
grid on;
legend({'EE before','EE with trade'})
datetick('x','mmmyy','keeplimits')
title('Expected Exposure before and after new trade');
```

```
ylabel('Exposure ($)')

subplot(2,1,2)
incrementalEE = profilesAfter.EE - profilesBefore.EE;
plot(simulationDates,incrementalEE);
grid on;
legend('incremental EE')
datetick('x','mmmyy','keeplimits')
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Compute Exposures for Counterparties Under Collateral Agreement

Load the data (`ccr.mat`) containing the mark-to-market contract values for a portfolio of swaps over many scenarios.

```
load ccr.mat
```

Only look at a single counterparty for this example.

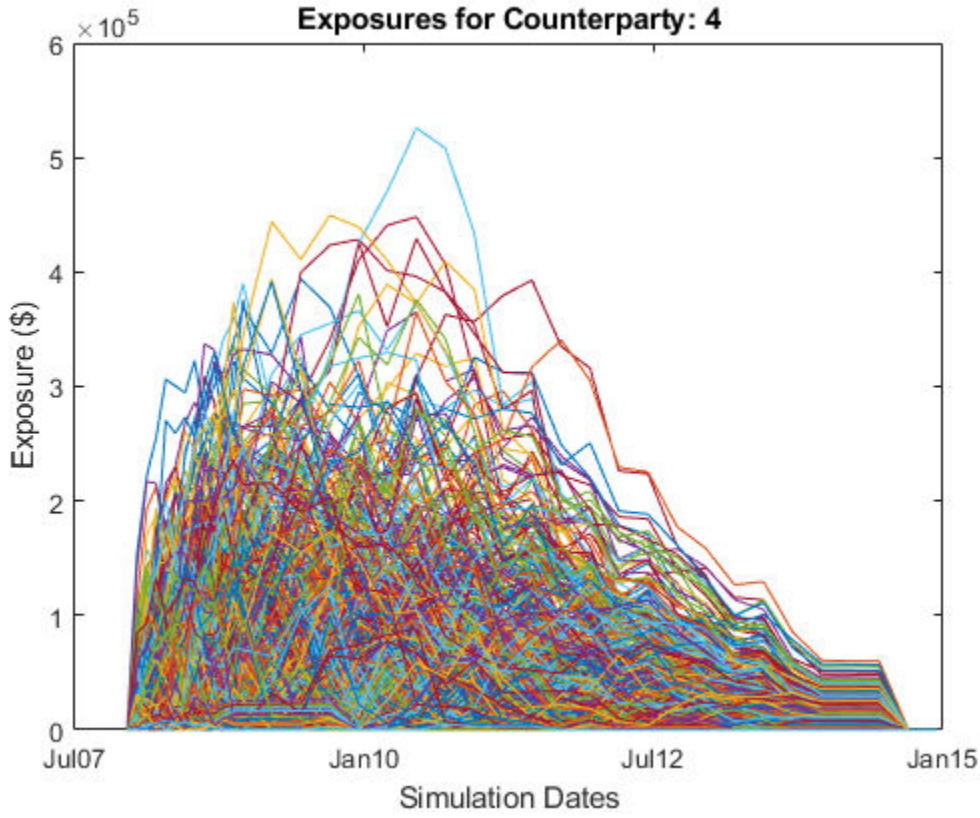
```
cpID = 4;
cpIdx = swaps.Counterparty == cpID;
cpValues = values(:,cpIdx,:);
```

Compute uncollateralized exposures.

```
exposures = creditexposures(cpValues,swaps.Counterparty(cpIdx),...
'NettingID',swaps.NettingID(cpIdx));
```

View credit exposure over time for the counterparty.

```
plot(simulationDates,squeeze(exposures));
expYLim = get(gca,'YLim');
title(sprintf('Exposures for Counterparty: %d',cpID));
datetick('x','mmmyy')
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



Now add a collateral agreement for the counterparty. The `'CollateralTable'` parameter is a MATLAB® table. You can create tables from spreadsheets or other data sources, in addition to building them inline as seen here. For more information, see `table`.

```
collateralVariables = {'Counterparty';'PeriodOfRisk';'Threshold';'MinimumTransfer'};
periodOfRisk = 14;
threshold = 100000;
minTransfer = 10000;
collateralTable = table(cpID,periodOfRisk,threshold,minTransfer,...
'VariableNames',collateralVariables)
```

```
collateralTable=1x4 table
    Counterparty    PeriodOfRisk    Threshold    MinimumTransfer
```

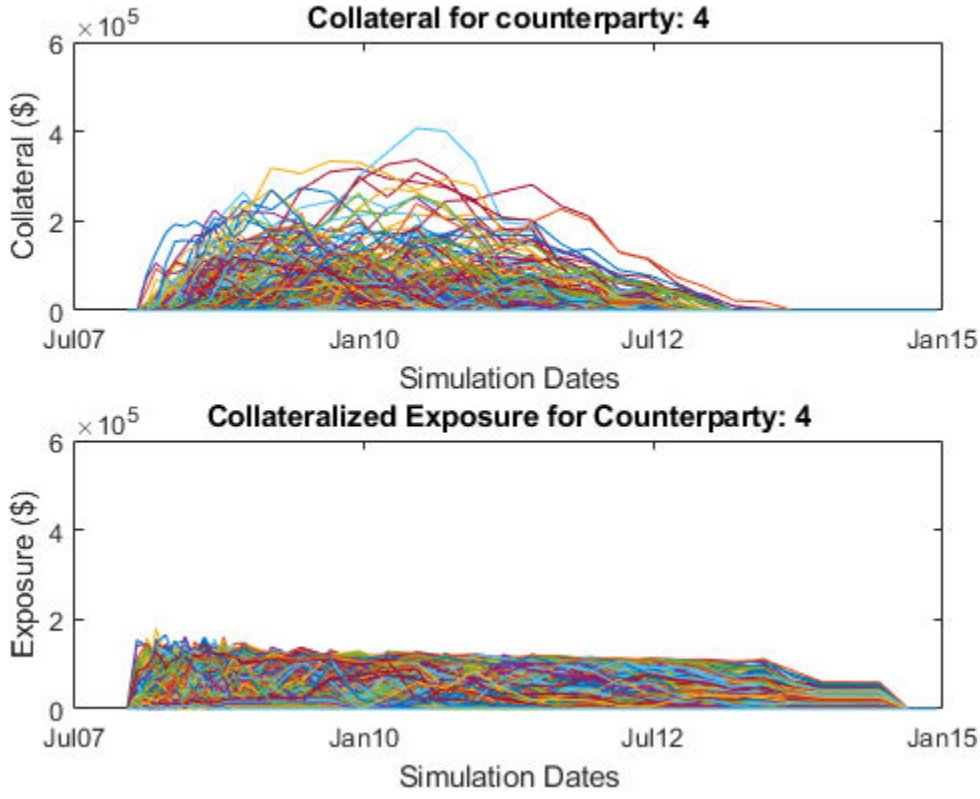
```
    ____________    _____________    _________    _________________

    4               14               1e+05        10000
```

Compute collateralized exposures.

```
[collatExp, collatcpty, collateral] = creditexposures(cpValues,...
    swaps.Counterparty(cpIdx),'NettingID',swaps.NettingID(cpIdx),...
    'CollateralTable',collateralTable,'Dates',simulationDates);
```

Plot collateral levels and collateralized exposures.

```
figure;
subplot(2,1,1)
plot(simulationDates,squeeze(collateral));
set(gca,'YLim',expYLim);
title(sprintf('Collateral for counterparty: %d',cpID));
datetick('x','mmmyy')
ylabel('Collateral ($)')
xlabel('Simulation Dates')

subplot(2,1,2)
plot(simulationDates,squeeze(collatExp));
set(gca,'YLim',expYLim);
title(sprintf('Collateralized Exposure for Counterparty: %d',cpID));
datetick('x','mmmyy')
ylabel('Exposure ($)')
xlabel('Simulation Dates');
```



- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)
- “Wrong Way Risk with Copulas” (Financial Instruments Toolbox)

Input Arguments

`values` — 3-D array of simulated mark-to-market values of portfolio of contracts

array

3-D array of simulated mark-to-market values of a portfolio of contracts simulated over a series of simulation dates and across many scenarios, specified as a `NumDates`-by-`NumContracts`-by-`NumScenarios` “cube” of contract values. Each row represents a

different simulation date, each column a different contract, and each “page” is a different scenario from a Monte-Carlo simulation.

Data Types: `double`

`counterparties` — Counterparties corresponding to each contract
vector | cell array

Counterparties corresponding to each contract in `values`, specified as a `NumContracts`-element vector of counterparties. Counterparties can be a vector of numeric IDs or a cell array of counterparty names. By default, each counterparty is assumed to have one netting set that covers all of its contracts. If counterparties are covered by multiple netting sets, then use the `NettingID` parameter. A value of `NaN` (or `''` in a cell array) indicates that a contract is not included in any netting set unless otherwise specified by `NettingID`. `counterparties` is case insensitive and leading or trailing white spaces are removed.

Data Types: `double` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[exposures,exposurecpty] = creditexposures(values,counterparties,'NettingID','10','ExposureType','Additive')`

`NettingID` — Netting set IDs indicate which netting set each contract belongs
vector | cell array

Netting set IDs to indicate to which netting set each contract in `values` belongs, specified by a `NumContracts`-element vector of netting set IDs. `NettingID` can be a vector of numeric IDs or else a cell array of character vector identifiers. The `creditexposures` function uses `counterparties` and `NettingID` to define each unique netting set (all contracts in a netting set must be with the same counterparty). By default, each counterparty has a single netting set which covers all of their contracts. A value of `NaN` (or `''` in a cell array) indicates that a contract is not included in any netting set. `NettingID` is case insensitive and leading or trailing white spaces are removed.

Data Types: `double` | `cell`

`ExposureType` — Calculation method for exposures
`'Counterparty'` (default) | character vector with value of `'Counterparty'` or `'Additive'`

Calculation method for exposures, specified with values:

- `'Counterparty'` — Compute exposures per counterparty.
- `'Additive'` — Compute additive exposures at the contract level. Exposures are computed per contract and sum to the total counterparty exposure.

Data Types: `char`

`CollateralTable` — Table containing information on collateral agreements of counterparties
MATLAB table

Table containing information on collateral agreements of counterparties, specified as a MATLAB table. The table consists of one entry (row) per collateralized counterparty and must have the following variables (columns):

- `'Counterparty'` — Counterparty name or ID. The Counterparty name or ID should match the parameter `'Counterparty'` for the `ExposureType` argument.
- `'PeriodOfRisk'` — Margin period of risk in days. The number of days from a margin call until the posted collateral is available from the counterparty.
- `'Threshold'` — Collateral threshold. When counterparty exposures exceed this amount, the counterparty must post collateral.
- `'MinimumTransfer'` — Minimum transfer amount. The minimum amount over/under the threshold required to trigger transfer of collateral.

Note When computing collateralized exposures, both the `CollateralTable` parameter and the `Dates` parameter must be specified.

Data Types: `table`

`Dates` — Simulation dates corresponding to each row of the values array
vector of date numbers | cell array of character vectors

Simulation dates corresponding to each row of the `values` array, specified as a `NUMDATES`-by-`1` vector of simulation dates. `Dates` is either a vector of MATLAB date numbers or else a cell array of character vectors in a known date format. See `datenum` for known date formats.

Note When computing collateralized exposures, both the `CollateralTable` parameter and the `Dates` parameter must be specified.

Data Types: `double` | `cell`

Output Arguments

`exposures` — 3-D array of credit exposures

array

3-D array of credit exposures representing the potential losses from each counterparty or contract at each date and over all scenarios. The size of `exposures` depends on the `ExposureType` input argument:

- When `ExposureType` is `'Counterparty'`, `exposures` returns a `NumDates`-by-`NumCounterparties`-by-`NumScenarios` "cube" of credit exposures representing potential losses that could be incurred over all dates, counterparties, and scenarios, if a counterparty defaulted (ignoring any post-default recovery).
- When `ExposureType` is `'Additive'`, `exposures` returns a `NumDates`-by-`NumContracts`-by-`NumScenarios` "cube," where each element is the additive exposure of each contract (over all dates and scenarios). Additive exposures sum to the counterparty-level exposure.

`exposurecpty` — Counterparties that correspond to columns of `exposures` array

vector

Counterparties that correspond to columns of the `exposures` array, returned as `NumCounterparties` or `NumContracts` elements depending on the `ExposureType`.

`collateral` — Simulated collateral amounts available to counterparties at each simulation date and over each scenario

3D array

Simulated collateral amounts available to counterparties at each simulation date and over each scenario, returned as a `NumDates`-by-`NumCounterparties`-by-`NumScenarios` 3D array. Collateral amounts are calculated using a Brownian bridge to estimate contract values between simulation dates. For more information, see "Brownian Bridge" on page 18-332. If the `CollateralTable` was not specified, this output is empty.

Definitions

Brownian Bridge

A Brownian bridge is used to simulate portfolio values at intermediate dates to compute collateral available at the subsequent simulation dates.

For example, to estimate collateral available at a particular simulation date, t_i, you need to know the state of the portfolio at time $t_i - dt$, where dt is the margin period of risk. Portfolio values are simulated at these intermediate dates by drawing from a distribution defined by the Brownian bridge between t_i and the previous simulation date, t_{i-1}.

If the contract values at time t_{i-1} and t_i are known and you want to estimate the contract value at time t_c (where t_c is $t_i - dt$), then a sample from a normal distribution is used with variance:

$$\frac{(t_i - t_c)(t_c - t_{i-1})}{(t_i - t_{i-1})}$$

and with mean that is simply the linear interpolation of the contract values between the two simulation dates at time t_c. For more details, see References.

References

[1] Lomibao, D., and S. Zhu. *"A Conditional Valuation Approach for Path-Dependent Instruments."* August 2005.

[2] Pykhtin M. *"Modeling credit exposure for collateralized counterparties."* December 2009.

[3] Pykhtin M., and S. Zhu. *"A Guide to Modeling Counterparty Credit Risk."* GARP, July/August 2007, issue 37.

[4] Pykhtin, Michael., and Dan Rosen. *"Pricing Counterparty Risk at the Trade Level and CVA Allocations."* FEDS Working Paper No. 10., February 1, 2010.

See Also

`datenum` | `exposureprofiles` | `table`

Topics

"Counterparty Credit Risk and CVA" (Financial Instruments Toolbox)
"Wrong Way Risk with Copulas" (Financial Instruments Toolbox)

Introduced in R2014a

exposureprofiles

Compute exposure profiles from credit exposures

Syntax

```
profilestructs = exposureprofiles(dates,exposures)
profilestructs = exposureprofiles(___,Name,Value)
```

Description

`profilestructs = exposureprofiles(dates,exposures)` computes common counterparty credit exposures profiles from an array of exposures.

`profilestructs = exposureprofiles(___,Name,Value)` adds optional name-value arguments.

Examples

View Exposure Profiles of a Particular Counterparty

After computing the mark-to-market contract values for a portfolio of swaps over many scenarios, view the exposure profiles of a particular counterparty.

First, load data (`ccr.mat`) containing the mark-to-market contract values for a portfolio of swaps over many scenarios.

```
load ccr.mat
```

Compute the exposure by counterparty.

```
[exposures, expcpty] = creditexposures(values,swaps.Counterparty,...
'NettingID',swaps.NettingID);
```

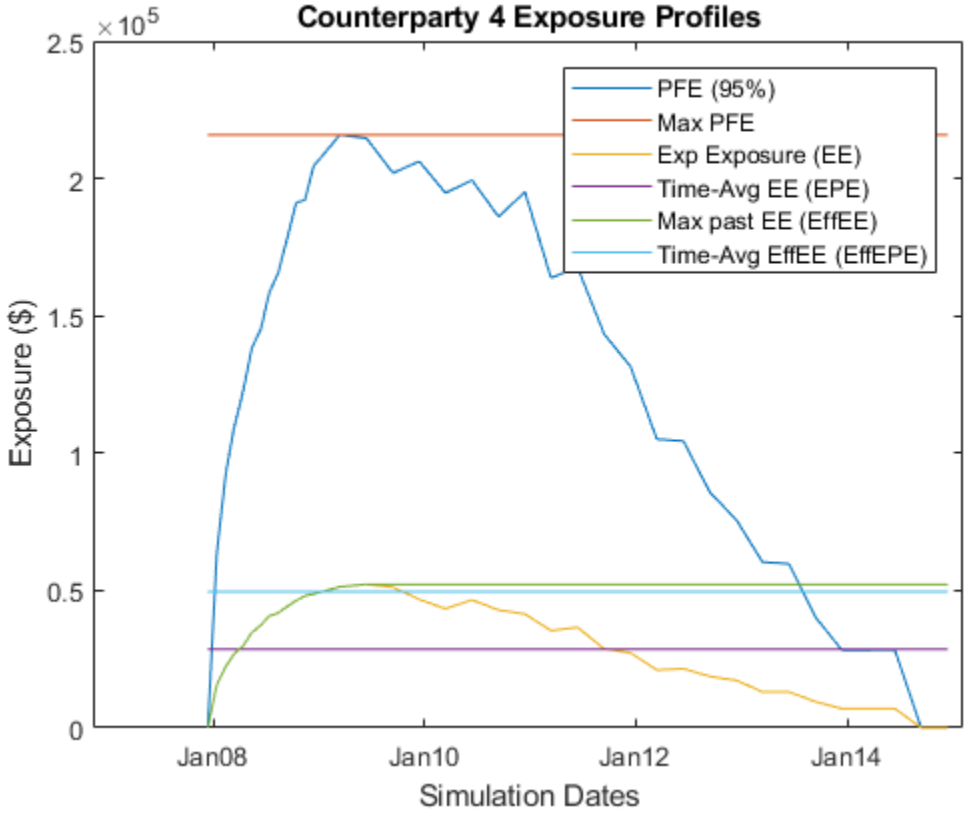
Compute the credit exposure profiles for all counterparties.

```
 cpProfiles = exposureprofiles(simulationDates,exposures)
```

```
cpProfiles = 5x1 struct array with fields:
    Dates
    EE
    PFE
    MPFE
    EffEE
    EPE
    EffEPE
```

Visualize the exposure profiles for a particular counterparty.

```
cpIdx = find(expcpty == 4);
numDates = numel(simulationDates);
plot(simulationDates,cpProfiles(cpIdx).PFE,...
        simulationDates,cpProfiles(cpIdx).MPFE * ones(numDates,1),...
        simulationDates,cpProfiles(cpIdx).EE,...
        simulationDates,cpProfiles(cpIdx).EPE * ones(numDates,1),...
        simulationDates,cpProfiles(cpIdx).EffEE,...
        simulationDates,cpProfiles(cpIdx).EffEPE * ones(numDates,1));
legend({'PFE (95%)','Max PFE','Exp Exposure (EE)',...
        'Time-Avg EE (EPE)','Max past EE (EffEE)',...
        'Time-Avg EffEE (EffEPE)'})
datetick('x','mmmyy','keeplimits')
title(sprintf('Counterparty %d Exposure Profiles',cpIdx));
ylabel('Exposure ($)')
xlabel('Simulation Dates')
```



- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)
- “Wrong Way Risk with Copulas” (Financial Instruments Toolbox)

Input Arguments

`dates` — Simulation dates

vector of date numbers | cell array of character vectors

Simulation dates, specified as vector of date numbers or a cell array of character vectors in a known date format. For more information for known date formats, see the function `datenum`.

Data Types: `double` | `char` | `cell`

`exposures` — 3-D array of potential losses due to counterparty default
array

3-D array of potential losses due to counterparty default on a set of instruments simulated over a series of simulation dates and across many scenarios, specified as a `NumDates`-by-`NumCounterParties`-by-`NumScenarios` "cube" of credit exposures. Each row represents a different simulation date, each column a different counterparty, and each "page" is a different scenario from a Monte-Carlo simulation.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `profilestructs = exposureprofiles(dates,exposures,'ProfileSpec','PFE','PFEProbabilityLevel',.9)`

`ProfileSpec` — Exposure profiles
`All` (generate all profiles) (default) | character vector with possible values `EE`, `PFE`, `MPE`, `EffEE`, `EPE`, `EffEPE`, `All` | cell array of character vectors with possible values `EE`, `PFE`, `MPE`, `EffEE`, `EPE`, `EffEPE`

Exposure profiles, specified as a character vector or cell array of character vectors with the following possible values:

- `EE` — Expected Exposure. The mean of the distribution of exposures at each date. A `[NumDates-by-1]` vector.
- `PFE` — Potential Future Exposure. A high percentile (default 95%) of the distribution of possible exposures at each date. This is sometimes referred to as "Peak Exposure." A `[NumDates-by-1]` vector.
- `MPFE` — Maximum Potential Future Exposure. The maximum potential future exposure (PFE) over all dates

- `EffEE` — Effective Expected Exposure. The maximum expected exposure (at a specific date) that occurs at that date or any prior date. This is the expected exposure, but constrained to be nondecreasing over time. A `[NumDates-by-1]` vector.
- `EPE` — Expected Positive Exposure. The weighted average over time of expected exposures. A scalar.
- `EffEPE` — Effective Expected Positive Exposure. The weighted average over time of the effective expected exposure (EffEE). A scalar.
- `All` — Generate all the previous profiles.

Note Exposure profiles are computed on a per-counterparty basis.

Data Types: `char` | `cell`

`PFEProbabilityLevel` — Level for potential future exposure (PFE) and maximum potential future exposure (MPFE)
`.95` (the 95th percentile) (default) | scalar with value `[0..1]`

Level for potential future exposure (PFE) and maximum potential future exposure (MPFE), specified as a scalar with value `[0..1]`.

Data Types: `double`

Output Arguments

`profilestructs` — Structure of credit exposure profiles
array of structs holding credit exposure profiles for each counterparty

Structure of credit exposure profiles, returned as an array of structs holding credit exposure profiles for each counterparty, returned as a struct, with the fields of the struct as the (abbreviated) names of every exposure profile. Profiles listed in the `ProfileSpec` (and their related profiles) are populated, while those not requested contain empty (`[]`). `profilestructs` contains the `dates` information as a vector of MATLAB date numbers requested in the `ProfileSpec` argument.

References

[1] *Basel II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework - Comprehensive Version.* at `http://www.bis.org/publ/bcbs128.htm`, 2006.

See Also

`creditexposures` | `datenum`

Topics

"Counterparty Credit Risk and CVA" (Financial Instruments Toolbox)
"Wrong Way Risk with Copulas" (Financial Instruments Toolbox)

Introduced in R2014a

cdyield

Yield on certificate of deposit (CD)

Syntax

```
Yield = cdyield(Price,CouponRate,Settle,Maturity,IssueDate)
Yield = cdyield(___,Basis)
```

Description

`Yield = cdyield(Price,CouponRate,Settle,Maturity,IssueDate)` computes the yield to maturity of a certificate of deposit given its clean price.

`cdyield` assumes that the certificates of deposit pay interest at maturity. Because of the simple interest treatment of these securities, this function is best used for short-term maturities (less than 1 year). The default simple interest calculation uses the `Basis` for the actual/360 convention (`2`).

`Yield = cdyield(___,Basis)` adds an optional argument for `Basis`.

Examples

Compute the Yield to Maturity of a Certificate of Deposit

This example shows how to compute the yield on the certificate of deposit (CD), given a CD with the following characteristics.

```
Price      = 101.125;
CouponRate = 0.05;
Settle     = '02-Jan-02';
Maturity   = '31-Mar-02';
IssueDate = '1-Oct-01';

Yield = cdyield(Price, CouponRate, Settle, Maturity, IssueDate)
```

```
Yield = 0.0039
```

Compute the Yield to Maturity of a Certificate of Deposit Using datetime Inputs

This example shows how to use `datetime` inputs to compute the yield on the certificate of deposit (CD), given a CD with the following characteristics.

```
Price      = 101.125;
CouponRate = 0.05;
Settle     = datetime('02-Jan-02','Locale','en_US');
Maturity   = datetime('31-Mar-02','Locale','en_US');
IssueDate = datetime('1-Oct-01','Locale','en_US');

Yield = cdyield(Price, CouponRate, Settle, Maturity, IssueDate)

Yield = 0.0039
```

- “Coupon Date Calculations” on page 2-34

Input Arguments

`Price` — Clean price of certificate of deposit per $100 face

numeric

Clean price of the certificate of deposit per $100 face, specified as a numeric value using a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

Data Types: `double`

`CouponRate` — Coupon annual interest rate

decimal

Coupon annual interest rate, specified as decimal using a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

Data Types: `double`

`Settle` — Settlement date of certificate of deposit

serial date number | date character vector | datetime

Settlement date of the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of certificate of deposit
serial date number | date character vector | datetime

Maturity date of the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`IssueDate` — Issue date for certificate of deposit
serial date number | date character vector | datetime

Issue date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Basis` — Day-count basis for certificate of deposit
`2` (actual/360) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis for the certificate of deposit, specified as a scalar or a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`Yield` — Simple yield to maturity of certificate of deposit

numeric

Simple yield to maturity of the certificate of deposit, returned as a `NCDS`-by-`1` or `1`-by-`NCDS` vector.

See Also

`bndprice` | `cdai` | `cdprice` | `datetime` | `stepcpnprice` | `tbillprice`

Topics

"Coupon Date Calculations" on page 2-34
"Yield Conventions" on page 2-34

Introduced before R2006a

cev class

Constant Elasticity of Variance (CEV) models

Description

The `cev` constructor creates and displays `cev` objects, which derive from the `sdeld` (SDE with drift rate expressed in linear form) class. Use `cev` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

This model allows you to simulate any vector-valued SDE of the form:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`NVARS` (generalized) expected instantaneous rate of return matrix.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Construction

`CEV = cev(Return,Alpha,Sigma)` constructs a default `cev` object.

`CEV = cev(Return,Alpha,Sigma,Name,Value)` constructs a `cev` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `cev` object, see `cev`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Return` — `Return` represents the parameter μ

array or deterministic function of time or deterministic function of time and state

`Return` represents the parameter μ, specified as an array or deterministic function of time.

If you specify `Return` as an array, it must be an `NVARS`-by-`NVARS` matrix representing the expected (mean) instantaneous rate of return.

As a deterministic function of time, when `Return` is called with a real-valued scalar time `t` as its only input, `Return` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Return` as a function of time and state, it must return an `NVARS`-by-`NVARS` matrix when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Alpha` — `Return` represents the parameter D
array or deterministic function of time or deterministic function of time and state

`Alpha` represents the parameter D, specified as an array or deterministic function of time.

If you specify `Alpha` as an array, it represents an `NVARS`-by-`1` column vector of exponents.

As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` matrix.

If you specify it as a function of time and state, `Alpha` must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter V
array or deterministic function of time or deterministic function of time and state

`Sigma` represents the parameter V, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `cev`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$

- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as an object or function accessible by $(t, X_t$.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t,X_t) = D(t,X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

StartState — Initial values of state variables

1 (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If StartState is a scalar, the gbm constructor applies the same initial value to all state variables on all trials.

If StartState is a column vector, the gbm constructor applies a unique initial value to each state variable on all trials.

If StartState is a matrix, the gbm constructor applies a unique initial value to each state variable on each trial.

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

Simulation — User-defined simulation function or SDE simulation method

if you do not specify a value for Simulation, the default method is simulation by Euler approximation (simByEuler) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

SetAccess	public
GetAccess	public

Data Types: function_handle

Methods

Inherited Methods

The following methods are inherited from the `sde` class.

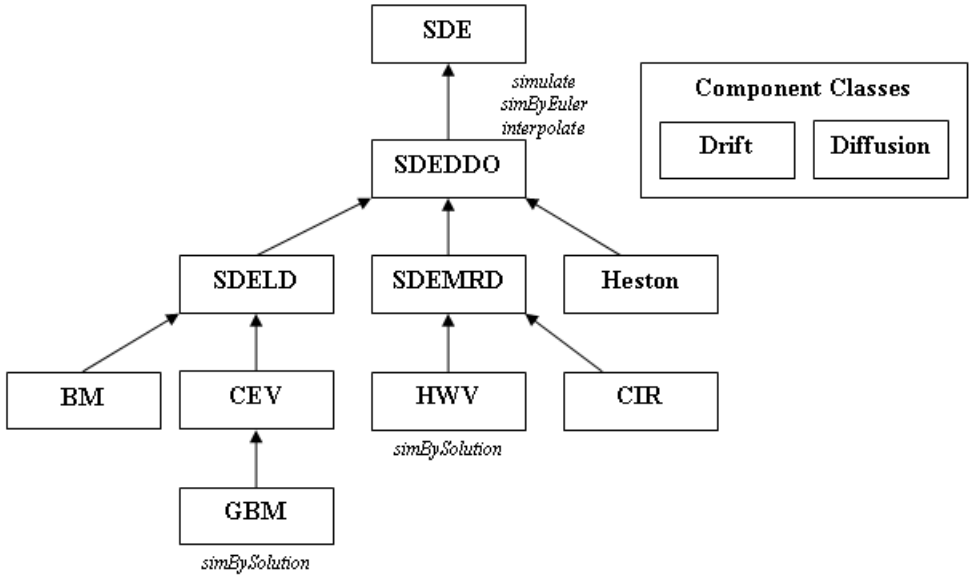
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a cev Object

Create a univariate `cev` object to represent the model: $dX_t = 0.25X_t + 0.3X_t^{\frac{1}{2}}dW_t$.

```
obj = cev(0.25, 0.5, 0.3) % (B = Return, Alpha, Sigma)
```

```
obj =
   Class CEV: Constant Elasticity of Variance
   ------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.25
          Alpha: 0.5
          Sigma: 0.3
```

`cev` objects display the parameter `B` as the more familiar `Return`

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19

- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `cev` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeld` | `simByEuler` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

cev

Construct Constant Elasticity of Variance (CEV) models

Syntax

```
CEV = cev(Return, Alpha, Sigma)

CEV = cev(Return, Alpha, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`cev`

Description

This constructor creates and displays `cev` objects, which derive from the`sdeld` (SDE with drift rate expressed in linear form) class. Use `cev` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

This constructor allows you to simulate any vector-valued SDE of the form:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`NVARS` (generalized) expected instantaneous rate of return matrix.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.

- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Return`	`Return` represents the parameter μ. If you specify `Return` as an array, it is a `NVARS`-by-`NVARS` 2-dimensional matrix that represents the expected (mean) instantaneous rate of return. As a deterministic function of time, when `Return` is called with a real-valued scalar time `t` as its only input, `Return` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Return` as a function of time and state, it calculates the expected instantaneous rate of return. This function must generate an `NVARS`-by-`NVARS` matrix when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.

`Alpha`	`Alpha` determines the format of the parameter *D*. If you specify `Alpha` as an array, it represents an `NVARS`-by-`1` column vector of exponents. As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` matrix. If you specify it as a function of time and state, `Alpha` must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.
`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it represents an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify it as a function of time and state, `Sigma` must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of these input arguments, each argument is specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `cev` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `cev` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `cev` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.

`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`CEV`	Object of class `cev` with the following displayed parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method • `Return`: Access function for the input argument `Return`, callable as a function of time and state • `Alpha`: Access function for the input argument `Alpha`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Examples

- “Creating Constant Elasticity of Variance (CEV) Models” on page 17-26
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects on page 17-37

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or

equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `cev` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeld`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16

"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

cfamounts

Cash flow and time mapping for bond portfolio

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Period`, `Basis`, `EndMonthRule`, `IssueDate`,`FirstCouponDate`, `LastCouponDate`, `StartDate`,`Face`, `AdjustCashFlowsBasis`, `BusinessDayConvention`, `CompoundingFrequency`, `DiscountBasis`, `Holidays`, and `PrincipalType`.

Syntax

```
[CFlowAmounts,CFlowDates,TFactors,CFlowFlags,CFPrincipal] =
cfamounts(CouponRate,Settle,Maturity)
[CFlowAmounts,CFlowDates,TFactors,CFlowFlags,CFPrincipal] =
cfamounts(___,Name,Value)
```

Description

`[CFlowAmounts,CFlowDates,TFactors,CFlowFlags,CFPrincipal] = cfamounts(CouponRate,Settle,Maturity)` returns matrices of cash flow amounts, cash flow dates, time factors, and cash flow flags for a portfolio of `NUMBONDS` fixed-income securities.

The elements contained in the `cfamounts` outputs for 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 accrued interest is zero in the case of all zero coupon bonds. `cfamounts` determines all cash flows and time mappings for a bond whether or not the coupon structure contains odd first or last periods. All output matrices are padded with `NaN`s as necessary to ensure that all rows have the same number of elements.

`[CFlowAmounts,CFlowDates,TFactors,CFlowFlags,CFPrincipal] = cfamounts(___,Name,Value)` adds optional name-value arguments.

Examples

Compute the Cash Flow Structure and Time Factors for a Bond Portfolio

This example shows how to compute the cash flow structure and time factors for a bond portfolio that contains a corporate bond paying interest quarterly and a Treasury bond paying interest semiannually.

```
Settle = '01-Nov-1993';
Maturity = ['15-Dec-1994';'15-Jun-1995'];
CouponRate= [0.06; 0.05];
Period = [4; 2];
Basis = [1; 0];
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ...
cfamounts(CouponRate,Settle, Maturity, Period, Basis)

CFlowAmounts =

   -0.7667    1.5000    1.5000    1.5000    1.5000  101.5000
   -1.8989    2.5000    2.5000    2.5000  102.5000       NaN


CFlowDates =

      728234      728278      728368      728460      728552      728643
      728234      728278      728460      728643      728825         NaN


TFactors =

         0    0.2404    0.7403    1.2404    1.7403    2.2404
         0    0.2404    1.2404    2.2404    3.2404       NaN


CFlowFlags =

     0     3     3     3     3     4
     0     3     3     3     4   NaN
```

Compute the Cash Flow Structure and Time Factors for a Bond Portfolio and Return a datatime array for `CFlowDates`

This example shows how to compute the cash flow structure and time factors for a bond portfolio that contains a corporate bond paying interest quarterly and a Treasury bond paying interest semiannually and `CFlowDates` is returned as a datatime array.

```
Settle = datetime('01-Nov-1993','Locale','en_US');
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 = 2x6 datetime array
Columns 1 through 5

   01-Nov-1993   15-Dec-1993   15-Mar-1994   15-Jun-1994   15-Sep-1994
   01-Nov-1993   15-Dec-1993   15-Jun-1994   15-Dec-1994   15-Jun-1995

Column 6

   15-Dec-1994
   NaT


TFactors =

         0    0.2404    0.7403    1.2404    1.7403    2.2404
         0    0.2404    1.2404    2.2404    3.2404       NaN


CFlowFlags =

     0     3     3     3     3     4
     0     3     3     3     4   NaN
```

Compute the Cash Flow Structure and Time Factors for a Bond Portfolio Using Optional Name-Value Pairs

This example shows how to compute the cash flow structure and time factors for a bond portfolio that contains a corporate bond paying interest quarterly and a Treasury bond paying interest semiannually. This example uses the following Name-Value pairs for `Period`, `Basis`, `BusinessDayConvention`, and `AdjustCashFlowsBasis`.

```
Settle = '01-Jun-2010';
Maturity = ['15-Dec-2011';'15-Jun-2012'];
CouponRate= [0.06; 0.05];
Period = [4; 2];
Basis = [1; 0];

[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ...
cfamounts(CouponRate,Settle, Maturity, 'Period',Period, ...
'Basis', Basis, 'AdjustCashFlowsBasis', true,...
'BusinessDayConvention','modifiedfollow')

CFlowAmounts =

  Columns 1 through 7

   -1.2667    1.5000    1.5000    1.5000    1.5000    1.5000    1.5000
   -2.3077    2.4932    2.5068    2.4932    2.5000  102.5000       NaN

  Column 8

  101.5000
       NaN


CFlowDates =

  Columns 1 through 6

       734290      734304      734396      734487      734577      734669
       734290      734304      734487      734669      734852      735035

  Columns 7 through 8

       734761      734852
```

```
         NaN         NaN


TFactors =

  Columns 1 through 7

         0    0.0778    0.5778    1.0778    1.5778    2.0778    2.5778
         0    0.0769    1.0769    2.0769    3.0769    4.0769       NaN

  Column 8

    3.0778
       NaN


CFlowFlags =

     0     3     3     3     3     3     3     4
     0     3     3     3     3     4   NaN   NaN
```

Use `cfamounts` With a `CouponRate` Schedule

This example shows how to use `cfamounts` with a `CouponRate` schedule. For `CouponRate` and `Face` that change over the life of the bond, schedules for `CouponRate` and `Face` can be specified with an `NINST`-by-1 cell array, where each element is a `NumDates`-by-2 matrix where the first column is dates and the second column is associated rates.

```
CouponSchedule = {[datenum('15-Mar-2012') .04;datenum('15- Mar -2013') .05;...
datenum('15- Mar -2015') .06]}

CouponSchedule = 1x1 cell array
    {3x2 double}


cfamounts(CouponSchedule,'01-Mar-2011','15-Mar-2015' )

ans =

  Columns 1 through 7
```

```
   -1.8453    2.0000    2.0000    2.0000    2.5000    2.5000    3.0000

  Columns 8 through 10

    3.0000    3.0000  103.0000
```

Use `cfamounts` With a `Face` Schedule

This example shows how to use `cfamounts` with a `Face` schedule. For `CouponRate` and `Face` that change over the life of the bond, schedules for `CouponRate` and `Face` can be specified with an `NINST`-by-1 cell array, where each element is a `NumDates`-by-2 matrix where the first column is dates and the second column is associated rates.

```
FaceSchedule = {[datenum('15-Mar-2012') 100;datenum('15- Mar -2013') 90;...
datenum('15- Mar -2015') 80]}

FaceSchedule = 1x1 cell array
    {3x2 double}


cfamounts(.05,'01-Mar-2011','15-Mar-2015', 'Face', FaceSchedule)

ans =

  Columns 1 through 7

   -2.3066    2.5000    2.5000   12.5000    2.2500   12.2500    2.0000

  Columns 8 through 10

    2.0000    2.0000   82.0000
```

Use `cfamounts` to Generate the Cash Flows for a Sinking Bond

This example shows how to use `cfamounts` to generate the cash flows for a sinking bond.

```
[CFlowAmounts,CFDates,TFactors,CFFlags,CFPrincipal] = cfamounts(.05,'04-Nov-2010',...
{'15-Jul-2014';'15-Jul-2015'},'Face',{[datenum('15-Jul-2013') 100;datenum('15-Jul-2014'
90;datenum('15-Jul-2015') 80]})

CFlowAmounts =

  Columns 1 through 7

   -1.5217    2.5000    2.5000    2.5000    2.5000    2.5000   12.5000
   -1.5217    2.5000    2.5000    2.5000    2.5000    2.5000   12.5000

  Columns 8 through 11

    2.2500   92.2500       NaN       NaN
    2.2500   12.2500    2.0000   82.0000


CFDates =

  Columns 1 through 6

      734446      734518      734699      734883      735065      735249
      734446      734518      734699      734883      735065      735249

  Columns 7 through 11

      735430      735614      735795         NaN         NaN
      735430      735614      735795      735979      736160


TFactors =

  Columns 1 through 7

         0    0.3913    1.3913    2.3913    3.3913    4.3913    5.3913
         0    0.3913    1.3913    2.3913    3.3913    4.3913    5.3913

  Columns 8 through 11

    6.3913    7.3913       NaN       NaN
    6.3913    7.3913    8.3913    9.3913


CFFlags =
```

```
     0     3     3     3     3     3    13     3     4   NaN   NaN
     0     3     3     3     3     3    13     3    13     3     4

CFPrincipal =

     0     0     0     0     0     0    10     0    90   NaN   NaN
     0     0     0     0     0     0    10     0    10     0    80
```

- “Analyzing and Computing Cash Flows” on page 2-21

Input Arguments

`CouponRate` — Annual percentage rate used to determine coupons payable on a bond
decimal

Annual percentage rate used to determine the coupons payable on a bond, specified as decimal using a scalar or a `NBONDS`-by-`1` vector.

`CouponRate` is `0` for zero coupon bonds.

Note `CouponRate` and `Face` can change over the life of the bond. Schedules for `CouponRate` and `Face` can be specified with an `NBONDS`-by-`1` cell array, where each element is a `NumDates`-by-`2` matrix or cell array, where the first column is dates (serial date numbers or character vectors) and the second column is associated rates. The date indicates the last day that the coupon rate or face value is valid.

Data Types: `double` | `cell` | `char`

`Settle` — Settlement date of bond
serial date number | date character vector | datetime

Settlement date of the bond, specified as a scalar or a `NBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of bond
serial date number | date character vector | datetime

Maturity date of the bond, specified as a scalar or a `NBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ... cfamounts(CouponRate,Settle, Maturity,'Period',4,'Basis', 3,'AdjustCashFlowsBasis',true,'BusinessDayConvention','modifiedfollo w')`

`Period` — Number of coupon payments per year for bond
`2` (default) | numeric with values `0`, `1`, `2`, `3`, `4`, `6` or `12`

Number of coupon payments per year for the bond, specified as scalar or a `NBONDS`-by-`1` vector using the values: `0`, `1`, `2`, `3`, `4`, `6`, or `12`.

Data Types: `double`

`Basis` — Day-count basis of bond
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count basis of the bond, specified as scalar or a `NBONDS`-by-`1` vector using a supported value:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a scalar or a `NBONDS`-by-`1` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime

Bond issue date (the date the bond begins to accrue interest), specified as a scalar or a `NBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays. The `IssueDate` cannot be after the `Settle` date.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
serial date number | date character vector | datetime

Irregular or normal first coupon date, specified as a scalar or a `NBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
serial date number | date character vector | datetime

Irregular or normal last coupon date, specified as a scalar or a `NBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`StartDate` — Forward starting date of coupon payments
serial date number | date character vector | datetime

Forward starting date of coupon payments after the `Settle` date, specified as a scalar or a `NBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Note To make an instrument forward starting, specify `StartDate` as a future date.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

Data Types: `double` | `char` | `datetime`

`Face` — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as a scalar or a `NBONDS`-by-`1` vector.

Note `CouponRate` and `Face` can change over the life of the bond. Schedules for `CouponRate` and `Face` can be specified with an `NBONDS`-by-`1` cell array where each element is a `NumDates`-by-`2` matrix or cell array, where the first column is dates (serial date numbers or character vectors) and the second column is associated rates. The date indicates the last day that the coupon rate or face value is valid.

Data Types: `double` | `cell` | `char`

`AdjustCashFlowsBasis` — Adjusts cash flows according to accrual amount based on actual period day count

`0` (false) (default) | nonnegative integer `0` or `1`

Adjusts cash flows according to the accrual amount based on the actual period day count, specified as a scalar or a `NBONDS`-by-`1` vector.

Data Types: `logical`

`BusinessDayConvention` — Business day conventions

`'actual'` (default) | character vector with values`'actual'`, `'follow'`, `'modifiedfollow'`, `'previous'`or `'modifiedprevious'`

Business day conventions, specified as a scalar or `NBONDS`-by-`1` cell array of character vectors of business day conventions to be used in computing payment dates. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `'actual'` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

`CompoundingFrequency` — Compounding frequency for yield calculation
SIA uses`2`, ICMA uses `1` (default) | integer with value of `1`, `2`, `3`, `4`, `6`, or `12`

Compounding frequency for yield calculation, specified as a scalar or a `NBONDS`-by-`1` vector. Values are:

- `1` — Annual compounding
- `2` — Semiannual compounding
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Data Types: `double`

`DiscountBasis` — Basis used to compute the discount factors for computing the yield
SIA uses `0` (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Basis used to compute the discount factors for computing the yield, specified as a scalar or a `NBONDS`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If a SIA day-count basis is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the default behavior is for SIA bases to use the `actual/actual` day count to compute discount factors.

If an ICMA day-count basis or BUS/252 is defined in the `Basis` input argument and there is no value assigned for `DiscountBasis`, the specified bases from the `Basis` input argument are used.

Data Types: `double`

`Holidays` — Dates for holidays
`holidays.m` used (default)

Dates for holidays, specified as `NHOLIDAYS`-by-`1` vector of MATLAB dates using serial date numbers, date character vectors, or datetime arrays. Holidays are used in computing business days.

Data Types: `double` | `char` | `datetime`

`PrincipalType` — Type of principal when a `Face` schedule is specified
`sinking` (default) | character vector with values `'sinking'` or `'bullet'`

Type of principal when a `Face` schedule is specified. `PrincipalType` is specified as `'sinking'` or `'bullet'` using a scalar or a `NBONDS`-by-`1` vector.

If `'sinking'`, principal cash flows are returned throughout the life of the bond.

If `'bullet'`, principal cash flow is only returned at maturity.

Data Types: `char` | `cell`

Output Arguments

`CFlowAmounts` — Cash flow amounts

matrix

Cash flow amounts, returned as a `NBONDS`-by-`NCFS` (number of cash flows) matrix. The first entry in each row vector is the accrued interest due at settlement. This amount could be zero, positive or negative. If no accrued interest is due, the first column is zero. If the bond is trading ex-coupon then the accrued interest is negative.

`CFlowDates` — Cash flow dates for a portfolio of bonds

matrix

Cash flow dates for a portfolio of bonds, returned as a `NBONDS`-by-`NCFS` matrix. Each row represents a single bond in the portfolio. Each element in a column represents a cash flow date of that bond.

If all the above inputs (`Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, `LastCouponDate`, and `StartDate`) are either serial date numbers or date character vectors, then `CFlowDates` is returned as a serial date number. If any of these inputs are datetime arrays, then `CFlowDates` is returned as a datetime array.

`TFactors` — Matrix of time factors for a portfolio of bonds

matrix

Matrix of time factors for a portfolio of bonds, returned as a `NBONDS`-by-`NCFS` matrix. 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 for price/yield conversion and time factors are in units of whole semiannual coupon periods plus any fractional period using an actual day count. For more information on time factors, see “Time Factors” on page 18-379.

`CFlowFlags` — Cash flow flags for a portfolio of bonds

matrix

Cash flow flags for a portfolio of bonds, returned as a `NBONDS`-by-`NCFS` matrix. Each row corresponds to the vector of cash flow flags for each bond. Each element in a column corresponds to the specific flag associated with each cash flow of a bond. Flags identify the type of each cash flow (for example, nominal coupon cash flow, front, or end partial, or "stub" coupon, maturity cash flow).

Flag	Cash Flow Type
0	Accrued interest due on a bond at settlement.
1	Initial cash flow amount smaller than normal due to a “stub” coupon period. A stub period is created when the time from issue date to first coupon date is shorter than normal.
2	Larger than normal initial cash flow amount because the first coupon period is longer than normal.
3	Nominal coupon cash flow amount.
4	Normal maturity cash flow amount (face value plus the nominal coupon amount).
5	End “stub” coupon amount (last coupon period is abnormally short and actual maturity cash flow is smaller than normal).
6	Larger than normal maturity cash flow because the last coupon period longer than normal.
7	Maturity cash flow on a coupon bond when the bond has less than one coupon period to maturity.
8	Smaller than normal maturity cash flow when the bond has less than one coupon period to maturity.
9	Larger than normal maturity cash flow when the bond has less than one coupon period to maturity.
10	Maturity cash flow on a zero coupon bond.
11	Sinking principal and initial cash flow amount smaller than normal due to a "stub" coupon period. A stub period is created when the time from issue date to first coupon date is shorter than normal.
12	Sinking principal and larger than normal initial cash flow amount because the first coupon period is longer than normal.
13	Sinking principal and nominal coupon cash flow amount.

`CFPrincipal` — Principal cash flows

matrix

Principal cash flows, returned as a `NBONDS`-by-`NCFS` matrix.

If `PrincipalType` is `'sinking'`, `CFPrincipal` output indicates when the principal is returned.

If `PrincipalType` is `'bullet'`, `CFPrincipal` is all zeros and, at `Maturity`, the appropriate `Face` value.

Definitions

Time Factors

Time factors help determine the present value of a stream of cash flows.

The term time factors refer to the exponent *TF* in the discounting equation

$$PV = \sum_{i=1}^{n} \left(\frac{CF}{(1 + \frac{z}{f})^{TF}} \right),$$

where:

PV =	Present value of a cash flow.
CF =	Cash flow amount.
z =	Risk-adjusted annualized rate or yield corresponding to a given cash flow. The yield is quoted on a semiannual basis.
f =	Frequency of quotes for the yield. Default is `2` for `Basis` values `0` to `7` and `13` and `1` for `Basis` values `8` to `12`. The default can be overridden by specifying the `CompoundingFrequency` name-value pair.
TF =	Time factor for a given cash flow. The time factor is computed using the compounding frequency and the discount basis. If these values are not specified, then the defaults are as follows: `CompoundingFrequency` default is `2` for `Basis` values `0` to `7` and `13` and `1` for `Basis` values `8` to `12`. `DiscountBasis` is `0` for `Basis` values `0` to `7` and `13` and the value of the input `Basis` for `Basis` values `8` to `12`.

Note The `Basis` is always used to compute accrued interest.

References

[1] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[2] Mayle, J. *"Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures."* SIA, Vol 2, Jan 1994.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`accrfrac` | `cfdates` | `cftimes` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

cfconv

Cash flow convexity

Syntax

```
CFlowConvexity = cfconv(CashFlow,Yield)
```

Arguments

`CashFlow`	A vector of real numbers.
`Yield`	Periodic yield. A scalar. Enter as a decimal fraction.

Description

`CFlowConvexity = cfconv(CashFlow,Yield)` returns the convexity of a cash flow in periods.

Examples

Compute the Convexity of a Cash Flow

This example shows how to return the convexity of a cash flow, given a cash flow of nine payments of $2.50 and a final payment $102.50, with a periodic yield of 2.5%.

```
CashFlow = [2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 102.5];

Convex = cfconv(CashFlow, 0.025)

Convex = 90.4493
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bnddury` | `cfdur`

Topics

"Analyzing and Computing Cash Flows" on page 2-21

Introduced before R2006a

cfdates

Cash flow dates for fixed-income security

Syntax

```
CFlowDates = cfdates(Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstCouponDat
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers, date character vectors, or datetime arrays.
`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) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 For more information, see **basis** on page Glossary-0 .
`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, specified as a serial date number, date character vector, or datetime array, when a bond was issued.

`FirstCouponDate`	(Optional) Date, specified as a serial date number, date character vector, or datetime array, when a bond makes its first coupon payment. `FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.
`LastCouponDate`	(Optional) Last coupon date of a bond before the maturity date, specified as a serial date number, date character vector, or datetime array. `LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.
`StartDate`	(Optional) Date, specified as a serial date number, date character vector, or datetime array, when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.

Required arguments must be number of bonds (`NUMBONDS`)-by-`1` or `1`-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `Maturity` contains N dates, then `Settle` must contain N dates or a single date.

Description

`CFlowDates = cfdates(Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstCouponDate,LastCouponDate,StartDate)` returns a matrix of cash flow dates for a bond or set of bonds. `cfdates` determines all cash flow dates for a bond whether or not the coupon payment structure is normal or the first and/or last coupon period is long or short.

`CFlowDates` is an N-row matrix of serial date numbers, padded with `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 character vectors.

If all of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, `LastCouponDate`, and `StartDate` are either serial date numbers or date character vectors, then `CFlowDates` is returned as a serial date number.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, `LastCouponDate`, or `StartDate` are datetime arrays, then `CFlowDates` is returned as a datetime array.

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
```

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, `LastCouponDate`, or `StartDate` are datetime arrays, then `CFlowDates` is returned as a datetime array. For example:

```
CFlowDates = cfdates('14-Mar-1997', datetime('30-Nov-1998','Locale','en_US'), 2, 0, 1)

CFlowDates =

   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` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

cfdatesq

Quasi-coupon dates for fixed-income security

Syntax

```
QuasiCouponDates = cfdatesq(Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstCo
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers, date character vectors, or datetime arrays.
`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) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 For more information, see **basis** on page Glossary-0 .
`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, specified as a serial date number, date character vector, or datetime array, when a bond was issued.

`FirstCouponDate`	(Optional) Date, specified as a serial date number, date character vector, or datetime array, when a bond makes its first coupon payment. `FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.
`LastCouponDate`	(Optional) Last coupon date of a bond before the maturity date, specified as a serial date number, date character vector, or datetime array. `LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.
`PeriodsBeforeSettle`	(Optional) Number of quasi-coupon dates on or before settlement to include (non-negative integer); default is `0`.
`PeriodsAfterMaturity`	(Optional) Number of quasi-coupon dates after maturity to include (non-negative integer); default is `0`.

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

```
QuasiCouponDates =
cfdatesq(Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstCo
```

uponDate,LastCouponDate, PeriodsBeforeSettle,PeriodsAfterMaturity) returns a matrix of quasi-coupon dates expressed in serial date format (default) or datetime format (if any inputs are in datetime format).

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. Quasi-coupon dates are determined regardless of whether the first or last coupon periods are normal, long, or short.

QuasiCouponDates has NUMBONDS rows and the number of columns is determined by the maximum number of quasi-coupon dates required to hold the bond portfolio. NaNs are padded for bonds which have less than the maximum number quasi-coupon dates. By default, quasi- coupon dates after settlement and on or preceding maturity are returned. If settlement occurs on maturity, and maturity is a quasi-coupon date, then the maturity date is returned.

If the date inputs for Settle, Maturity, IssueDate, FirstCouponDate, and LastCouponDate are either serial date numbers or date character vectors, then QuasiCouponDates is returned as a serial date number.

If any of the date inputs for Settle, Maturity, IssueDate, FirstCouponDate, or LastCouponDate are datetime arrays, then QuasiCouponDates is returned as a datetime array.

Examples

```
QuasiCouponDates = cfdatesq('14-Mar-1997', '30-Nov-1998', 2, 0, 1)
```

```
QuasiCouponDates =

       729541      729724      729906      730089
```

If any of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, or LastCouponDate are datetime arrays, then CFlowDates is returned as a datetime array. For example:

```
QuasiCouponDates = cfdatesq('14-Mar-1997', datetime('30-Nov-1998','Locale','en_US'), 2, 0, 1)
```

```
QuasiCouponDates =

   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 | datetime

Topics

"Analyzing and Computing Cash Flows" on page 2-21

Introduced before R2006a

cfdur

Cash-flow duration and modified duration

Syntax

```
[Duration,ModDuration] = cfdur(CashFlow,Yield)
```

Arguments

`CashFlow`	A vector or matrix of real numbers. When using a matrix, each column of the matrix is a separate `CashFlow`.
`Yield`	Periodic yield. A scalar or vector. Enter as a decimal fraction.

Description

`[Duration,ModDuration] = cfdur(CashFlow,Yield)` calculates the duration and modified duration of a cash flow in periods.

Examples

Compute the Duration and Modified Duration of a Cash Flow

This example shows how to calculate the duration and modified duration of a cash flow, given a cash flow of nine payments of $2.50 and a final payment $102.50, with a periodic yield of 2.5%.

```
CashFlow=[2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 102.5];

[Duration, ModDuration] = cfdur(CashFlow, 0.025)

Duration = 8.9709
```

```
ModDuration = 8.7521
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bnddury` | `cfconv`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

cfplot

Visualize cash flows of financial instruments

Syntax

```
cfplot(CFlowDates,CFlowAmounts)
cfplot(___,Name,Value)

h = cfplot(___,Name,Value)
[h,axes_handle] = cfplot(___,Name,Value)
```

Description

`cfplot(CFlowDates,CFlowAmounts)` plots a cash flow diagram for the specified cash flow amounts (`CFlowAmounts`) and dates (`CFlowDates`). The length and orientation of each arrow correspond to the cash flow amount.

`cfplot(___,Name,Value)` plots a cash flow diagram for the specified cash flow amounts (`CFlowAmounts`), dates (`CFlowDates`), and optional name-value pair arguments.

`h = cfplot(___,Name,Value)` returns the handle to the line objects used in the cash flow diagram.

`[h,axes_handle] = cfplot(___,Name,Value)` returns the handles to the line objects and the axes using optional name-value pair arguments.

Examples

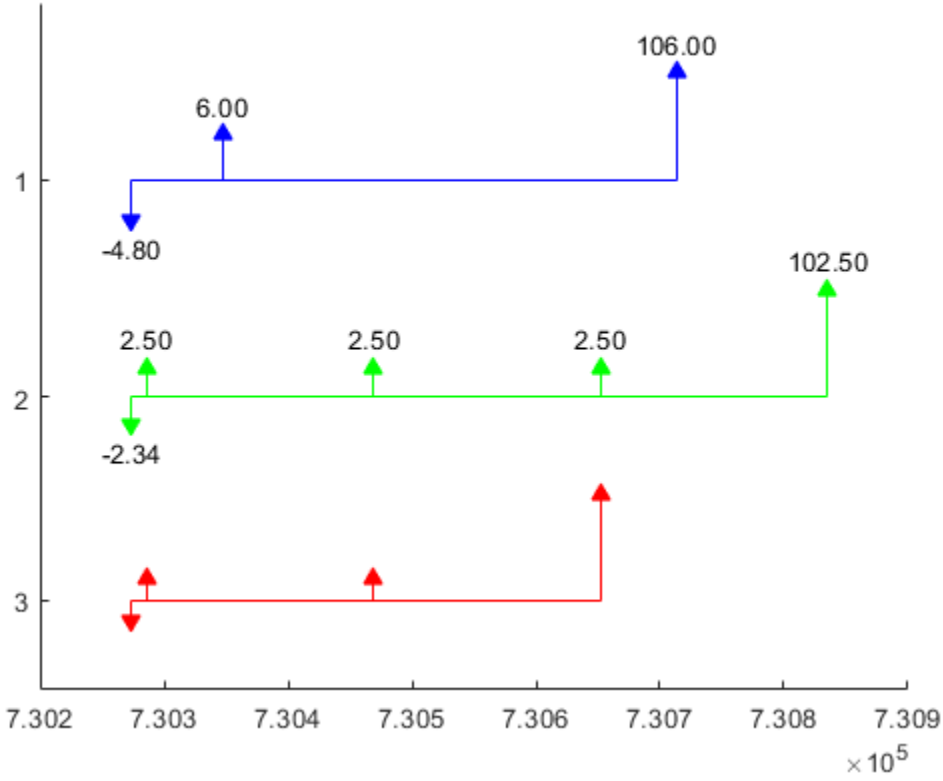
Plot Cash Flows

Define `CFlowAmounts` and `CFlowDates` using the `cfamounts` function.

```
CouponRate = [0.06; 0.05; 0.03];
Settle = '03-Jun-1999';
Maturity = ['15-Aug-2000';'15-Dec-2000';'15-Jun-2000'];
Period = [1; 2; 2];  Basis = [1; 0; 0];
[CFlowAmounts, CFlowDates] = cfamounts(...
CouponRate, Settle, Maturity, Period, Basis)

CFlowAmounts =

   -4.8000    6.0000  106.0000       NaN       NaN
   -2.3352    2.5000    2.5000    2.5000  102.5000
   -1.4011    1.5000    1.5000  101.5000       NaN


CFlowDates =

      730274      730347      730713         NaN         NaN
      730274      730286      730469      730652      730835
      730274      730286      730469      730652         NaN
```

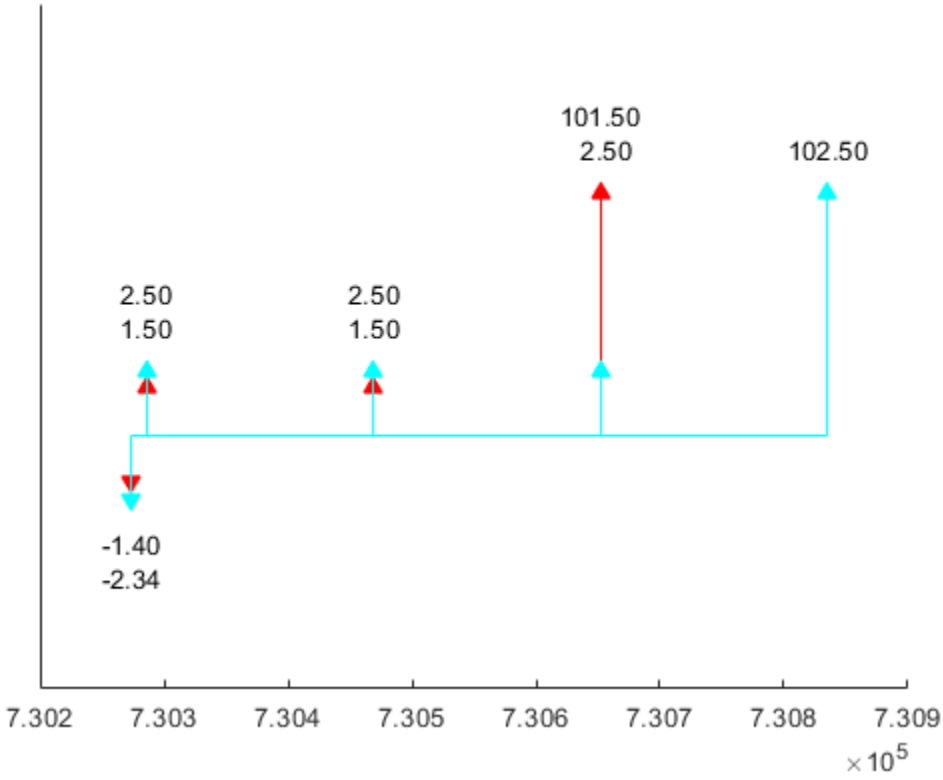
Plot all cash flows on the same axes, and label the first two.

```
cfplot(CFlowDates, CFlowAmounts, 'ShowAmnt', [1 2])
```



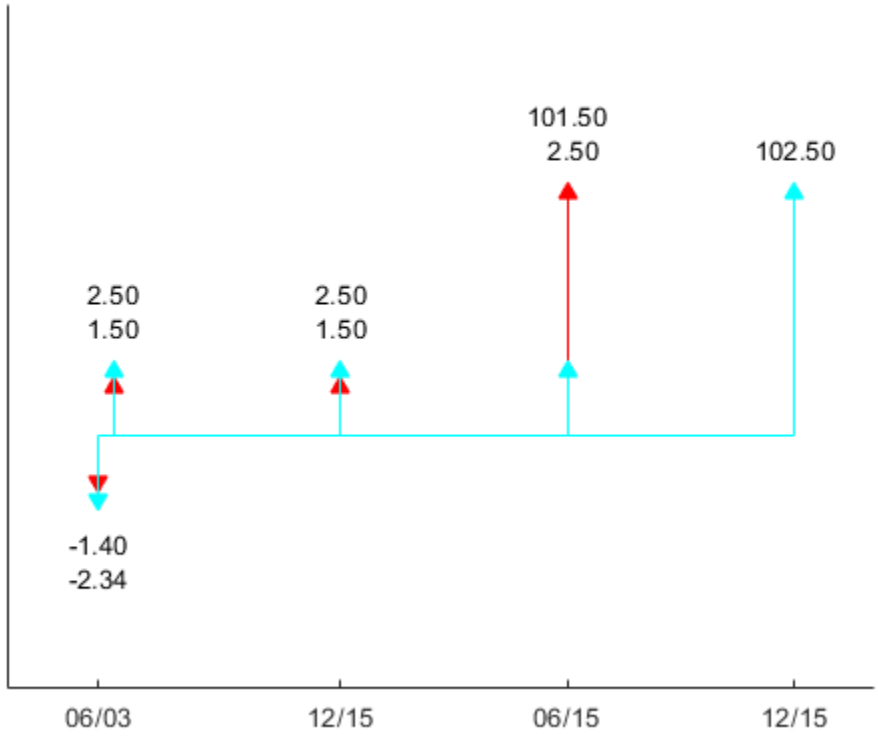
Group the second and third cash flows.

```
figure;
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmnt', 1);
```



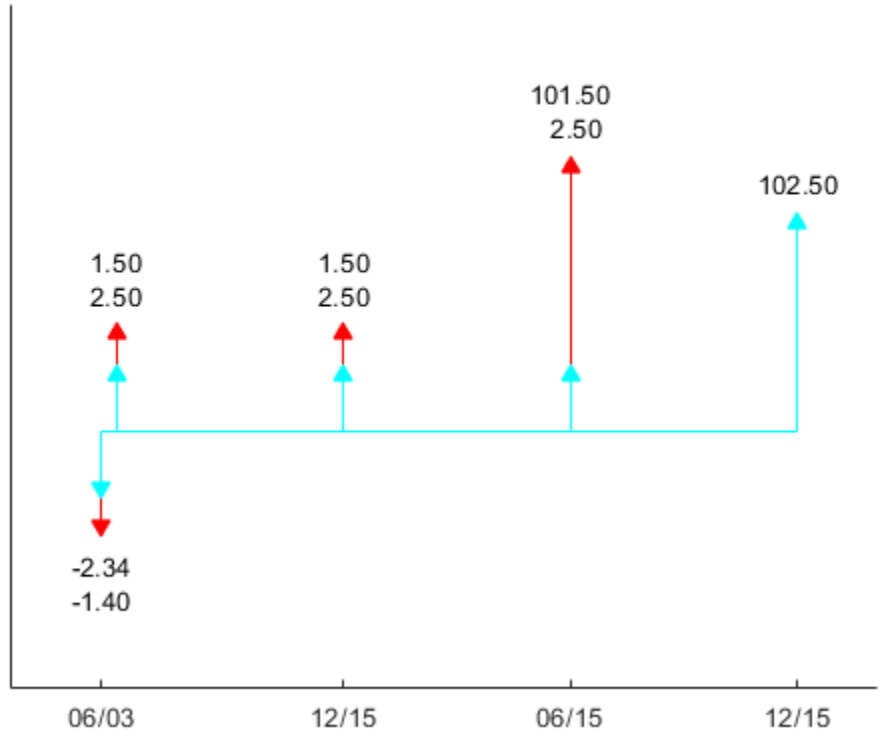
Format the date axis and place ticks on actual cash flow dates.

```
figure;
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmnt', 1, ...
'DateFormat', 6, 'DateSpacing', 100);
```



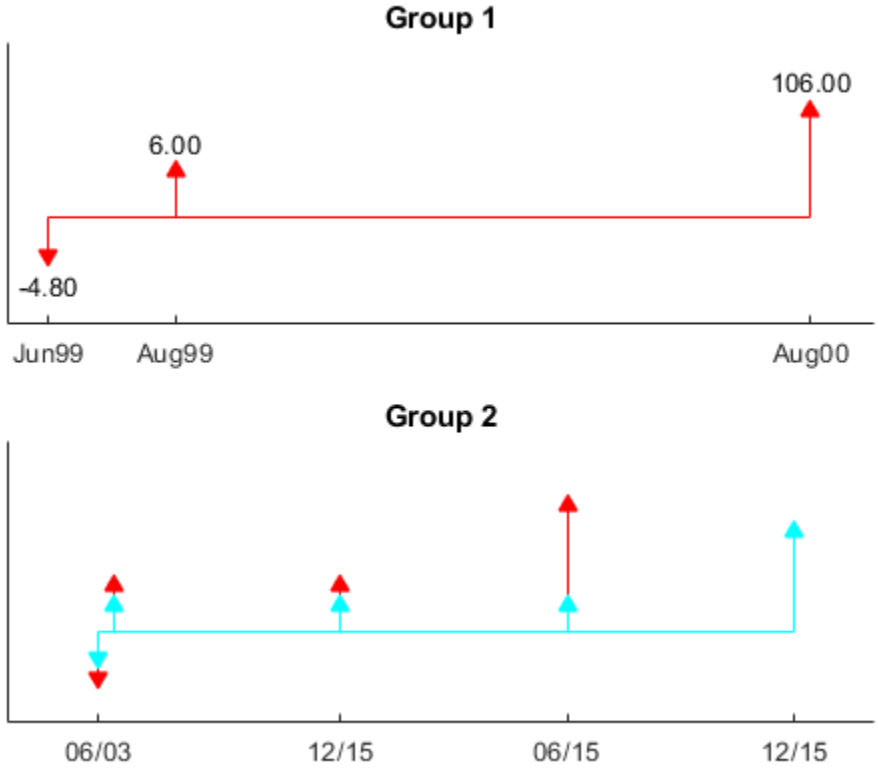
Stack the cash flow arrows occurring on the same dates.

```
figure;
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmnt', 1, ...
'DateFormat', 6, 'DateSpacing', 100, 'Stacked', 1);
```



Form subplots of multiple groups and add titles using axes handles.

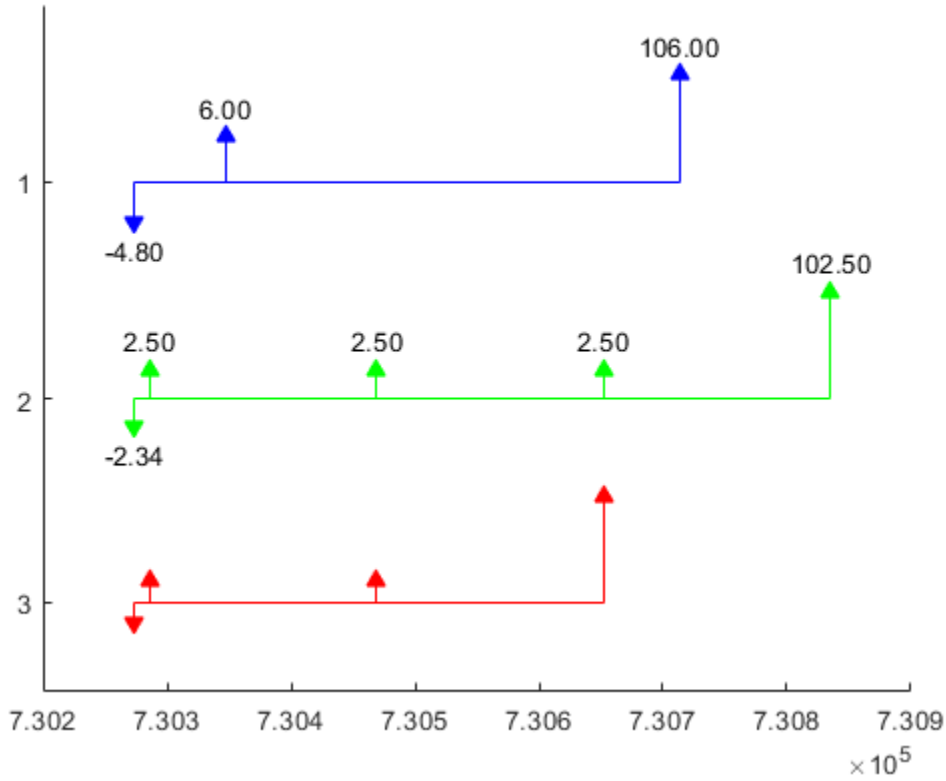
```
figure;
[h, axes_handle] = cfplot(CFlowDates, CFlowAmounts, ...
'Groups', {[1] [2 3]}, 'ShowAmnt', 1, 'Stacked', 2, ...
'DateSpacing', [1 60 2 100], 'DateFormat', [1 12 2 6]);
title(axes_handle(1), 'Group 1', 'FontWeight', 'bold');
title(axes_handle(2), 'Group 2', 'FontWeight', 'bold');
```



Plot Cash Flows Using datetime Input for CFlowDates

Define `CFlowDates` using `datetime` input and plot the cash flow.

```
CouponRate = [0.06; 0.05; 0.03];
Settle = '03-Jun-1999';
Maturity = ['15-Aug-2000';'15-Dec-2000';'15-Jun-2000'];
Period = [1; 2; 2];  Basis = [1; 0; 0];
[CFlowAmounts, CFlowDates] = cfamounts(...
CouponRate, Settle, Maturity, Period, Basis);
cfplot(datetime(CFlowDates,'ConvertFrom','datenum','Locale','en_US'), CFlowAmounts, 'Sh
```



Plot Cash Flows for Swap

Define the swap using the `swapbyzero` function.

```
Settle = datenum('08-Jun-2010');
RateSpec = intenvset('Rates', [.005 .0075 .01 .014 .02 .025 .03]',...
'StartDates',Settle, 'EndDates',{'08-Dec-2010','08-Jun-2011',...
'08-Jun-2012','08-Jun-2013','08-Jun-2015','08-Jun-2017',...
'08-Jun-2020'}');
Maturity = datenum('15-Sep-2020');
LegRate = [.025 50];
```

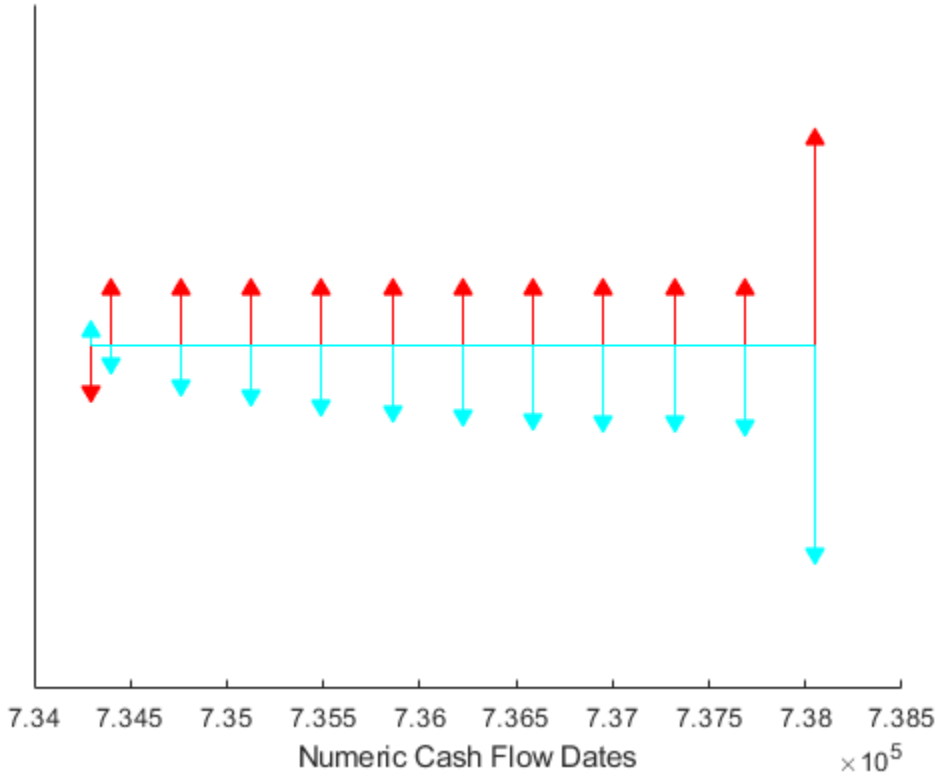
```
LegType = [1 0]; % fixed/floating
LatestFloatingRate = .005;
[Price, SwapRate, AI, RecCF, RecCFDates, PayCF,PayCFDates] = ...
swapbyzero(RateSpec, LegRate, Settle, Maturity,'LegType',LegType,...
'LatestFloatingRate',LatestFloatingRate)
```

```
Price = -6.7258

SwapRate = NaN

AI = 1.4575

RecCF =

  Columns 1 through 7

   -1.8219    2.5000    2.5000    2.5000    2.5000    2.5000    2.5000

  Columns 8 through 12

    2.5000    2.5000    2.5000    2.5000  102.5000


RecCFDates =

  Columns 1 through 6

      734297      734396      734761      735127      735492      735857

  Columns 7 through 12

      736222      736588      736953      737318      737683      738049


PayCF =

  Columns 1 through 7

   -0.3644    0.5000    1.4048    1.9823    2.8436    3.2842    3.8218

  Columns 8 through 12

    4.1733    4.5164    4.4666    4.8068  104.6743


PayCFDates =
```

```
Columns 1 through 6

    734297      734396      734761      735127      735492      735857

Columns 7 through 12

    736222      736588      736953      737318      737683      738049
```

Define `CFlowDates` and `CFlowAmounts` for the swap and generate a cash flow plot using `cfplot`.

```
CFlowDates = [PayCFDates;RecCFDates];
CFlowAmounts = [-PayCF;RecCF];
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[1 2]});
xlabel('Numeric Cash Flow Dates');
```



- “Analyzing and Computing Cash Flows” on page 2-21

Input Arguments

`CFlowDates` — Matrix of serial date numbers for cash flows

vector

Matrix of serial date numbers or datetime arrays for cash flows, specified as a `NINST`-by-`(Number of cash flows)` matrix of cash flow dates in date numbers, with empty entries padded with `NaN`s.

Each row of the `CFlowDates` matrix represents an instrument so that `CFlowDates(k,:)` is the vector of cash flow dates for the `k`th instrument. Rows are padded with trailing `NaN`s if the number of cash flows is not the same for all instruments.

`cfamounts` can be used to generate `CFlowDates`.

Data Types: `double`

`CFlowAmounts` — Matrix of cash flow amounts
vector

Matrix of cash flow amounts, specified as a `NINST`-by-`(Number of cash flows)` matrix of cash flow amounts, with empty entries padded with `NaN`s. The `CFlowAmounts` matrix must be the same size as `CFlowDates`.

`cfamounts` can be used to generate `CFlowAmounts`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `cfplot(CFlowDates,CFlowAmounts,'Groups',{[2 3]},'ShowAmnt',1,'DateFormat',6,'DateSpacing',100)`

`Groups` — Group cash flows
`'off'` (default) | character vector with value `'off'` or `'individual'` | cell array of character vectors

Group cash flows specified using the following values:

- `'off'` — Show all instruments in one set of axes, arranged from top to bottom.
- `'individual'` — Generate subplots and plot each instrument in its own axis.
- `GRP` — Cell array of instrument groups, `{Group1, Group2,... }`. This generates subplots and plots each group in each axis. When specifying `{Group1, Group2,... }`, each Group must be mutually exclusive vectors of `INSTIndex`. Unspecified instruments are not shown in the grouped plot.

Data Types: `char` | `cell`

`Stacked` — Stack arrows if cash flows are in same direction on same day
ignored when `'Groups'` is `'off'`, otherwise `'off'` (default) | character vector with values `'off'`, `'all'`, or `'GRPIndex'`

Stack arrows if the cash flows are in the same direction on the same day specified using the following values:

- `'off'` — For all groups, all arrows originate from the horizontal line.
- `'all'` — For all groups, arrows are stacked if the cash flows are in the same direction on the same day.
- `'GRPIndex'` — For specified groups, arrows are stacked if the cash flows are in the same direction on the same day.

Data Types: `char`

`ShowAmnt` — Show amount on arrows
`'off'` (default) | character vector with values `'off'` or `'individual'` | cell array of character vectors

Show amount on the arrows specified using the following values:

- `'off'` — Hide cash flow amounts on arrows.
- `'all'` — Show cash flow amounts on arrows.
- [`INSTIndex` or `GRPIndex`] — Show cash flow amounts for the specified vector of instruments (when `'Groups'` is `'off'`) or groups.

Data Types: `char` | `cell`

`DateSpacing` — Control for date axis tick spacing
`'off'` (default) | character vector with values `'off'` or `TickDateSpace` | numeric value for `TickDateSpace`

Control for data spacing specified by the following values:

- `'off'` — The date axis ticks are spaced regularly.
- `TickDateSpace` — The date axis ticks are placed on actual cash flow dates. The ticks skip some cash flows if they are less than `TickDateSpace` apart.

Data Types: `char` | `double`

`DateFormat` — Date format

`'off'` (default) | character vector with values `'off'` or `DateFormNum` | numeric value for `DateFormNum`

Date format is specified by the following values:

- `'off'` — The date axis tick labels are in date numbers.
- `DateFormNum` — The date format number (`2` = `'mm/dd/yy'`, `6` = `'mm/dd'`, and `10` = `'yyyy'`). Additional values for `DateFormNum` are as follows:

DateFormNum	Example
2	03/01/00
3	Mar
5	03
6	03/01
7	01
8	Wed
9	W
10	2000
11	00
12	Mar00
17	Q1–00
18	Q1
19	01/03
20	01/03/00
27	Q1–2000
28	Mar2000
29	2000–03–01

Data Types: `char` | `double`

Output Arguments

`h` — Handles to line objects

vector

Handles to line objects, returned as a `NINST`-by-`3` matrix of handles to line objects, containing `[hLines, hUArrowHead, hDArrowHead]` where:

- `hLines` — Horizontal and vertical lines used in the cash flow diagram
- `hUArrowHead` — "Up" arrowheads
- `hDArrowHead` — "Down" arrowheads

`axes_handle` — Handles to axes for plot or subplots

vector

Handles to axes for the plot or subplots, returned as a (`Number of axes`)-by-`1` vector of handles to axes.

See Also

`cfamounts` | `cfdates` | `datetime` | `swapbyzero`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced in R2013a

cfport

Portfolio form of cash flow amounts

Syntax

```
[CFBondDate,AllDates,AllTF,IndByBond] = cfport(CFlowAmounts,CFlowDates,TFactors)
```

Arguments

`CFlowAmounts`	Number of bonds (`NUMBONDS`) by number of cash flows (`NUMCFS`) matrix with entries listing cash flow amounts corresponding to each date in `CFlowDates`.
`CFlowDates`	`NUMBONDS`-by-`NUMCFS` matrix with rows listing cash flow dates, specified as a serial date number, date character vector, or datetime array, for each bond and padded with `NaN`s. If `CFlowDates` is a serial date number or a date character vector, `AllDates` is returned as an array of serial date numbers. If `CFlowDates` is a datetime array, then `AllDates` is returned as a datetime array.
`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`.

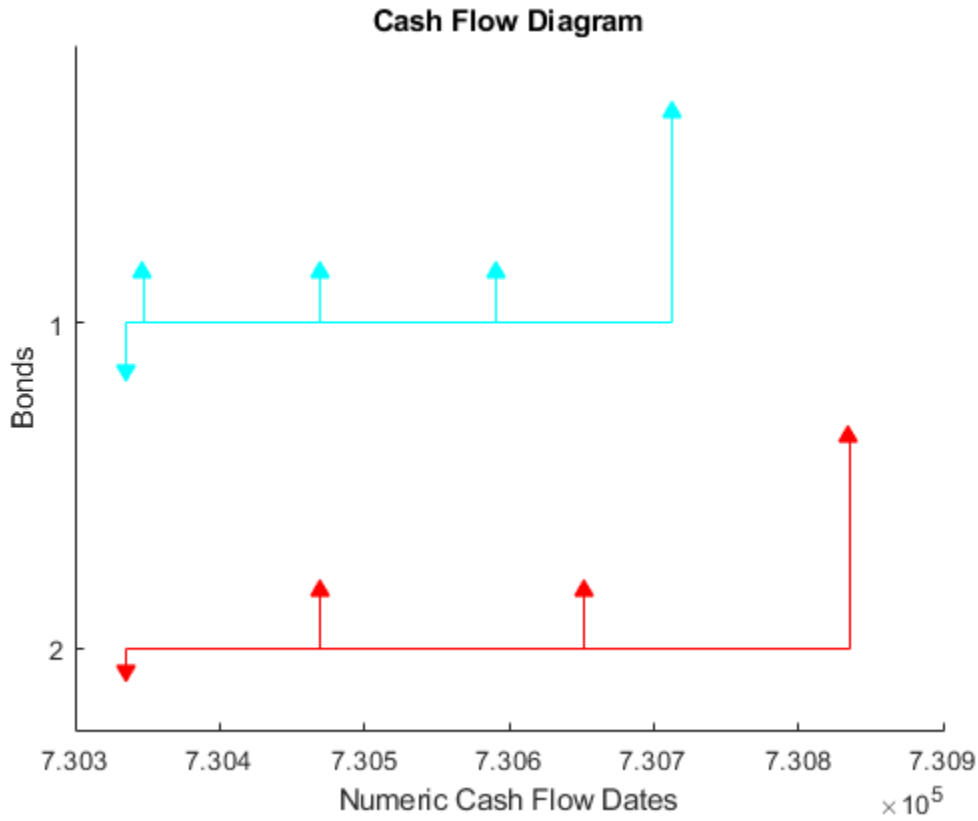
`IndByBond` is a `NUMBONDS`-by-`NUMCFS` matrix of indices. The *i*th row contains a list of indices into `AllDates` where the *i*th bond has cash flows. Since some bonds have more cash flows than others, the matrix is padded with `NaN`s.

Examples

Calculate the Cash Flow Amounts, Cash Flow Dates, and Time Factors for Each of Two Bonds

Use the function `cfamounts` to calculate the cash flow amounts, cash flow dates, and time factors for each of two bonds. Then use the function `cfplot` to plot the cash flow diagram.

```
Settle = '03-Aug-1999';
Maturity = ['15-Aug-2000';'15-Dec-2000'];
CouponRate= [0.06; 0.05];
Period = [3;2];
Basis = [1;0];
[CFlowAmounts, CFlowDates, TFactors] = cfamounts(CouponRate,...
Settle, Maturity, Period, Basis);
cfplot(CFlowDates,CFlowAmounts)
xlabel('Numeric Cash Flow Dates')
ylabel('Bonds')
title('Cash Flow Diagram')
```



Call the function `cfport` to map the cash flow amounts to the cash flow dates. Each row in the resultant `CFBondDate` matrix represents a bond. Each column represents a date on which one or more of the bonds has a cash flow. A 0 means the bond did not have a cash flow on that date. The dates associated with the columns are listed in `AllDates`. For example, the first bond had a cash flow of `2.000` on `730347`. The second bond had no cash flow on this date For each bond, `IndByBond` indicates the columns of `CFBondDate`, or dates in `AllDates`, for which a bond has a cash flow.

```
[CFBondDate, AllDates, AllTF, IndByBond] = ...
cfport(CFlowAmounts, CFlowDates, TFactors)

CFBondDate =
```

```
   -1.8000    2.0000    2.0000    2.0000         0  102.0000         0
   -0.6694         0    2.5000         0    2.5000         0  102.5000


AllDates =

      730335
      730347
      730469
      730591
      730652
      730713
      730835


AllTF =

         0
    0.0663
    0.7322
    1.3989
    1.7322
    2.0663
    2.7322


IndByBond =

     1     2     3     4     6
     1     3     5     7   NaN
```

Calculate the Cash Flow Amounts, Cash Flow Dates Using a datetime Array, and Time Factors for Each of Two Bonds

Use the function `cfamounts` to calculate the cash flow amounts, cash flow dates, and time factors for each of two bonds.

```
Settle = datetime('03-Aug-1999','Locale','en_US');
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);
```

Call the function `cfport` to map the cash flow amounts to the cash flow dates. Each row in the resultant `CFBondDate` matrix represents a bond. Each column represents a date on which one or more of the bonds has a cash flow. A `0` means the bond did not have a cash flow on that date. The dates associated with the columns are listed in `AllDates` returned as a datetime array.

```
[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 = 7x1 datetime array
   03-Aug-1999
   15-Aug-1999
   15-Dec-1999
   15-Apr-2000
   15-Jun-2000
   15-Aug-2000
   15-Dec-2000


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
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`cfamounts` | `cfplot` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

cfprice

Compute price for cash flow given yield to maturity

Syntax

```
Price = cfprice(CFAmounts,CFDates,Yield,Settle)
Price = cfprice(CFAmounts,CFDates,Yield,Settle,Name,Value)
```

Description

`Price = cfprice(CFAmounts,CFDates,Yield,Settle)` computes a price given yield for a cash flow.

`Price = cfprice(CFAmounts,CFDates,Yield,Settle,Name,Value)` computes a price for a cash flow given yield to maturity with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`CFlowAmounts`

`NINST`-by-`MOSTCFS` matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than `MOSTCFS` cash flows, the end of the row is padded with `NaN`s.

`CFlowDates`

`NINST`-by-`MOSTCFS` matrix of cash flow dates, specified as a serial date number, date character vector, or datetime array. Each entry contains the date of the corresponding cash flow in `CFlowAmounts`.

`Yield`

`NINST`-by-`1` vector of yields.

`Settle`

Settlement date, specified as a serial date number, date character vector, or datetime array. Settlement date is the date on which the cash flows are priced.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note Any optional input of size `N`-by-`1` is also acceptable as an array of size `1`-by-`N`, or as a single value applicable to all contracts. Single values are internally expanded to an array of size `N`-by-`1`.

`Basis`

`N`-by-`1` vector of day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Default: `0` (actual/actual)

`CompoundingFrequency`

`NINST`-by-`1` vector of Compounding Frequency. By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Default: `2`

Output Arguments

`Price`

Price of cash flows.

Examples

Compute the Price for a Cash Flow Given Yield to Maturity

Use `cfprice` to compute the price for a cash flow given yield to maturity.

Define data for the yield curve.

```
Settle = datenum('01-Jul-2003');
Yield = .05;
CFAmounts = [30;40;30];
CFDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'});
```

Compute the `Price`.

```
Price = cfprice(CFAmounts, CFDates, Yield, Settle)
```

```
Price =

   28.4999
   36.1689
```

```
   25.8195
```

Compute the Price for a Cash Flow Given Yield to Maturity Using datetime Inputs

Use `cfprice` to compute the price for a cash flow given yield to maturity using `datetime` inputs.

```
Settle = datenum('01-Jul-2003');
Yield = .05;
CFAmounts = [30;40;30];
CFDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'});

CFDates = datetime(CFDates,'ConvertFrom','datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
Price = cfprice(CFAmounts, CFDates, Yield, Settle)

Price =

   28.4999
   36.1689
   25.8195
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`cfbyzero` | `cfspread` | `cfyield` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced in R2012a

cfspread

Compute spread over yield curve for cash flow

Syntax

```
Spread = cfspread(RateSpec,Price,CFAmounts,CFDates,Settle)
Spread = cfspread(RateSpec,Price,CFAmounts,CFDates,Settle,
Name,Value)
```

Description

`Spread = cfspread(RateSpec,Price,CFAmounts,CFDates,Settle)` computes spread over a yield curve for a cash flow.

`Spread = cfspread(RateSpec,Price,CFAmounts,CFDates,Settle,
Name,Value)` computes spread over a yield curve for a cash flow with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`RateSpec`

Interest-rate specification for the initial risk free rate curve. See `intenvset` for information on declaring an interest-rate variable.

`Price`

Price of cash flows.

`CFlowAmounts`

`NINST`-by-`MOSTCFS` matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than `MOSTCFS` cash flows, the end of the row is padded with `NaN`s.

CFlowDates

NINST-by-MOSTCFS matrix of cash flow dates, specified as a serial date number, date character vector, or datetime array. Each entry contains the date of the corresponding cash flow in CFlowAmounts.

Settle

Settlement date, specified as a serial date number, date character vector, or datetime array. Settlement date is the date on which the cash flows are priced.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Note Any optional input of size N-by-1 is also acceptable as an array of size 1-by-N, or as a single value applicable to all contracts. Single values are internally expanded to an array of size N-by-1.

Basis

N-by-1 vector of day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)

- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Default: `0` (actual/actual)

`CompFreq`

Compounding frequency. By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Default: `actual`

Output Arguments

`Spread`

Spread of cash flows over a zero curve.

Examples

Compute Spread Over a Yield Curve for a Cash Flow

Use `cfspread` to compute the spread over a yield curve for a cash flow.

Define data for the yield curve.

```
Settle = datenum('01-Jul-2003');
CurveDates = daysadd(Settle,360*[.25 .5 1 2 3 5 7 10 20],1);
ZeroRates = [.0089 .0096 .0107 .0130 .0166 .0248 .0306 .0356 .0454]';
```

Compute the `RateSpec`.

```
RateSpec = intenvset('StartDates', Settle, 'EndDates', CurveDates,...
'Rates', ZeroRates)

RateSpec = struct with fields:
           FinObj: 'RateSpec'
      Compounding: 2
             Disc: [9x1 double]
            Rates: [9x1 double]
          EndTimes: [9x1 double]
        StartTimes: [9x1 double]
          EndDates: [9x1 double]
        StartDates: 731763
    ValuationDate: 731763
            Basis: 0
     EndMonthRule: 1
```

Compute the spread.

```
Price = 98;
CFAmounts = [30;40;30];
CFDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'});

Spread = cfspread(RateSpec, Price, CFAmounts, CFDates, Settle)

Spread =

   1.0e+03 *

   -8.7956
   -4.0774
   -3.7073
```

Compute Spread Over a Yield Curve for a Cash Flow Using datetime Inputs

Use `cfspread` to compute the spread over a yield curve for a cash flow using `datetime` inputs.

```
Settle = datenum('01-Jul-2003');
CurveDates = daysadd(Settle,360*[.25 .5 1 2 3 5 7 10 20],1);
ZeroRates = [.0089 .0096 .0107 .0130 .0166 .0248 .0306 .0356 .0454]';
```

```
RateSpec = intenvset('StartDates', Settle, 'EndDates', CurveDates,...
'Rates', ZeroRates);
Price = 98;
CFAmounts = [30;40;30];
CFDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'});

CFDates = datetime(CFDates,'ConvertFrom','datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
Spread = cfspread(RateSpec, Price, CFAmounts, CFDates, Settle)

Spread =

   1.0e+03 *

   -8.7956
   -4.0774
   -3.7073
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`cfbyzero` | `cfprice` | `cfyield` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced in R2012a

cfyield

Compute yield to maturity for cash flow given price

Syntax

```
Yield = cfyield(CFAmounts,CFDates,Price,Settle)
Yield = cfyield(CFAmounts,CFDates,Price,Settle,Name,Value)
```

Description

`Yield = cfyield(CFAmounts,CFDates,Price,Settle)` computes yield to maturity for a cash flow given price.

`Yield = cfyield(CFAmounts,CFDates,Price,Settle,Name,Value)` computes yield to maturity for a cash flow given price with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`CFlowAmounts`

`NINST`-by-`MOSTCFS` matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than `MOSTCFS` cash flows, the end of the row is padded with `NaN`s.

`CFlowDates`

`NINST`-by-`MOSTCFS` matrix of cash flow dates, specified as a serial date number, date character vector, or datetime array. Each entry contains the serial date of the corresponding cash flow in `CFlowAmounts`.

`Price`

Price.

Settle

Settlement date, specified as a serial date number, date character vector, or datetime array. Settlement date is the date on which the cash flows are priced.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note Any optional input of size `N`-by-`1` is also acceptable as an array of size `1`-by-`N`, or as a single value applicable to all contracts. Single values are internally expanded to an array of size `N`-by-`1`.

Basis

`N`-by-`1` vector of day-count basis:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Default: `0` (actual/actual)

`CompoundingFrequency`

Compounding frequency. By default, SIA bases (`0`-`7`) and `BUS/252` use a semiannual compounding convention and ICMA bases (`8`-`12`) use an annual compounding convention.

Default: `actual`

Output Arguments

`Yield`

Yield for cash flows.

Examples

Compute the Yield to Maturity for a Cash Flow When Given a Price

Use `cfyield` to compute yield to maturity for a cash flow when given a price.

Define data for the yield curve and price.

```
Settle = datenum('01-Jul-2003');
Price = 98;
CFlowAmounts = [30 40 30];
CFlowDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'})';
```

Compute the `Yield`.

```
 Yield = cfyield(CFlowAmounts, CFlowDates, Price, Settle)
```

```
Yield = 0.0099
```

Compute the Yield to Maturity for a Cash Flow When Given a Price Using datetime Inputs

Use `cfyield` to compute yield to maturity for a cash flow, when given a price using `datetime` inputs.

```
Settle = datenum('01-Jul-2003');
Price = 98;
CFlowAmounts = [30 40 30];
CFlowDates = datenum({'15-Jul-2004', '15-Jul-2005', '15-Jul-2006'})';

CFlowDates = datetime(CFlowDates,'ConvertFrom','datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
Yield = cfyield(CFlowAmounts, CFlowDates, Price, Settle)

Yield = 0.0099
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`cfbyzero` | `cfprice` | `cfspread` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced in R2012a

cftimes

Time factors corresponding to bond cash flow dates

Syntax

```
[TFactors] = cftimes(Settle,Maturity)
[TFactors] =
cftimes(Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstCou
ponDate,LastCouponDate,StartDate)
[TFactors] =
cftimes(Settle,Maturity,'ParameterName',ParameterValue, ...)
```

Description

`[TFactors] = cftimes(Settle,Maturity)` determines the time factors corresponding to the cash flows of a bond or set of bonds.

`cftimes` computes the time factor of a cash flow, which is the difference between the settlement date and the cash flow date, in units of semiannual coupon periods. In computing time factors, use SIA actual/actual day count conventions for all time factor calculations.

`[TFactors] = cftimes(Settle,Maturity,Period,Basis,EndMonthRule,IssueDate,FirstCouponDate,LastCouponDate,StartDate)` determines the time factors corresponding to the cash flows of a bond or set of bonds, including optional inputs.

`[TFactors] = cftimes(Settle,Maturity,'ParameterName',ParameterValue, ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. `'ParameterName'` is the name of the parameter inside single quotes. `ParameterValue` is the value corresponding to `'ParameterName'`. Specify parameter/value pairs in any order. Names are case-insensitive.

Input Arguments

`Settle`

Settlement date. A vector of serial date numbers, date character vectors, or datetime array. `Settle` must be earlier than `Maturity`.

`Maturity`

Maturity date. A vector of serial date numbers, date character vectors, or datetime array.

Ordered Input or Parameter–Value Pairs

Enter the following inputs using an ordered syntax or as parameter/value pairs. You cannot mix ordered syntax with parameter/value pairs.

`Period`

Coupons per year of the bond. A vector of integers. Values are `0`, `1`, `2`, `3`, `4`, `6`, and `12`.

Default: `2`

`Basis`

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Default: `0`

`EndMonthRule`

End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. `0` = ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month. `1` = set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Default: `1`

`IssueDate`

Issue date, specified as a serial date number, date character vector, or datetime array, for a bond.

`FirstCouponDate`

Date, specified as a serial date number, date character vector, or datetime array, when a bond makes its first coupon payment. `FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

Default: If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

`LastCouponDate`

Last coupon date of a bond before the maturity date, specified as a serial date number, date character vector, or datetime array. `LastCouponDate` is used when bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date.

Default: If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

`StartDate`

Date when a bond actually starts (the date from which a bond cash flow is considered), specified as a serial date number, date character vector, or datetime array. To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.

Parameter–Value Pairs

Enter the following inputs only as parameter/value pairs.

`CompoundingFrequency`

Compounding frequency for yield calculation. By default, SIA bases (0–7) and BUS/252 use a semiannual compounding convention and ICMA bases (8–12) use an annual compounding convention.

`DiscountBasis`

Basis used to compute the discount factors for computing the yield. The default behavior is for SIA bases to use the actual/actual day count to compute discount factors. If you use ICMA day counts and BUS/252, the specified bases are used.

Output Arguments

`TFactors`

`TFactors` has `NUMBONDS` rows and the number of columns is determined by the maximum number of cash flow payment dates required to hold the bond portfolio. `NaN`s are padded for bonds which have less than the maximum number of cash flow payment dates.

Examples

Compute the Time Factor of a Cash Flow

This example shows how to calculate a cash flow time factor.

```
Settle = '15-Mar-1997';
Maturity = '01-Sep-1999';
Period = 2;
TFactors = cftimes(Settle, Maturity, Period)

TFactors =

    0.9239    1.9239    2.9239    3.9239    4.9239
```

- “Analyzing and Computing Cash Flows” on page 2-21

References

Krgin, Dragomir. *Handbook of Global Fixed Income Calculations*. John Wiley & Sons, 2002.

Mayle, Jan. *“Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures.”* SIA, Vol 2, Jan 1994.

Stigum, Marcia, and Franklin Robinson. *Money Market and Bond Calculations*. McGraw-Hill, 1996.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `date2time`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

chaikosc

Chaikin oscillator

Syntax

```
chosc = chaikosc(highp,lowp,closep,tvolume)

chosc = chaikosc([highp lowp closep tvolume])

choscts = chaikosc(tsobj)

choscts = chaikosc(tsobj,'ParameterName',ParameterValue, ... )
```

Arguments

`highp`	High price (vector)
`lowp`	Low price (vector)
`closep`	Closing price (vector)
`tvolume`	Volume traded (vector)
`tsobj`	Financial time series object

Description

The Chaikin oscillator is calculated by subtracting the 10-period exponential moving average of the Accumulation/Distribution (A/D) line from the three-period exponential moving average of the A/D line.

`chosc = chaikosc(highp,lowp,closep,tvolume)` calculates the Chaikin oscillator (vector), `chosc`, for the set of stock price and volume traded data (`tvolume`). The required inputs are the prices for the high (`highp`), low (`lowp`), and closing (`closep`) prices and the volume traded data `(tvolume)`.

`chosc = chaikosc([highp lowp closep tvolume])` accepts a four-column matrix as input.

`choscts = chaikosc(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

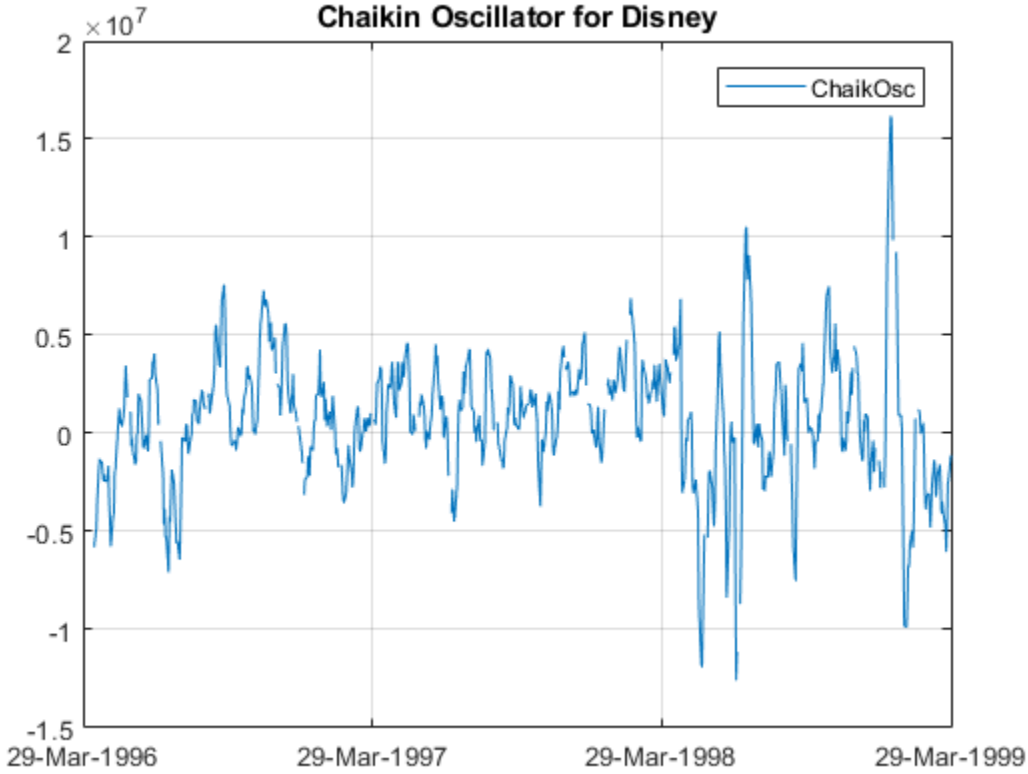
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Chaikin Oscillator

This example shows how to compute the Chaikin oscillator for Disney stock and plot the results.

```
load disney.mat
dis_CHAIKosc = chaikosc(dis);
plot(dis_CHAIKosc)
title('Chaikin Oscillator for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 91–94.

See Also

`adline`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

chaikvolat

Chaikin volatility

Syntax

```
chvol = chaikvolat(highp,lowp)

chvol = chaikvolat([highp lowp])

chvol = chaikvolat(high,lowp,nperdiff,manper)

chvol = chaikvolat([high lowp],nperdiff,manper)

chvts = chaikvolat(tsobj)

chvts = chaikvolat(tsobj,nperdiff,manper,'ParameterName',ParameterValue, ...)
```

Arguments

`highp`	High price (vector).
`lowp`	Low price (vector).
`nperdiff`	Period difference (vector). Default = `10`.
`manper`	Length of exponential moving average in periods (vector). Default = `10`.
`tsobj`	Financial time series object.

Description

`chvol = chaikvolat(highp,lowp)` calculates the Chaikin volatility from the series of stock prices, `highp` and `lowp`. The vector `chvol` contains the Chaikin volatility values, calculated on a 10-period exponential moving average and 10-period difference.

`chvol = chaikvolat([highp lowp])` accepts a two-column matrix as the input.

`chvol = chaikvolat(high,lowp,nperdiff,manper)` manually sets the period difference `nperdiff` and the length of the exponential moving average `manper` in periods.

`chvol = chaikvolat([high lowp],nperdiff,manper)` accepts a two-column matrix as the first input.

`chvts = chaikvolat(tsobj)` calculates the Chaikin volatility from the financial time series object `tsobj`. The object must contain at least two series named `High` and `Low`, representing the high and low prices per period. `chvts` is a financial time series object containing the Chaikin volatility values, based on a 10-period exponential moving average and 10-period difference. `chvts` has the same dates as `tsobj` and a series called `ChaikVol`.

`chvts = chaikvolat (tsobj,nperdiff,manper,'ParameterName', ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name

Parameter values are the character vectors that represent the valid parameter names.

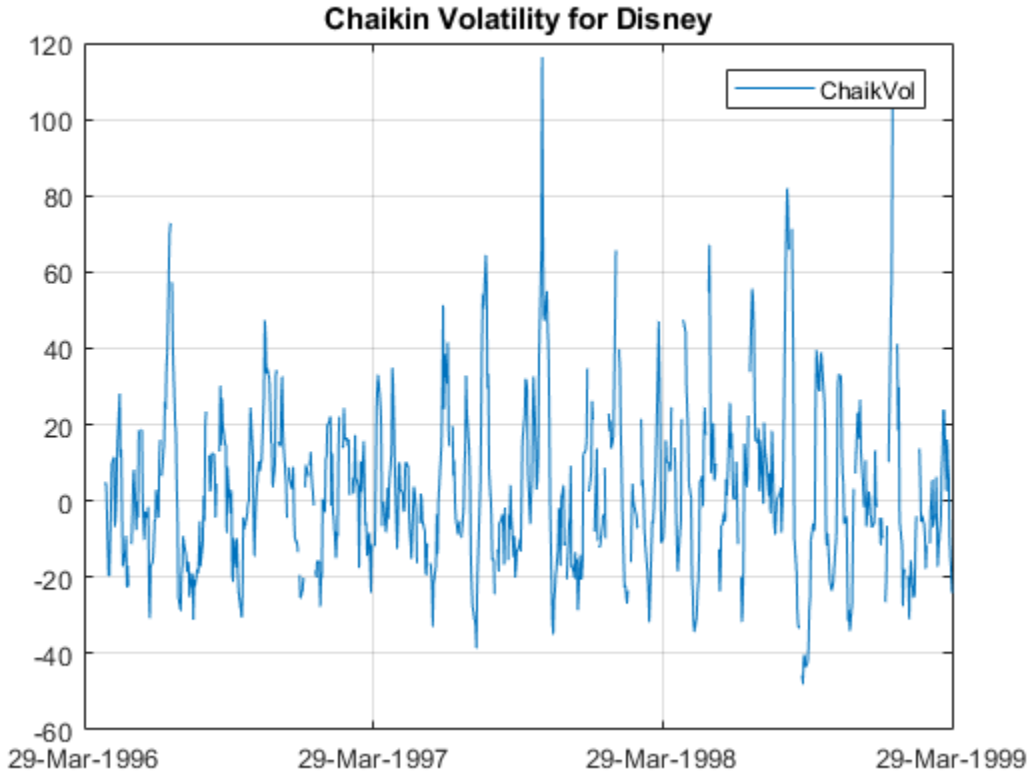
`nperdiff`, the period difference, and `manper`, the length of the exponential moving average in periods, can also be set with this form of `chaikvolat`.

Examples

Compute the Chaikin Volatility

This example shows how to compute the Chaikin volatility for Disney stock and plot the results.

```
load disney.mat
dis_CHAIKvol = chaikvolat(dis);
plot(dis_CHAIKvol)
title('Chaikin Volatility for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 304–305.

See Also

`chaikosc`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

chartfts

Interactive display

Syntax

```
chartfts(tsobj)
```

Description

`chartfts(tsobj)` produces a figure window that contains one or more plots. You can use the mouse to observe the data at a particular time point of the plot.

Examples

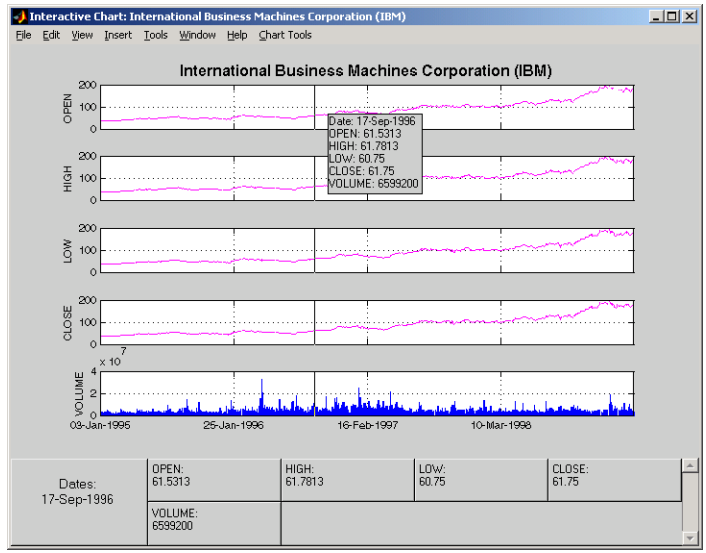
Create a financial time series object from the supplied data file `ibm9599.dat`:

```
ibmfts = ascii2fts('ibm9599.dat', 1, 3, 2);
```

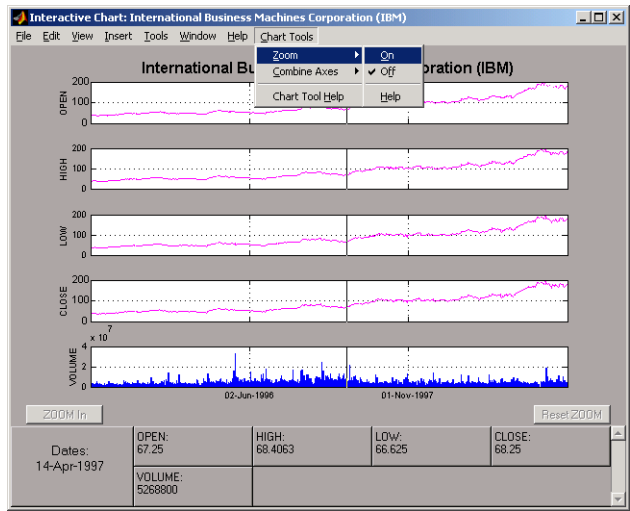
Chart the financial time series object `ibmfts`:

```
chartfts(ibmfts)
```

With the **Zoom** feature set `off`, a mouse click on the indicator line displays object data in a pop-up box.



With the **Zoom** feature set `on`, mouse clicks indicate the area of the chart to zoom.



You can find a tutorial on using `chartfts` in “Visualizing Financial Time Series Objects” on page 11-16. See “Zoom Tool” on page 11-19 for details on performing the zoom. Also see “Combine Axes Tool” on page 11-22 for information about combining axes for specified plots.

See Also

`candle` | `highlow` | `plot`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

checkFeasibility

Check feasibility of input portfolios against portfolio object

Use the `checkFeasibility` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to check the feasibility of input portfolios against a portfolio object.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
status = checkFeasibility(obj,pwgt)
```

Description

`status = checkFeasibility(obj,pwgt)` checks the feasibility of input portfolios against a portfolio object.

Examples

Determine if the Portfolio Is Feasible for a Portfolio Object

Given portfolio `p`, determine if `p` is feasible.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
```

```
pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)

ans = 1x10 logical array
   1   1   1   1   1   1   1   1   1   1
```

Determine if the Portfolio Is Feasible for a PortfolioCVaR Object

Given portfolio `p`, determine if `p` is feasible.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)

ans = 1x10 logical array
   1   1   1   1   1   1   1   1   1   1
```

Determine if the Portfolio Is Feasible for a PortfolioMAD Object

Given portfolio `p`, determine if `p` is feasible.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p);

checkFeasibility(p, pwgt)

ans = 1x10 logical array
   1   1   1   1   1   1   1   1   1   1
```

- "Validate the Portfolio Problem for Portfolio Object" on page 4-104
- "Validate the CVaR Portfolio Problem" on page 5-96
- "Validate the MAD Portfolio Problem" on page 6-91
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`pwgt` — Portfolios to check
matrix

Portfolios to check, specified as a `NumAssets`-by-`NumPorts` matrix.

Data Types: `double`

Output Arguments

`status` — Indicator if portfolio is feasible
row vector

Indicator if portfolio is feasible, returned as a row vector of `NumPorts` indicators that are `true` if portfolio is feasible and `false` otherwise.

Note By definition, any portfolio set must be nonempty and bounded. If the set is empty, no portfolios can be feasible. Use `estimateBounds` to test for nonempty and bounded sets.

Feasibility `status` is returned for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to check the feasibility of input portfolios against a portfolio object.

  ```
  status = obj.checkFeasibility(pwgt);
  ```

- The constraint tolerance to assess whether a constraint is satisfied is obtained from the hidden property `obj.defaultTolCon`.

See Also

`estimateBounds`

Topics

"Validate the Portfolio Problem for Portfolio Object" on page 4-104
"Validate the CVaR Portfolio Problem" on page 5-96
"Validate the MAD Portfolio Problem" on page 6-91
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

chfield

Change data series name

Syntax

```
newfts = chfield(oldfts,oldname,newname)
```

Arguments

`oldfts`	Name of an existing financial time series object.
`oldname`	Name of the existing component in `oldfts`. A MATLAB character vector or column cell array.
`newname`	New name for the component in `oldfts`. A MATLAB character vector 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`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

convert2sur

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` 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 $1 \le$ `NUMGROUPS` $\le$ `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 on the value of `NUMSERIES`:

- If `NUMSERIES = 1`, `DesignSUR = Design`, which is a `NUMSAMPLES`-by-`NUMPARAMS` matrix.
- If `NUMSERIES > 1` and `NUMGROUPS` groups are to be formed, `Design` is a cell array with `NUMSAMPLES` cells, where each cell contains a `NUMSERIES`-by-`(NUMGROUPS * NUMPARAMS)` matrix of known values.

The original collection of parameters that are common to all series are replicated to form collections of parameters for each group.

Examples

Use convert2sur to Estimate Stock Alpha and Beta Values

This example shows a CAPM demonstration using 6 stocks and 60 months of simulated asset returns, where the model for each stock is `AssetReturn = Alpha * 1 + CashReturn + Beta * (MarketReturn - CashReturn) + Noise` and the parameters to estimate are `Alpha` and `Beta`.

Using simulated data, where the `Alpha` estimate(s) are displayed in the first row(s) and the `Beta` estimate(s) are display in the second row(s).

```
Market = (0.1 - 0.04) + 0.17*randn(60, 1);
Asset = (0.1 - 0.04) + 0.35*randn(60, 6);

Design = cell(60, 1);
for i = 1:60
            Design{i} = repmat([ 1, Market(i) ], 6, 1);
end
```

Obtain the aggregate estimates for all stocks.

```
[Param, Covar] = mvnrmle(Asset, Design);

disp({'All 6 Assets Combined'});

    'All 6 Assets Combined'

disp(Param);

    0.0233
    0.1050
```

Estimate parameters for individual stocks using `convert2sur`

```
Group = 1:6;
DesignSUR = convert2sur(Design, Group);
[Param, Covar] = mvnrmle(Asset, DesignSUR);
Param = reshape(Param, 2, 6);

disp({ 'A', 'B', 'C', 'D', 'E', 'F' });

    'A'    'B'    'C'    'D'    'E'    'F'

disp(Param);

    0.0144    0.0270    0.0046    0.0419    0.0376    0.0291
    0.3264   -0.1716    0.3248   -0.0630   -0.0001    0.0637
```

Estimate parameters for pairs of stocks by forming groups.

```
disp({'A & B', 'C & D','E & F'});

    'A & B'    'C & D'    'E & F'

Group = { [1,2 ],[3,4],[5,6]};
DesignSUR = convert2sur(Design, Group);
[Param, Covar] = mvnrmle(Asset, DesignSUR);

Param = reshape(Param, 2, 3);

disp(Param);

    0.0186    0.0190    0.0334
    0.0988    0.1757    0.0293
```

- “Seemingly Unrelated Regression Without Missing Data” on page 9-21

See Also

`ecmnfish` | `mvnrfish`

Topics

"Seemingly Unrelated Regression Without Missing Data" on page 9-21
"Multivariate Normal Linear Regression" on page 9-2

Introduced in R2006a

cir class

Cox-Ingersoll-Ross mean-reverting square root diffusion models

Description

`cir` objects derive from the `sdemrd` (SDE with drift rate expressed in mean-reverting form) class. Use the `cir` constructor to create `cir` objects to simulate sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form. These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time CIR stochastic processes with square root diffusions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\frac{1}{2}})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- S is an `NVARS`-by-`NVARS` matrix of mean reversion speeds (the rate of mean reversion).
- L is an `NVARS`-by-`1` vector of mean reversion levels (long-run mean or level).
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the square root of the corresponding element of the state vector.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Construction

`CIR = cir(Speed,Level,Sigma)` constructs a default `cir` object.

`CIR = cir(Speed,Level,Sigma,Name,Value)` constructs a `cir` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `cir` object, see `cir`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Speed` — `Speed` represents the parameter *S*

array or deterministic function of time or deterministic function of time and state

`Speed` represents the parameter *S*, specified as an array or deterministic function of time.

If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate at which the state vector reverts to its long-run average `Level`).

As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed` as a function of time and state, it calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Level` — `Level` represents the parameter L

array or deterministic function of time or deterministic function of time and state

`Level` represents the parameter L, specified as an array or deterministic function of time.

If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels.

As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1` column vector. If you specify `Level` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of reversion levels when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter V

array or deterministic function of time or deterministic function of time and state

`Sigma` represents the parameter V, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `cir`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$

- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by $(t, X_t$.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t,X_t) = D(t,X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

StartState — Initial values of state variables

1 (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If StartState is a scalar, the gbm constructor applies the same initial value to all state variables on all trials.

If StartState is a column vector, the gbm constructor applies a unique initial value to each state variable on all trials.

If StartState is a matrix, the gbm constructor applies a unique initial value to each state variable on each trial.

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

Simulation — User-defined simulation function or SDE simulation method

if you do not specify a value for Simulation, the default method is simulation by Euler approximation (simByEuler) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

SetAccess	public
GetAccess	public

Data Types: function_handle

Methods

simBySolution	Simulate approximate solution of diagonal-drift HWV processes

Inherited Methods

The following methods are inherited from the`sde` class.

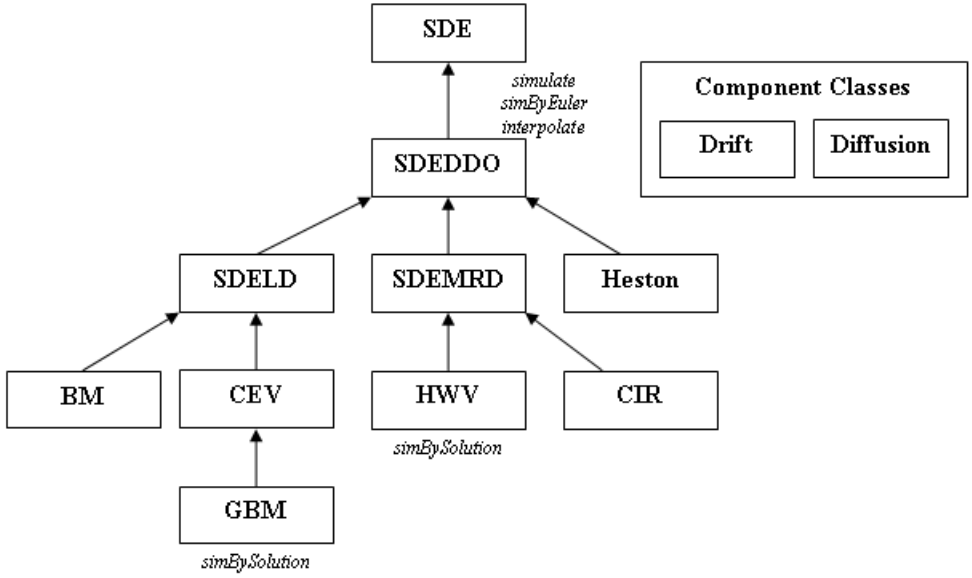
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a cir Object

The Cox-Ingersoll-Ross (CIR) short rate class derives directly from SDE with mean-reverting drift (`SDEMRD`): $dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\frac{1}{2}})V(t)dW$

where D is a diagonal matrix whose elements are the square root of the corresponding element of the state vector.

Create a `cir` object to represent the model: $dX_t = 0.2(0.1 - X_t)dt + 0.05X_t^{\frac{1}{2}}dW$.

```
obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
obj =
   Class CIR: Cox-Ingersoll-Ross
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

Although the last two objects are of different classes, they represent the same mathematical model. They differ in that you create the `cir` object by specifying only three input arguments. This distinction is reinforced by the fact that the `Alpha` parameter does not display – it is defined to be `1/2`.

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Stratified Sampling” on page 17-70
- “Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `cir` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. “Testing Continuous-Time Models of the Spot Interest Rate.” *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeddo` | `simByEuler` | `simulate`

Topics

“Simulating Equity Prices” on page 17-34
“Simulating Interest Rates” on page 17-59
“Stratified Sampling” on page 17-70
“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

cir

Construct Cox-Ingersoll-Ross mean-reverting square root diffusion models

Syntax

```
CIR = cir(Speed, Level, Sigma)

CIR = cir(Speed, Level, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`cir`

Description

This constructor creates and displays `cir` objects, which derive from the `sdemrd` (SDE with drift rate expressed in mean-reverting form) class. Use `cir` objects to simulate sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form. These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time `cir` stochastic processes with square root diffusions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\frac{1}{2}})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- S is an `NVARS`-by-`NVARS` matrix of mean reversion speeds (the rate of mean reversion).
- L is an `NVARS`-by-`1` vector of mean reversion levels (long-run mean or level).

- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the square root of the corresponding element of the state vector.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Speed`	`Speed` represents S. If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate or speed at which the state vector reverts to its long-run average `Level`). As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed` as a function of time and state, it must generate an `NVARS`-by-`NVARS` matrix of reversion rates when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.
`Level`	`Level` represents L. If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels. As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1` matrix. If you specify `Level` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of reversion levels when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.

`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of these input arguments, each argument is specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.

`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `cir` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `cir` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `cir` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`CIR`	Object of class `CIR` with the following displayed parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method • `Speed`: Access function for the input argument `Speed`, callable as a function of time and state • `Level`: Access function for the input argument `Level`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Examples

"Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models" on page 17-29

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t followed by a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `cir` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25

"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

convertto

Convert to specified frequency

Syntax

```
newfts = convertto(oldfts,newfreq)

newfts = convertto(oldfts,newfreq,'param1','value1','param2','value2', ... )
```

Arguments

`oldfts`	Name of an existing financial time series object.

`newfreq`	`1`, `DAILY`, `Daily`, `daily`, `D`, `d`
	
	`2`, `WEEKLY`, `Weekly`, `weekly`, `W`, `w`
	
	`3`, `MONTHLY`, `Monthly`, `monthly`, `M`, `m`
	
	`4`, `QUARTERLY`, `Quarterly`, `quarterly`, `Q`, `q`
	
	`5`, `SEMIANNUAL`, `Semiannual`, `semiannual`, `S`, `s`
	
	`6`, `ANNUAL`, `Annual`, `annual`, `A`, `a`

Description

`convertto` converts a financial time series of any frequency to one of a specified frequency.

`newfts = convertto(oldfts,newfreq)` converts the object `oldfts` to the new time series object `newfts` with the frequency `newfreq`.

Refer to the documentation for each frequency conversion function to determine the valid parameter/value pairs.

See Also

`toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

corrcoef

Correlation coefficients

Syntax

```
r = corrcoef(X)

r = corrcoef(X,Y)
```

Arguments

`X`	Matrix where each row is an observation and each column is a variable.
`Y`	Matrix where each row is an observation and each column is a variable.

Description

`corrcoef` for financial time series objects is based on the MATLAB `corrcoef` function. See `corrcoef`.

`r=corrcoef(X)` calculates a matrix `r` of correlation coefficients for an array `X`, in which each row is an observation, and each column is a variable.

`r=corrcoef(X,Y)`, where `X` and `Y` are column vectors, is the same as `r=corrcoef([X Y])`. `corrcoef` converts `X` and `Y` to column vectors if they are not; that is, `r = corrcoef(X,Y)` is equivalent to `r=corrcoef([X(:) Y(:)])` in that case.

If `c` is the covariance matrix, `c= cov(X)`, then `corrcoef(X)` is the matrix whose (*`i,j`*) 'th element is `c`*`i,j`*/`sqrt`(`c`*`i,i`*`*c`(*`j,j`*)).

`[r,p]=corrcoef(...)` also returns `p`, a matrix of `p`-values for testing the hypothesis of no correlation. Each `p`-value is the probability of getting a correlation as large as the

observed value by random chance, when the true correlation is zero. If `p(i,j)` is less than 0.05, then the correlation `r(i,j)` is significant.

`[r,p,rlo,rup]=corrcoef(...)` also returns matrices `rlo` and `rup`, of the same size as `r`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(...,'PARAM1',VAL1,'PARAM2',VAL2,...)` specifies additional parameters and their values. Valid parameters are:

- `'alpha'` — A number from `0` through `1` to specify a confidence level of 100*(1-ALPHA)%. Default is `0.05` for 95% confidence intervals.
- `'rows'` — Either `'all'` (default) to use all rows, `'complete'` to use rows with no `NaN` values, or `'pairwise'` to compute `r(i,j)` using rows with no `NaN` values in column `i` or `j`.

The `p`-value is computed by transforming the correlation to create a t-statistic having `N` – 2 degrees of freedom, where `N` is the number of rows of `X`. The confidence bounds are based on an asymptotic normal distribution of 0.5*log((1 + r)/(1 – r)), with an approximate variance equal to 1/(N – 3). These bounds are accurate for large samples when `X` has a multivariate normal distribution. The `'pairwise'` option can produce an `r` matrix that is not positive definite.

Examples

Compute Correlation Coefficients

This example shows how to generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4);        % uncorrelated data
x(:,4) = sum(x,2);      % introduce correlation
f = fints((today:today+29)', x);  % create a fints object using x
[r,p] = corrcoef(x);     % compute sample correlation and p-values
[i,j] = find(p<0.05);  % find significant correlations
[i,j]                  % display their (row,col) indices

ans =
```

```
     4     1
     3     2
     2     3
     1     4
```

Class support for inputs X,Y: float: double and single.

- "Using Time Series to Predict Equity Return" on page 12-25

See Also

`cov` | `std` | `var`

Topics

"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

corr2cov

Convert standard deviation and correlation to covariance

Syntax

```
ExpCovariance = corr2cov(ExpSigma,ExpCorrC)
```

Arguments

`ExpSigma`	Vector of length n with the standard deviations of each process. n is the number of random processes.
`ExpCorrC`	(Optional) n-by-n correlation coefficient matrix. If `ExpCorrC` is not specified, the processes are assumed to be uncorrelated, and the identity matrix is used.

Description

`corr2cov` converts standard deviation and correlation to covariance.

`ExpCovariance` is an n-by-n covariance matrix, where n is the number of processes.

```
ExpCov(i,j) = ExpCorrC(i,j)*ExpSigma(i)*ExpSigma(j)
```

Examples

Convert Standard Deviation and Correlation to Covariance

This example shows how to convert standard deviation and correlation to covariance.

```
ExpSigma = [0.5  2.0];

ExpCorrC = [1.0 -0.5
```

```
           -0.5  1.0];

ExpCovariance = corr2cov(ExpSigma, ExpCorrC)

ExpCovariance =

    0.2500   -0.5000
   -0.5000    4.0000
```

- “Data Transformation and Frequency Conversion” on page 12-12

See Also

corrcoef | cov | cov2corr | ewstats | std

Topics

“Data Transformation and Frequency Conversion” on page 12-12

Introduced before R2006a

cov

Covariance matrix

Syntax

```
cov(X)
cov(X,Y)
```

Arguments

X	Financial times series object.
Y	Financial times series object.

Description

cov for financial time series objects is based on the MATLAB cov function. See cov.

If X is a financial time series object with one series, cov(X) returns the variance. For a financial time series object containing multiple series, where each row is an observation, and each series a variable, cov(X) is the covariance matrix.

diag(cov(X)) is a vector of variances for each series and sqrt(diag(cov(X))) is a vector of standard deviations.

cov(X, Y), where X and Y are financial time series objects with the same number of elements, is equivalent to cov([X(:) Y(:)]).

cov(X) or cov(X, Y) normalizes by (N -1) if N > 1, where N is the number of observations. This makes cov(X) the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For N = 1, cov normalizes by N.

cov(X, 1) or cov(X, Y, 1) normalizes by N and produces the second moment matrix of the observations about their mean. cov(X, Y, 0) is the same as cov(X, Y) and

`cov(X, 0)` is the same as `cov(X)`. The mean is removed from each column before calculating the result.

Examples

Create a Covariance Matrix

This example shows how to create a covariance matrix for the following dates.

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007'};
A = [-1 1 2 ; -2 3 1 ; 4 0 3];
f = fints(dates, A);

c = cov(f)

c =

   10.3333   -4.1667    3.0000
   -4.1667    2.3333   -1.5000
    3.0000   -1.5000    1.0000
```

- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`corrcoef` | `cov` | `mean` | `std` | `var`

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

cov2corr

Convert covariance to standard deviation and correlation coefficient

Syntax

```
[ExpSigma,ExpCorrC] = cov2corr(ExpCovariance)
```

Arguments

`ExpCovariance`	*n*-by-*n* covariance matrix; for example, 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

Convert Covariance to Standard Deviations and Correlation Coefficients

This example shows how to convert a covariance matrix to standard deviations and correlation coefficients.

```
ExpCovariance = [0.25 -0.5
                 -0.5   4.0];
```

```
[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)

ExpSigma =

    0.5000    2.0000


ExpCorrC =

    1.0000   -0.5000
   -0.5000    1.0000
```

- “Data Transformation and Frequency Conversion” on page 12-12

See Also

corr2cov | corrcoef | cov | ewstats | std

Topics

“Data Transformation and Frequency Conversion” on page 12-12

Introduced before R2006a

cpncount

Coupon payments remaining until maturity

Syntax

```
NumCouponsRemaining = cpncount(Settle,Maturity)
NumCouponsRemaining = cpncount( ___ ,Period,Basis,EndMonthRule,
IssueDate,FirstCouponDate,LastCouponDate)
```

Description

`NumCouponsRemaining = cpncount(Settle,Maturity)` returns the whole number of coupon payments between the `Settle` and `Maturity` dates for a coupon bond or set of bonds. Coupons falling on or before `Settle` are not counted, except for the `Maturity` payment which is always counted.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`NumCouponsRemaining = cpncount( ___ ,Period,Basis,EndMonthRule, IssueDate,FirstCouponDate,LastCouponDate)` returns the whole number of coupon payments between the `Settle` and `Maturity` dates for a coupon bond or set of bonds using optional input arguments.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Examples

Find Coupon Payments Remaining Until Maturity

This example shows how to find the coupon payments remaining until maturity.

```
NumCouponsRemaining = cpncount('14 Mar 1997', '30 Nov 2000',...
2, 0, 0)

NumCouponsRemaining = 8
```

Find Coupon Payments Remaining Until Maturity for Different Maturity Dates

This example shows how to find the coupon payments remaining until maturity, given three coupon bonds with different maturity dates and the same default arguments.

```
Maturity = ['30 Sep 2000'; '31 Oct 2001'; '30 Nov 2002'];
NumCouponsRemaining = cpncount('14 Sep 1997', Maturity)

NumCouponsRemaining =

     7
     9
    11
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date

serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the bond

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the bond, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as scalar nonnegative integer [0, 1] or a using an N-by-1 vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: logical

IssueDate — Bond issue date
serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: double | char | datetime

FirstCouponDate — Date when bond makes first coupon payment
serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

FirstCouponDate is used when a bond has an irregular first coupon period. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure. If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

Data Types: double | char | datetime

LastCouponDate — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

LastCouponDate is used when a bond has an irregular last coupon period. In the absence of a specified FirstCouponDate, a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate, regardless of where it falls, and is followed only by the bond's

maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumCouponsRemaining` — Whole number of coupon payments between the settlement and maturity dates for a coupon bond or set of bonds
vector

Whole number of coupon payments between the settlement and maturity dates for a coupon bond or set of bonds, returned as an `NBONDS`-by-`1` vector.

Coupons falling on or before settlement are not counted, except for the maturity payment which is always counted.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz` | `datetime`

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Introduced before R2006a

cpndaten

Next coupon date for fixed-income security

Syntax

```
NextCouponDate = cpndaten(Settle,Maturity)
NextCouponDate = cpndaten( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)
```

Description

`NextCouponDate = cpndaten(Settle,Maturity)` returns the next coupon date after the `Settle` date. This function finds the next coupon date whether or not the coupon structure is synchronized with the `Maturity` date.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`NextCouponDate = cpndaten( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate)` returns the next coupon date after the `Settle` date using optional input arguments.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NextCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NextCouponDate` is returned as a datetime array.

Examples

Calculate the Next Coupon Date After the Settlement Date

Determine the `NextCouponDate` when using character vectors for input arguments.

```
NextCouponDate = cpndaten('14-Mar-1997', '30-Nov-2000', 2, 0, 0);
datestr(NextCouponDate)

ans =
'30-May-1997'
```

Determine the `NextCouponDate` when using datetime arrays for input arguments.

```
NextCouponDate = cpndaten('14-Mar-1997', datetime('30-Nov-2000','Locale','en_US'),...
2, 0, 0)

NextCouponDate = datetime
   30-May-1997
```

Determine the `NextCouponDate` when using character vectors for input arguments and the optional argument for `EndMonthRule`.

```
NextCouponDate = cpndaten('14-Mar-1997', '30-Nov-2000', 2, 0, 1);
datestr(NextCouponDate)

ans =
'31-May-1997'
```

Determine the `NextCouponDate` when using an input vector for `Maturity`.

```
Maturity = ['30-Sep-2000'; '31-Oct-2000'; '30-Nov-2000'];
NextCouponDate = cpndaten('14-Mar-1997', Maturity);
datestr(NextCouponDate)

ans = 3x11 char array
    '31-Mar-1997'
    '30-Apr-1997'
    '31-May-1997'
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date

serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the bond

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the bond, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as scalar nonnegative integer [`0`, `1`] or a using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NextCouponDate` — Next coupon date after the settlement date
vector

Next coupon date after the settlement date, returned as an `NUMBONDS`-by-`1` vector of next actual coupon dates after settlement. If settlement is a coupon date, this function never returns the settlement date. Instead, the actual coupon date strictly after settlement is returned, but not exceeding the maturity date. Thus, this function will always return the lesser of the actual maturity date and the next coupon payment date.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NextCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NextCouponDate` is returned as a datetime array.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz` | `datetime`

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Introduced before R2006a

cpndatenq

Next quasi-coupon date for fixed-income security

Syntax

```
NextQuasiCouponDate = cpndatenq(Settle,Maturity)
NextQuasiCouponDate = cpndatenq(___,Period,Basis,EndMonthRule,
IssueDate,FirstCouponDate,LastCouponDate)
```

Description

`NextQuasiCouponDate = cpndatenq(Settle,Maturity)` 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.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`NextQuasiCouponDate = cpndatenq(___,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 using optional input arguments.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NextQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NextQuasiCouponDate` is returned as a datetime array.

Examples

Determine the Next Quasi Coupon Date for a Portfolio of Fixed-Income Securities

Given a pair of bonds with the following characteristics:

```
Settle = char('30-May-1997','10-Dec-1997');
Maturity = char('30-Nov-2002','10-Jun-2004');
```

Compute `NextCouponDate` for this pair of bonds.

```
NextCouponDate = cpndaten(Settle, Maturity);
datestr(NextCouponDate)
```

```
ans = 2x11 char array
    '31-May-1997'
    '10-Jun-1998'
```

Compute the next quasi coupon dates for these two bonds.

```
NextQuasiCouponDate = cpndateq(Settle, Maturity);
datestr(NextQuasiCouponDate)
```

```
ans = 2x11 char array
    '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 = 2x11 char array
    '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 = 2x11 char array
    '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.

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date
serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond
`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NextQuasiCouponDate` — Next quasi coupon date for portfolio of `NUMBONDS` fixed income securities

vector

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, returned as a `NUMBONDS`-by-`1` vector.

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.

If all of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NextQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NextQuasiCouponDate` is returned as a datetime array.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz` | `datetime`

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Introduced before R2006a

cpndatepq

Previous quasi-coupon date for fixed-income security

Syntax

```
PreviousQuasiCouponDate = cpndatepq(Settle,Maturity)
PreviousQuasiCouponDate = cpndatepq(___,Period,Basis,EndMonthRule,
IssueDate,FirstCouponDate,LastCouponDate)
```

Description

`PreviousQuasiCouponDate = cpndatepq(Settle,Maturity)` determines the previous quasi-coupon date for a set of `NUMBONDS` fixed income securities. Prior 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. This function finds the previous quasi-coupon date for bonds with a coupon structure whose first or last period is either normal, short, or long.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`PreviousQuasiCouponDate = cpndatepq(___,Period,Basis,EndMonthRule, IssueDate,FirstCouponDate,LastCouponDate)`, using optional input arguments, determines the previous quasi-coupon date for a set of `NUMBONDS` fixed income securities.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `PreviousQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `PreviousQuasiCouponDate` is returned as a datetime array.

Examples

Determine the Previous Quasi Coupon Date for a Portfolio of Fixed-Income Securities

Given a pair of bonds with the following characteristics:

```
Settle = char('30-May-1997','10-Dec-1997');
Maturity = char('30-Nov-2002','10-Jun-2004');
```

With no `FirstCouponDate` explicitly supplied, compute the `PreviousCouponDate` for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity);
datestr(PreviousCouponDate)
```

```
ans = 2x11 char array
    '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 = 2x11 char array
    '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 = 2x11 char array
    '30-Nov-1996'
    '10-Dec-1997'
```

For the first bond the settlement date is not a normal coupon date. The `PreviousQuasiCouponDate` is the coupon date before or on the settlement date. Since the coupon structure is synchronized to `FirstCouponDate`, the previous quasi coupon date is `30-Nov-1996`. `PreviousQuasiCouponDate` disregards `IssueDate` and `FirstCouponDate` in this case. For the second bond the settlement date (10-Dec-1997) occurs on a date when a coupon would normally be paid in the absence of an explicit `FirstCouponDate`. `cpndatepq` returns this date as `PreviousQuasiCouponDate`.

- "Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date

serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`PreviousQuasiCouponDate` — Previous quasi coupon date for portfolio of `NUMBONDS` fixed income securities

vector

Previous quasi coupon date for a portfolio of `NUMBONDS` fixed income securities, whether or not the first or last coupon is normal, short, or long, returned as a `NUMBONDS`-by-`1` vector of previous quasi-coupon dates before settlement. If settlement is a coupon date, this function returns the settlement date.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `PreviousQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `PreviousQuasiCouponDate` is returned as a datetime array.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndaysn` | `cpndaysp` | `cpnpersz` | `datetime`

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Introduced before R2006a

cpndatep

Previous coupon date for fixed-income security

Syntax

```
PreviousCouponDate = cpndatep(Settle,Maturity)
PreviousCouponDate = cpndatep(___,Period,Basis,EndMonthRule,
IssueDate,FirstCouponDate,LastCouponDate)
```

Description

`PreviousCouponDate = cpndatep(Settle,Maturity)` 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.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`PreviousCouponDate = cpndatep(___,Period,Basis,EndMonthRule,
IssueDate,FirstCouponDate,LastCouponDate)` returns the previous coupon date on or before settlement for a portfolio of bonds.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `PreviousCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `PreviousCouponDate` is returned as a datetime array.

Examples

Calculate the Previous Coupon Date on or Before Settlement

Determine the `PreviousCouponDate` when using character vectors for input arguments.

```
PreviousCouponDate = cpndatep('14-Mar-1997', '30-Jun-2000',...
2, 0, 0);
datestr(PreviousCouponDate)
```

```
ans =
'30-Dec-1996'
```

Determine the `PreviousCouponDate` when using datetime arrays for input arguments.

```
PreviousCouponDate = cpndatep(datetime('14-Mar-1997','Locale','en_US'), '30-Jun-2000',.
2, 0, 0)
```

```
PreviousCouponDate = datetime
   30-Dec-1996
```

Determine the `PreviousCouponDate` when using character vectors for input arguments and the optional argument for `EndMonthRule`.

```
PreviousCouponDate = cpndatep('14-Mar-1997', '30-Jun-2000',...
2, 0, 1);
datestr(PreviousCouponDate)
```

```
ans =
'31-Dec-1996'
```

Determine the `PreviousCouponDate` when using an input vector for `Maturity`.

```
Maturity = ['30-Apr-2000'; '31-May-2000'; '30-Jun-2000'];
PreviousCouponDate = cpndatep('14-Mar-1997', Maturity);
datestr(PreviousCouponDate)
```

```
ans = 3x11 char array
    '31-Oct-1996'
    '30-Nov-1996'
    '31-Dec-1996'
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date

serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond

`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date
serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`PreviousCouponDate` — Previous coupon date on or before settlement for portfolio of bonds
vector

Previous coupon date on or before settlement for portfolio of bonds, returned as an `NUMBONDS`-by-`1` vector. If settlement is a coupon date, this function returns the settlement date. The actual coupon date strictly on or before settlement is returned, but not exceeding the issue date, if available. Thus, this function always returns the lesser of the actual issue date and the previous coupon payment date with respect to the settlement date.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then

`PreviousCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `PreviousCouponDate` is returned as a datetime array.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `cpnpersz` | `datetime`

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Introduced before R2006a

cpndaysn

Number of days to next coupon date

Syntax

```
NumDaysNext = cpndaysn(Settle,Maturity)
NumDaysNext = cpndaysn(___,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)
```

Description

`NumDaysNext = cpndaysn(Settle,Maturity)` 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. `NumDaysNext` returns a double for serial date number, date character vector, and datetime inputs.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`NumDaysNext = cpndaysn(___,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)` returns the number of days from the settlement date to the next coupon date for a bond or set of bonds using optional input arguments.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NumDaysNext` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NumDaysNext` is returned as a datetime array.

Examples

Calculate the Number of Days From Settlement Date to Next Coupon Date

Determine the `NumDaysNext` when using character vectors for input arguments.

```
NumDaysNext = cpndaysn('14-Sep-2000', '30-Jun-2001', 2, 0, 0)
```

```
NumDaysNext = 107
```

Determine the `NumDaysNext` when using datetime arrays for input arguments.

```
NumDaysNext = cpndaysn(datetime('14-Sep-2000','Locale','en_US'), '30-Jun-2001', 2, 0, 0
```

```
NumDaysNext = 107
```

Determine the `NumDaysNext` when using character vectors for input arguments and the optional argument for `EndMonthRule`.

```
NumDaysNext = cpndaysn('14-Sep-2000', '30-Jun-2001', 2, 0, 1)
```

```
NumDaysNext = 108
```

Determine the `NumDaysNext` when using an input vector for `Maturity`.

```
Maturity = ['30-Apr-2001'; '31-May-2001'; '30-Jun-2001'];
NumDaysNext = cpndaysn('14-Sep-2000', Maturity)
```

```
NumDaysNext =

    47
    77
   108
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date
serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond
`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date

serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment

serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDaysNext` — Number of days from settlement date to next coupon date
vector

Number of days from settlement date to next coupon date, returned as an `NUMBONDS`-by-`1` vector. For zero coupon bonds coupon dates are computed as if the bonds have a semiannual coupon structure. `NumDaysNext` returns a double for serial date number, date character vector, and datetime inputs.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NumDaysNext` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NumDaysNext` is returned as a datetime array.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatenq | cpndatep | cpndatepq | cpndaysn | cpndaysp | cpnpersz | datetime

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Introduced before R2006a

cpndaysp

Number of days since previous coupon date

Syntax

```
NumDaysPrevious = cpndaysp(Settle,Maturity)
NumDaysPrevious = cpndaysp( ___ ,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)
```

Description

`NumDaysPrevious = cpndaysp(Settle,Maturity)` 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. `NumDaysPrevious` returns a double for serial date number, date character vector, and datetime inputs.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`NumDaysPrevious = cpndaysp( ___ ,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate)` returns the number of days between the previous coupon date and the settlement date for a bond or set of bonds using optional input arguments.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NumDaysPrevious` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NumDaysPrevious` is returned as a datetime array.

Examples

Calculate the Number of Days Between Previous Coupon Date and Settlement Date

Determine the `NumDaysPrevious` when using character vectors for input arguments.

```
NumDaysPrevious = cpndaysp('14-Mar-2000', '30-Jun-2001', 2, 0, 0)
```

```
NumDaysPrevious = 75
```

Determine the `NumDaysPrevious` when using a datetime array for an input argument.

```
NumDaysPrevious = cpndaysp(datetime('14-Mar-2000','Locale','en_US'), '30-Jun-2001', 2,
```

```
NumDaysPrevious = 75
```

Determine the `NumDaysPrevious` when using character vectors for input arguments and the optional argument for `EndMonthRule`.

```
NumDaysPrevious = cpndaysp('14-Mar-2000', '30-Jun-2001', 2, 0, 1)
```

```
NumDaysPrevious = 74
```

Determine the `NumDaysPrevious` when using an input vector for `Maturity`.

```
Maturity = ['30-Apr-2001'; '31-May-2001'; '30-Jun-2001'];
NumDaysPrevious = cpndaysp('14-Mar-2000', Maturity)
```

```
NumDaysPrevious =

   135
   105
    74
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date
serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond
`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a with value `0` through `13` or an `N`-by-`1` vector of integers with values `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date

serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment

serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDaysPrevious` — Number of days between previous coupon date and settlement date
vector

Number of days between the previous coupon date and the settlement date, returned as an `NUMBONDS`-by-`1` vector. If the settlement date is a coupon date, this function always returns the settlement date.

When the coupon frequency is 0 (a zero coupon bond), the previous coupon date is calculated as if the frequency were semiannual. `NumDaysPrevious` returns a double for serial date number, date character vector, and datetime inputs.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NumDaysPrevious` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NumDaysPrevious` is returned as a datetime array.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatenq | cpndatep | cpndatepq | cpndaysn | cpndaysp | cpnpersz | datetime

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Introduced before R2006a

cpnpersz

Number of days in coupon period

Syntax

```
NumDaysPeriod = cpnpersz(Settle,Maturity)
NumDaysPeriod = cpnpersz(___,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)
```

Description

`NumDaysPeriod = cpnpersz(Settle,Maturity)` 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. `NumDaysPeriod` returns a double for serial date number, date character vector, and datetime inputs.

Required input arguments must be number of bonds, `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS`, conforming vectors or scalars.

`NumDaysPeriod = cpnpersz(___,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate)` returns the number of days in the coupon period containing the settlement date using optional input arguments.

Optional input arguments must be either `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NumDaysPeriod` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NumDaysPeriod` is returned as a datetime array.

Examples

Calculate the Number of Days in the Coupon Period Containing Settlement Date

Determine the `NumDaysPeriod` when using character vectors for input arguments.

```
NumDaysPeriod = cpnpersz('14-Sep-2000', '30-Jun-2001', 2, 0, 0)
```

```
NumDaysPeriod = 183
```

Determine the `NumDaysPeriod` when using a datetime array for an input argument.

```
NumDaysPeriod = cpnpersz(datetime('14-Sep-2000','Locale','en_US'), '30-Jun-2001', 2, 0,
```

```
NumDaysPeriod = 183
```

Determine the `NumDaysPeriod` when using character vectors for input arguments and the optional argument for `EndMonthRule`.

```
NumDaysPeriod = cpnpersz('14-Sep-2000', '30-Jun-2001', 2, 0, 1)
```

```
NumDaysPeriod = 184
```

Determine the `NumDaysPeriod` when using an input vector for `Maturity`.

```
Maturity = ['30-Apr-2001'; '31-May-2001'; '30-Jun-2001'];
NumDaysPeriod = cpnpersz('14-Sep-2000', Maturity)
```

```
NumDaysPeriod =

   184
   183
   184
```

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-25

Input Arguments

`Settle` — Settlement date

serial date numbers | date character vector | datetime object

Settlement date, specified as a vector of serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date
serial date number | date character vector | datetime array

Maturity date, specified as a vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Period` — Coupons per year of the bond
`2` (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `single` | `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or an `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a nonnegative integer [`0`, `1`] using an `N`-by-`1` vector of values. 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, meaning that a bond's coupon payment date is always the last actual day of the month.

Data Types: `logical`

`IssueDate` — Bond issue date

serial date number | date character vector | datetime array

Bond issue date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment

serial date number | date character vector | datetime array

Date when a bond makes its first coupon payment, specified as a serial date number, date character vector, or datetime array.

`FirstCouponDate` is used when a bond has an irregular first coupon period. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure. If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
serial date number | date character vector | datetime array

Last coupon date of a bond before maturity date, specified as a serial date number, date character vector, or datetime array.

`LastCouponDate` is used when a bond has an irregular last coupon period. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate`, regardless of where it falls, and is followed only by the bond's maturity cash flow date. If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDaysPeriod` — Number of days in coupon period containing settlement date
vector

Number of days in the coupon period containing the settlement date, returned as an `NUMBONDS`-by-`1` vector. For zero coupon bonds, coupon dates are computed as if the bonds have a semiannual coupon structure.

If all the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are either serial date numbers or date character vectors, then `NumDaysPeriod` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date character vector.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, and `LastCouponDate` are datetime arrays, then `NumDaysPeriod` is returned as a datetime array.

See Also

`accrfrac` | `cfamounts` | `cfdates` | `cftimes` | `cpncount` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndatepq` | `cpndaysn` | `cpndaysp` | `datetime`

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-25

Introduced before R2006a

createholidays

Create trading calendars

Syntax

```
createholidays(Filename,Codefile,InfoFile,TargetDir,IncludeWkds,
Wprompt,NoGUI)
```

Description

`createholidays(Filename,Codefile,InfoFile,TargetDir,IncludeWkds, Wprompt,NoGUI)` programmatically generates the market-specific `holidays.m` files (from `FinancialCalendar.com` financial center holiday data) without displaying the interface.

Note To use `createholidays`, you must obtain data, codes, and info files from `http://www.FinancialCalendar.com` trading calendars. The data files must be in the required MATLAB format.

Examples

Create `holidays.m` Files Using `createholidays`

Use `createholidays` to create `holidays*.m` files from `My_datafile.csv` in the folder `c:\work`. Weekends are included in the holidays list based on the input flag `INCLUDEWDKS = 1`

```
createholidays('FinancialCalendar\My_datafile.csv',...
'FinancialCalendar\My_codesfile.csv',...
'FinancialCalendar\My_infofile.csv','c:\work',1,1,1)
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Filename` — Data file name

character vector

Data file name, specified using a character vector.

Data Types: `char`

`Codefile` — Code file name

character vector

Code file name, specified using a character vector.

Data Types: `char`

`InfoFile` — Info file name

character vector

Info file name, specified using a character vector.

Data Types: `char`

`TargetDir` — Target folder where to write new `holidays.m` files

character vector

Target folder where to write the new `holidays.m` files, specified using a character vector.

Data Types: `char`

`IncludeWkds` — Option to include weekends in holiday list

numeric with value `0` or `1`

Option to include weekends in the holiday list, specified using a numeric with value `0` or `1`. Values are:

- `0` – Do not include weekends in the holiday list.
- `1` – Include weekends in the holiday list.

Data Types: `logical`

`Wprompt` — Option to prompt for file location for each `holiday.m` file that is created
numeric with value `0` or `1`

Option to prompt for the file location for each `holiday.m` file that is created, specified using a numeric with value `0` or `1`. Values are:

- `0` – Do not prompt for the file location.
- `1` – Prompt for the file location.

Data Types: `logical`

`NoGUI` — Run `createholidays` without displaying Trading Calendars user interface
numeric with value `0` or `1`

Run `createholidays` without displaying the Trading Calendars user interface, specified using a numeric with value `0` or `1`. Values are:

- `0` – Display the GUI.
- `1` – Do not display the GUI.

Data Types: `logical`

See Also

`holidays`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced in R2007b

cumsum

Cumulative sum

Syntax

```
newfts = cumsum(oldfts)
```

Description

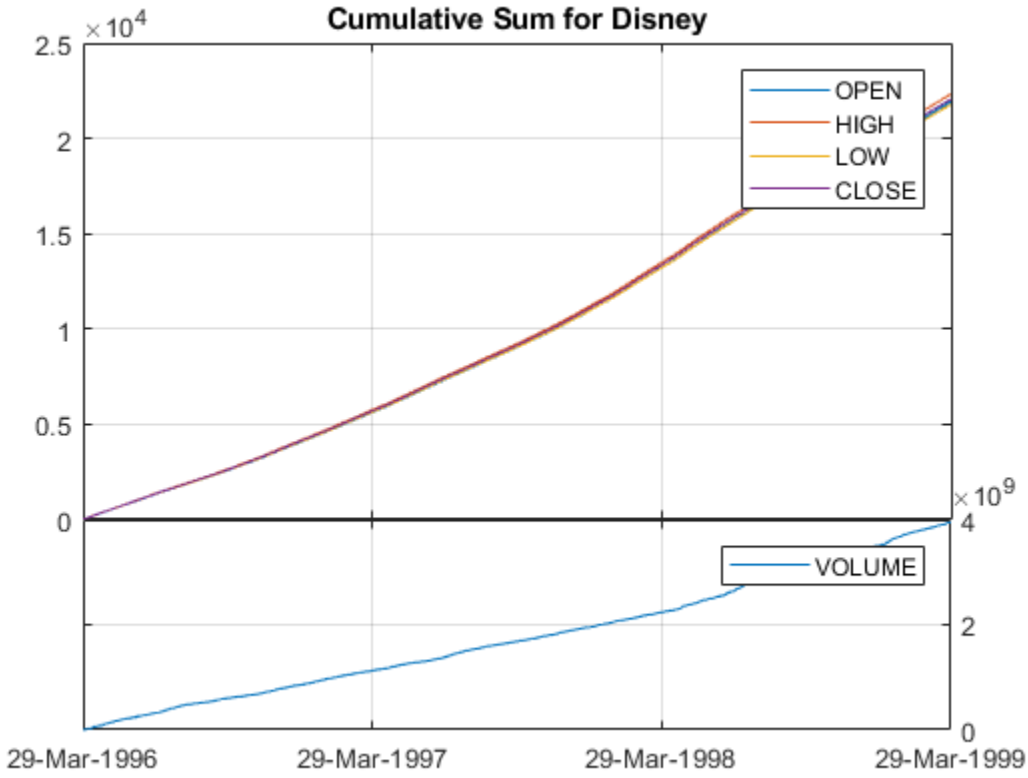
`newfts = cumsum(oldfts)` calculates the cumulative sum of each individual time series data series in the financial time series object `oldfts` and returns the result in another financial time series object `newfts`. `newfts` contains the same data series names as `oldfts`.

Examples

Compute the Cumulative Sum

This example shows how to compute the cumulative sum for Disney stock and plot the results.

```
load disney.mat
cs_dis = cumsum(fillts(dis));
plot(cs_dis)
title('Cumulative Sum for Disney')
```



- “Financial Time Series Operations” on page 12-8
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`fints`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

cur2frac

Decimal currency values to fractional values

Syntax

```
Fraction = cur2frac(Decimal,Denominator)
```

Description

`Fraction = cur2frac(Decimal,Denominator)` converts decimal currency values to fractional values. `Fraction` is returned as a character vector.

Examples

Convert Decimal Currency Values to Fractional Values

This example shows how to convert decimal currency values to fractional values.

```
Fraction = cur2frac(12.125, 8)
```

```
Fraction =
'12.1'
```

Input Arguments

`Decimal` — Decimal currency value

numeric decimal

Decimal currency value, specified as a scalar or vector using numeric decimal values.

Data Types: `double`

`Denominator` — Denominator of the fractions
numeric

Denominator of the fractions, specified as a scalar or vector using numeric values for the denominator.

Data Types: `double`

Output Arguments

`Fraction` — Fractional values
character vector | cell array of character vectors

Fractional values, returned as a character vector or cell array of character vectors.

See Also

`cur2str` | `frac2cur`

Introduced before R2006a

cur2str

Bank-formatted text

Syntax

```
BankText = cur2str(Value,Digits)
```

Description

`BankText = cur2str(Value,Digits)` returns the given value in bank format.

The output format for `BankText` is a numerical format with dollar sign prefix, two decimal places, and negative numbers in parentheses; for example, ($123.45) and $6789.01. The standard MATLAB bank format uses two decimal places, no dollar sign, and a minus sign for negative numbers; for example, −123.45 and 6789.01.

Examples

Return Bank Formatted Text

Return bank formatted text for a negative numeric value.

```
BankText = cur2str(-826444.4456,3)
```

```
BankText =
'($826444.446)'

% Negative numbers are displayed in parentheses.
```

Input Arguments

`Value` — Value to be formatted

numeric value

Value to be formatted, specified as a numeric value.

Data Types: `double`

`Digits` — Number of significant digits

`2` (default) | integer

Number of significant digits, specified as an integer. A negative `Digits` rounds the value to the left of the decimal point.

Data Types: `double`

Output Arguments

`BankText` — Bank-formatted text

character vector

Bank-formatted text (`BankText`) is returned as a character vector with a leading dollar sign (`$`). Negative numbers are displayed in parentheses.

See Also

`cur2frac` | `frac2cur`

Introduced before R2006a

date2time

Time and frequency from dates

Syntax

```
[TFactors,F] = date2time(Settle,Maturity)
[TFactors,F] = date2time(___,Compounding,Basis,EndMonthRule)
```

Description

`[TFactors,F] = date2time(Settle,Maturity)` computes time factors appropriate to compounded rate quotes between the `Settle` and `Maturity` dates. `date2time` is the inverse of `time2date`.

`[TFactors,F] = date2time(___,Compounding,Basis,EndMonthRule)` computes time factors appropriate to compounded rate quotes between the `Settle` and `Maturity` dates using optional input arguments for `Compounding`, `Basis`, and `EndMonthRule`. `date2time` is the inverse of `time2date`.

Examples

Compute `date2time` Using an actual/actual Basis

To get the `date2time` period between '31-Jul-2015' and '30-Sep-2015' using an actual/actual basis:

```
date2time('31-Jul-2015', '30-Sep-2015', 2, 0, 1)
```

```
ans =

    0.3333
```

When using `date2time` quasi coupon, two quasi coupon dates are computed for a bond with a maturity corresponding to the `Dates` input. In this case, that would be "30-Sep-2015". Assuming that the compounding frequency is `2`, the other quasi coupon date is six months prior to this date. Assuming the end of month rule is in place, then the other quasi coupon date is "31-Mar-2015". You can use these two dates to compute the total number of actual days in a period (which is 183). Given this, the fraction of time between the start and end date for the actual/actual basis is computed as follows.

```
(Actual Days between Start Date and End Date)/(Actual Number of Days
between Quasi Coupon Dates)
```

There are 61 days between 31-Jul-2015 and 30-Sep-2015 and 183 days between the quasi coupon dates ("31-Mar-2015" and "30-Sep-2015") which leads to a final result of 61/183 or exactly 1/3.

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Settle` — Settlement date

serial date number | date character vector | datetime object

Settlement date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date

serial date number | date character vector

Maturity date, specified as a scalar or `N`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Compounding` — Rate at which input zero rates are compounded when annualized

`2` (Semiannual compounding) (default) | scalar with numeric values of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `12`, `365`, `-1`

Rate at which input zero rates are compounded when annualized, specified as a scalar with numeric values of: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `12`, `365`, or `-1`. Allowed values are defined as:

- `0` — Simple interest (no compounding)
- `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

The optional `Compounding` argument determines the formula for the discount factors (`Disc`):

- `Compounding` = `0` for simple interest
  - `Disc = 1/(1 + Z * T)`, where `T` is time in years and simple interest assumes annual times `F = 1`.
- `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, for example, `T = F` is one year.
- `Compounding` = `365`
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.
- `Compounding` = `-1`
  - `Disc = exp(-T*Z)`, where `T` is time in years.

`Basis` — Day-count basis

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis, specified as an integer with a value `0` through `13` or a `N`-by-`1` vector of integers with values `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as scalar nonnegative integer [`0`, `1`] or a using a `N`-by-`1` vector of values. 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 payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`TFactors` — Time factors

vector

Time factors, appropriate to compounded rate quotes between `Settle` and `Maturity` dates, returned as a vector.

`F` — Compounding frequencies

scalar

Compounding frequencies, returned as a scalar.

Definitions

Difference Between `yearfrac` and `date2time`

The difference between `yearfrac` and `date2time` is that `date2time` counts full periods as a whole integer, even if the number of actual days in the periods are different. `yearfrac` does not count full periods.

For example,

```
yearfrac('1/1/2000', '1/1/2001', 9)

ans =

    1.0167
```

`yearfrac` for `Basis 9` (ACT/360 ICMA) calculates 366/360 = 1.0167. So, even if the dates have the same month and date, with a difference of 1 in the year, the returned value may not be exactly 1. On the other hand, `date2time` calculates one full year period:

```
date2time('1/1/2000', '1/1/2001', 1, 9)

ans =

     1
```

See Also

`cftimes` | `datetime` | `disc2rate` | `rate2disc` | `time2date` | `yearfrac`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

dateaxis

Convert serial-date axis labels to calendar-date axis labels

Syntax

```
dateaxis(Tickaxis,DateForm,StartDate)
```

Arguments

`Tickaxis`	(Optional) Determines which axis tick labels—*x*, *y*, or *z*—to replace. Enter as a character vector. 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` on page 18-550 format descriptions.
`StartDate`	(Optional) Assigns the date to the first axis tick value, specified as a serial date number, date character vector, or datetime array. The tick values are treated as serial date numbers. The default `StartDate` is the lower axis limit converted to the appropriate date number. For example, a tick value of `1` is converted to the date 01-Jan-0000. Entering `StartDate` as `'06-apr-1999'` assigns the date April 6, 1999 to the first tick value and the axis tick labels are set accordingly.

Description

`dateaxis(Tickaxis,DateForm,StartDate)` replaces axis tick labels with date labels on a graphic figure.

See the MATLAB `set` command for information on modifying the axis tick values and other axis parameters.

DateForm	Format	Description
0	01-Mar-1999 15:45:17	day-month-year hour:minute:second
1	01-mar-1999	day-month-year
2	03/01/99	month/day/year
3	Mar	month, three letters
4	M	month, single letter
5	3	month
6	03/01	month/day
7	1	day of month
8	Wed	day of week, three letters
9	W	day of week, single letter
10	1999	year, four digits
11	99	year, two digits
12	Mar99	month year
13	15:45:17	hour:minute:second
14	03:45:17 PM	hour:minute:second AM or PM
15	15:45	hour:minute
16	03:45 PM	hour:minute AM or PM
17	95/03/01	year month day

Examples

```
dateaxis('x') or dateaxis
```

converts the *x*-axis labels to an automatically determined date format.

```
dateaxis('y', 6)
```

converts the *y*-axis labels to the month/day format.

```
dateaxis('x', 2, '03-Mar-1999')
```

or

```
dateaxis('x', 2, datetime('03-Mar-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` | `datetime` | `highlow` | `movavg` | `pointfig`

Topics

"Charting Financial Data" on page 2-14

Introduced before R2006a

datedisp

Display date entries

Syntax

```
datedisp(NumMat)
datedisp(___,DateForm)
CharMat = datedisp(NumMat,DateForm)
```

Description

`datedisp(NumMat)` displays a matrix with the serial dates formatted as date character vectors, 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.

`datedisp(___,DateForm)`, using the optional argument `DateForm`, displays a matrix with the serial dates formatted as date character vectors, 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 = datedisp(NumMat,DateForm)` returns `CharMat`, character array representing `NumMat`. If no output variable is assigned, the function prints the array to the display.

Examples

Display a Matrix with the Serial Dates Formatted as Date Character Vectors

This example shows how to display a matrix with the serial dates formatted as date character vectors.

```
NumMat = [730730, 0.03, 1200 730100;
          730731, 0.05, 1000 NaN];

datedisp(NumMat)

01-Sep-2000   0.03   1200   11-Dec-1998
02-Sep-2000   0.05   1000      NaN
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`NumMat` — Numeric matrix to display

serial date numbers

Numeric matrix to display, specified as a serial date numbers.

Data Types: `double`

`DateForm` — Date format

scalar character vector to indicate format of text representing dates

Date format, specified as a scalar character vector to indicate the format of text representing dates. See `datestr` for available and default format flags.

Data Types: `char`

Output Arguments

`CharMat` — Character array representing `NumMat`

array of date character vectors

Character array representing `NumMat`, returned as an array of date character vectors. If no output variable is assigned, the function prints the array to the display.

See Also

`datenum` | `datestr`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

datefind

Indices of dates in matrix

Syntax

```
Indices = datefind(Subset,Superset)
Indices = datefind(___,Tolerance)
```

Description

`Indices = datefind(Subset,Superset)` returns a vector of indices to the date numbers in `Superset` that are present in `Subset`. If no date numbers match, `Indices = []`.

`Indices = datefind(___,Tolerance)` returns a vector of indices to the date numbers in `Superset` that are present in `Subset`, plus the optional argument for `Tolerance`. If no date numbers match, `Indices = []`.

Examples

Return a Vector of Indices to Date Numbers

This example shows how to return a vector of indices to date numbers.

```
Superset = datenum(1999, 7, 1:31);
Subset = [datenum(1999, 7, 10); datenum(1999, 7, 20)];
Indices = datefind(Subset, Superset, 1)
```

```
Indices =

     9
    10
    11
    19
```

```
20
21
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Subset` — Subset of dates to find matching dates

matrix of nonnegative integers with values for serial date numbers or datetime arrays

Subset of dates to find matching dates in `Superset`, specified as a matrix of nonnegative integers with values for serial date numbers or datetime arrays.

`Subset` and `Superset` can be either serial date numbers or datetime arrays. These types do not have to match. `datefind` determines the underlying date to match dates of different data types.

Note The elements of `Subset` must be contained in `Superset`, without repetition. `datefind` works with non-repeating sequences of dates.

Example: `Subset = [datenum(1997,7,10); datenum(1997,7,20)];`

Data Types: `single` | `double`

`Superset` — Superset of dates

matrix of nonnegative integers with values for serial date numbers or datetime arrays

Superset of dates, specified as a matrix of serial date numbers or datetime arrays, whose elements are sought.

`Subset` and `Superset` can be either serial date numbers or datetime arrays. These types do not have to match. `datefind` determines the underlying date to match dates of different data types.

Note The elements of `Subset` must be contained in `Superset`, without repetition. `datefind` works with non-repeating sequences of dates.

Example: `Superset = datenum(1997,7,1:31);`

Data Types: `single` | `double`

`Tolerance` — Tolerance for matching dates in `Superset`

`0` (default) | positive integer or duration object

Tolerance for matching dates (+/-) in `Superset`, specified as a positive integer or duration object.

Data Types: `single` | `double`

Output Arguments

`Indices` — Indices of dates in `Superset` that are present in `Subset`

vector

Indices of dates in `Superset` that are present in `Subset` (plus or minus the tolerance if defined using the optional argument `Tolerance`), returned as a vector of indices to the date numbers or datetimes.

See Also

`datenum` | `datetime`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

datemnth

Date of day in future or past month

Syntax

```
TargetDate = datemnth(StartDate,NumberMonths)
TargetDate = datemnth(___,DayFlag,Basis,EndMonthRule)
```

Description

`TargetDate = datemnth(StartDate,NumberMonths)` determines a date in a future or past month based on movement either forward or backward in time by a given number of months.

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 character vectors, 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.

If `StartDate` is a serial date number or date character vector, `TargetDate` is returned as a serial date number. Use `datestr` to convert serial date numbers to formatted date character vectors.

If `StartDate` is a datetime array, then `TargetDate` is returned as a datetime array.

`TargetDate = datemnth(___,DayFlag,Basis,EndMonthRule)` determines a date in a future or past month based on movement either forward or backward in time by a given number of months, using optional input arguments for `DayFlag`,`Basis`, and `EndMonthRule`.

Examples

Determine the Dates of Days in a Future Month

Determine the `TargetDate` in a future month using a date character vector for `StartDate`.

```
StartDate = '03-Jun-1997';
NumberMonths = 6;
DayFlag = 0;
Basis = 0;
EndMonthRule = 1;

TargetDate = datemnth(StartDate, NumberMonths, DayFlag,...
Basis, EndMonthRule)

TargetDate = 729727

datestr(TargetDate)

ans =
'03-Dec-1997'
```

Determine the `TargetDate` in a future month using a datetime array for `StartDate`.

```
Day = datemnth(datetime('3-jun-2001','Locale','en_US'), 6, 0, 0, 0)

Day = datetime
   03-Dec-2001
```

Determine the `TargetDate` in a future month using a vector for `NumberMonths`.

```
NumberMonths = [1; 3; 5; 7; 9];
TargetDate = datemnth('31-jan-2001', NumberMonths);
datestr(TargetDate)

ans = 5x11 char array
    '28-Feb-2001'
    '30-Apr-2001'
    '30-Jun-2001'
    '31-Aug-2001'
    '31-Oct-2001'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`NumberMonths` — Number of months in future (positive) or past (negative)

positive or negative integers

Number of months in future (positive) or past (negative), specified as an `N`-by-`1` or `1`-by-`N` vector containing positive or negative integers.

Data Types: `double`

`DayFlag` — Flag for how actual day number for target date in future or past month is determined

`0` (day number should be the day in the future or past month corresponding to the actual day number of the start date) (default) | numeric with values `0`, `1`, or `2`

Flag for how the actual day number for the target date in future or past month is determined, specified as an `N`-by-`1` or `1`-by-`N` vector using a numeric with values`0`, `1`, or `2`.

Possible values are:

- `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`.

Data Types: `double`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis to be used when determining the past or future date, specified as a scalar value with an integer with value of `0` through `13`, or an `N`-by-`1` or `1`-by-`N` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days

`0` (not in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as a scalar with a nonnegative integer `0` or `1`, or as an `N`-by-`1` or `1`-by-`N` vector of values `0` or `1`.

- `0` = Ignore rule, meaning that rule is not in effect.
- `1` = Set rule on, 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.

Data Types: `logical`

Output Arguments

`TargetDate` — Target date in the future or past month
vector

Target date in the future or past month, returned as an `N`-by-`1` or `1`-by-`N` vector containing the serial date number (default) or datetime (if `StartDate` is a datetime array) of the target date.

See Also

`datestr` | `datetime` | `datevec` | `days360` | `days365` | `daysact` | `daysdif` | `wrkdydif`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2
"UICalendar User Interface" on page 15-4

Introduced before R2006a

datewrkdy

Date of future or past workday

Syntax

```
EndDate = datewrkdy(StartDate,NumberWorkDays,NumberHolidays)
```

Description

`EndDate = datewrkdy(StartDate,NumberWorkDays,NumberHolidays)` returns the serial number of the date a given number of workdays before or after the start date.

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 character vectors, 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.

If `StartDate` is a serial date number or date character vector, `EndDate` is returned as a date number. Use `datestr` to convert serial date numbers to formatted date character vectors.

If `StartDate` is a datetime array, then `EndDate` is returned as a datetime array.

Examples

Determine the Date for a Future Workday

Determine the `EndDate` for a future workday using a date character vector for `StartDate`.

```
StartDate = '20-Dec-1994';
NumberWorkDays = 16;
```

```
NumberHolidays = 2;

EndDate = datewrkdy(StartDate, NumberWorkDays, NumberHolidays)

EndDate = 728671

datestr(EndDate)

ans =
'12-Jan-1995'
```

Determine the `EndDate` for a future workday using a datetime array for `StartDate`.

```
EndDate = datewrkdy(datetime('12-dec-2000','Locale','en_US'), 16, 2)

EndDate = datetime
   04-Jan-2001
```

Determine the `EndDate` for future workdays using a vector for `NumberWorkDays`.

```
NumberWorkDays = [16; 20; 44];
EndDate = datewrkdy('12-dec-2000', NumberWorkDays, 2);
datestr(EndDate)

ans = 3x11 char array
    '04-Jan-2001'
    '10-Jan-2001'
    '13-Feb-2001'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`NumberWorkDays` — Number of work or business days in future (positive) or past (negative)
positive or negative integers

Number of work or business days in future (positive) or past (negative) that includes the starting date, specified as an `N`-by-`1` or `1`-by-`N` vector containing positive or negative integers.

`NumberHolidays` and `NumberWorkDays` must have the same sign.

Data Types: `double`

`NumberHolidays` — Number of holidays within `NumberWorkDays`
positive or negative integers

Number of holidays within `NumberWorkDays`, specified as positive or negative integers using an `N`-by-`1` or `1`-by-`N` containing values for the number of days movement in terms of holidays into the future (if positive) or past (if negative).

`NumberHolidays` and `NumberWorkDays` must have the same sign.

Data Types: `double`

Output Arguments

`EndDate` — Date of future or past workday
vector

Date of future or past workday, returned as an `N`-by-`1` or `1`-by-`N` vector containing the serial date number (default) or the datetime (if `StartDate` is a datetime array) of the future or past date.

See Also

`busdate` | `datetime` | `holidays` | `isbusday` | `wrkdydif`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

day

Day of month

Syntax

```
DayMonth = day(Date)
DayMonth = day(___,F)
```

Description

`DayMonth = day(Date)` returns the day of the month given a serial date number or date character vector.

`DayMonth = day(___,F)` returns the day of the month, given a serial date number, or character vector and an optional argument, `F`, defining the date format for `Date`.

Examples

Determine the Day of the Month for Various Date Formats

Find the day of the month using a serial date number.

```
DayMonth = day(730544)
```

```
DayMonth = 28
```

Find the day of the month using a date character vector.

```
DayMonth = day('2/28/00')
```

```
DayMonth = 28
```

Find the day of the month using a date character vector with an optional argument `F` for a country-specific date format.

```
DayMonth = day('28/02/00','dd/mm/yyyy')

DayMonth = 28
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Date` — Date to determine day of month

serial date number | date character vector | cell array of date character vectors

Date to determine day of month, specified as a serial date number, date character vector, or cell array of date character vectors.

All the date character vectors in `Date` must have same date character vector format. For more information on supported date character vector formats, see `datestr`.

Example: `DayMonth = day({'2/28/00','3/10/06'})`

Data Types: `single` | `double` | `char` | `cell`

`F` — Country-specific date format

character vector designating date format

Country-specific date format, specified as a character vector to designate the date format for the input argument `Date`. For more information on supported date character vector format symbols, see `datestr`. Note, formats with `'Q'` are not accepted.

Example: `DayMonth = day('28/02/00','dd/mm/yyyy')`

Data Types: `char`

Output Arguments

`DayMonth` — Day of the month

nonnegative integer

Day of the month, returned as a nonnegative integer, given a serial date number, or date character vector.

See Also

`datestr` | `datevec` | `eomday` | `month` | `year`

Topics

"Handle and Convert Dates" on page 2-2

Introduced before R2006a

days252bus

Number of business days between dates

Syntax

```
NumberDays = days252bus(StartDate,EndDate)

NumberDays = days252bus(StartDate,EndDate,HolidayVector)
```

Arguments

`StartDate`	`N`-by-`1` or `1`-by-`N` vector or scalar value, in serial date number, date character vector, or datetime array form, representing the start date.
`EndDate`	`N`-by-`1` or `1`-by-`N` vector or scalar value, in serial date number, date character vector, or datetime array form, representing the end date.
`HolidayVector`	(Optional) `N`-by-`1` or `1`-by-`N` vector, in serial date number, date character vector, or datetime array form, representing holidays.

Description

`NumberDays = days252bus(StartDate,EndDate,HolidayVector)` computes the number of business days (that is, non-holiday or non-weekend) between the two input dates. Note that a holiday vector may be optionally specified; if it is not, then the `holidays.m` file is used to determine the holidays.

`days252bus` returns `NumberDays`, a `N`-by-`1` or `1`-by-`N` vector or scalar value for the number of days between two dates. `NumberDays` returns as a double for serial date number, date character vector, and datetime inputs.

Examples

Computes the Number of Business Days Between Two Input Dates

This example shows how to compute the number of business days (i.e. non-holiday or non-weekend) between two dates using the days252bus convention.

```
NumberDays = days252bus('1/1/2009', '8/1/2009')
```

```
NumberDays = 146
```

Computes the Number of Business Days Between Two Input Dates Using a datetime Array

This example shows how to compute the number of business days (i.e. non-holiday or non-weekend) between two dates, specified as a datetime array, using the `days252bus` convention.

```
NumberDays = days252bus(datetime('1-Jan-2009','Locale','en_US'), '8/1/2009')
```

```
NumberDays = 146
```

- “Handle and Convert Dates” on page 2-2

See Also

`datetime` | `days360psa` | `days365` | `daysact` | `daysdif`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

days360

Days between dates based on 360-day year

Syntax

```
NumDays = days360(StartDate,EndDate)
```

Description

`NumDays = days360(StartDate,EndDate)` returns the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

Either input argument 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 character vectors, 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.

Examples

Determine the Number of Days Between Two Dates Based on a 360-Day Year

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate`.

```
NumDays = days360('15-jan-2000', '15-mar-2000')
```

```
NumDays = 60
```

Determine the `NumDays` using a datetime array for `StartDate`.

```
NumDays = days360(datetime('15-jan-2000','Locale','en_US'), '15-mar-2000')
```

```
NumDays = 60
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];
NumDays = days360('15-jan-2000', MoreDays)

NumDays =

    60
    90
   150
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDays` — Number of days between two dates

vector

Number of days between two dates, returned as a scalar or an `N`-by-`1` or `1`-by-`N` vector containing the number of days.

`NumDays` returns as a double for serial date number, date character vector, or datetime inputs for `StartDate` and `EndDate`.

References

[1] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures*. Vol. 2, Spring 1995.

See Also

`datetime` | `days365` | `daysact` | `daysdif` | `wrkdydif` | `yearfrac`

Topics

"Handle and Convert Dates" on page 2-2

Introduced before R2006a

days360e

Days between dates based on 360-day year (European)

Syntax

```
NumDays = days360e(StartDate,EndDate)
```

Description

`NumDays = days360e(StartDate,EndDate)` returns the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative. This day count convention is used primarily in Europe. Under this convention, all months contain 30 days.

Either input argument 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 character vectors, then `EndDate` must be an `N`-by-`1` vector of integers or a single integer. “Determine the Number of Days Between Two Dates Given a Basis of 30/360 Based on European Convention” on page 18-576`NumDays` is then an `N`-by-`1` vector of date numbers.

Examples

Determine the Number of Days Between Two Dates Given a Basis of 30/360 Based on European Convention

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate` for the month of January.

```
StartDate = '1-Jan-2002';
EndDate = '1-Feb-2002';
NumDays = days360e(StartDate, EndDate)

NumDays = 30
```

Determine the `NumDays` in the month of January using a datetime array for `StartDate`.

```
NumDays = days360e(datetime('1-Jan-2002','Locale','en_US'), '1-Feb-2002')
```

```
NumDays = 30
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];
NumDays = days360e('15-jan-2000', MoreDays)
```

```
NumDays =

    60
    90
   150
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDays` — Number of days between two dates given a basis of 30/360 based on European convention

vector

Number of days between two dates given a basis of 30/360 based on European convention, returned as a scalar or an `N`-by-`1` or `1`-by-`N` vector containing the number of days.

`NumDays` returns as a double for serial date number, date character vector, or datetime inputs for `StartDate` and `EndDate`.

References

[1] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures.* Vol. 2, Spring 1995.

See Also

`datetime` | `days360` | `days360isda` | `days360psa`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

days360isda

Days between dates based on 360-day year (International Swap Dealer Association (ISDA) compliant)

Syntax

```
NumDays = days360isda(StartDate,EndDate)
```

Description

`NumDays = days360isda(StartDate,EndDate)` returns the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days) and is International Swap Dealer Association (ISDA) compliant. If `EndDate` is earlier than `StartDate`, `NumDays` is negative. Under this convention, all months contain 30 days.

Either input argument 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 character vectors, 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.

Examples

Determine the Number of Days Between Two Dates Given a Basis of 30/360 Based on ISDA Compliance

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate` for the month of January.

```
StartDate = '1-Jan-2002';
EndDate = '1-Feb-2002';
NumDays = days360isda(StartDate, EndDate)
```

```
NumDays = 30
```

Determine the `NumDays` in the month of January using a datetime array for `StartDate`.

```
NumDays = days360isda(datetime('1-Jan-2002','Locale','en_US'), '1-Feb-2002')
```

```
NumDays = 30
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];
NumDays = days360isda('15-jan-2000', MoreDays)
```

```
NumDays =

    60
    90
   150
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDays` — Number of days between two dates given a basis of 30/360 based on European convention

vector

Number of days between two dates given a basis of 30/360 based on International Swap Dealer Association (ISDA) compliance, returned as a scalar or an `N`-by-`1` or `1`-by-`N` vector containing the number of days.

`NumDays` returns as a double for serial date number, date character vector, or datetime inputs for `StartDate` and `EndDate`.

References

[1] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures.* Vol. 2, Spring 1995.

See Also

`datetime` | `days360` | `days360e`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

days360psa

Days between dates based on 360-day year (Public Securities Association (PSA) compliant)

Syntax

```
NumDays = days360isda(StartDate,EndDate)
```

Description

`NumDays = days360isda(StartDate,EndDate)` returns the number of days between `StartDate` and `EndDate` based on a 360-day year (that is, all months contain 30 days) and is Public Securities Association (PSA) compliant. If `EndDate` is earlier than `StartDate`, `NumDays` is negative. Under this convention, all months contain 30 days.

Either input argument 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 character vectors, 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.

Examples

Determine the Number of Days Between Two Dates Given a Basis of 30/360 Based on PSA Compliance

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate` for the month of January.

```
StartDate = '1-Jan-2002';
EndDate = '1-Feb-2002';
NumDays = days360psa(StartDate, EndDate)

NumDays = 30
```

Determine the `NumDays` in the month of January using a datetime array for `StartDate`.

```
NumDays = days360psa(datetime('1-Jan-2002','Locale','en_US'), '1-Feb-2002')
```

```
NumDays = 30
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];
NumDays = days360psa('15-jan-2000', MoreDays)
```

```
NumDays =

    60
    90
   150
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDays` — Number of days between two dates given a basis of 30/360 based on European convention

vector

Number of days between two dates given a basis of 30/360 based on Public Securities Association (PSA) compliance, returned as a scalar or an `N`-by-`1` or `1`-by-`N` vector containing the number of days.

`NumDays` returns as a double for serial date number, date character vector, or datetime inputs for `StartDate` and `EndDate`.

References

[1] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures.* Vol. 2, Spring 1995.

See Also

`datetime` | `days360` | `days360e` | `days360isda`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

days365

Days between dates based on 365-day year

Syntax

```
NumDays = days365(StartDate,EndDate)
```

Description

`NumDays = days365(StartDate,EndDate)` returns the number of days between `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. Under this convention, all months contain 30 days.

Either input argument 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 character vectors, 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.

Examples

Determine the Number of Days Between Two Dates Based on a 365-Day Year

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate`.

```
NumDays = days365('15-jan-2000', '15-mar-2000')
```

```
NumDays = 59
```

Determine the `NumDays` using a datetime array for `StartDate`.

```
NumDays = days365(datetime('15-jan-2000','Locale','en_US'), '15-mar-2000')
```

```
NumDays = 59
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];
NumDays = days365('15-jan-2000', MoreDays)
```

```
NumDays =

    59
    90
   151
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDays` — Number of days between two dates based on 365-day year

vector

Number of days between two dates based on a 365-day year, returned as a scalar or an `N`-by-`1` or `1`-by-`N` vector containing the number of days.

`NumDays` returns as a double for serial date number, date character vector, or datetime inputs for `StartDate` and `EndDate`.

References

[1] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures.* Vol. 2, Spring 1995.

See Also

`datetime` | `days360` | `daysact` | `daysdif` | `wrkdydif` | `yearfrac`

Topics

"Handle and Convert Dates" on page 2-2

Introduced before R2006a

daysact

Actual number of days between dates

Syntax

```
NumDays = daysact(StartDate)
NumDays = daysact(___,EndDate)
```

Description

`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.

`NumDays = daysact(___,EndDate)` returns the actual number of days between `StartDate` and the optional argument `EndDate`.

If `EndDate` is earlier than `StartDate`, `NumDays` is negative. Under this convention, all months contain 30 days.

Either input argument 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 character vectors, 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.

Examples

Determine the Number of Days Between Two Dates Based the Actual Number of Days

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate`.

```
NumDays = daysact('7-sep-2002',  '25-dec-2002')
```

```
NumDays = 109
```

Determine the `NumDays` using a datetime array for `StartDate`.

```
NumDays = daysact(datetime('7-sep-2002','Locale','en_US'),  '25-dec-2002')
```

```
NumDays = 109
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['09/07/2002'; '10/22/2002'; '11/05/2002'];
NumDays = daysact(MoreDays, '12/25/2002')
```

```
NumDays =

   109
    64
    50
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`NumDays` — Number of days between two dates based actual number of days
vector

Number of days between two dates based on the actual number of days, returned as a scalar or an `N`-by-`1` or `1`-by-`N` vector containing the number of days.

`NumDays` returns as a double for serial date number, date character vector, or datetime inputs for `StartDate` and `EndDate`.

References

[1] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures.* Vol. 2, Spring 1995.

See Also

`datenum` | `datetime` | `datevec` | `days360` | `days365` | `daysdif`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

daysadd

Date away from starting date for any day-count basis

Syntax

```
NewDate = daysadd(StartDate,NumDays)
NewDate = daysadd(___,Basis)
```

Description

`NewDate = daysadd(StartDate,NumDays)` returns a date `NewDate` number of days away from `StartDate`.

If `StartDate` is a serial date number or date character vector, `NewDate` is returned as a date number.

If `StartDate` is a datetime array, then `NewDate` is returned as a datetime array.

`NewDate = daysadd(___,Basis)` returns a date `NewDate` number of days away from `StartDate`, using the optional argument `Basis` for day-count.

If `StartDate` is a serial date number or date character vector, `NewDate` is returned as a date number.

If `StartDate` is a datetime array, then `NewDate` is returned as a datetime array.

Examples

Determine the Date for Given Number of Days Away From `StartDate`

Determine the `NewDate` using a date character vector for `StartDate`.

```
NewDate = daysadd('01-Feb-2004', 31)
```

```
NewDate = 732009

datestr(NewDate)

ans =
'03-Mar-2004'
```

Determine the `NewDate` using a datetime array for `StartDate`.

```
NewDate = daysadd(datetime('01-Feb-2004','Locale','en_US'), 31)

NewDate = datetime
   03-Mar-2004
```

Determine the `NewDate` using a vector for `StartDate`.

```
MoreDays = ['09/07/2002'; '10/22/2002'; '11/05/2002'];
NewDate = daysadd(MoreDays, 31 ,2)

NewDate =

      731497
      731542
      731556


datestr(NewDate)

ans = 3x11 char array
    '08-Oct-2002'
    '22-Nov-2002'
    '06-Dec-2002'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`NumDays` — Number of days from `StartDate`
positive or negative integer

Number of days from `StartDate`, specified an `N`-by-`1` or `1`-by-`N` vector using positive or negative integers. Enter a negative integer for dates before start date.

Data Types: `double`

`Basis` — Day-count basis of the instrument
`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or a `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note When using the 30/360 day-count basis, it is not always possible to find the exact date `NewDate` number of days away because of a known discontinuity in the method of counting days. A warning is displayed if this occurs.

Data Types: `single` | `double`

Output Arguments

`NewDate` — Date for given number of days away from `StartDate`

vector

Date for given number of days away from `StartDate`, returned as a scalar or an `N`-by-`1` vector containing dates.

If `StartDate` is a serial date number or date character vector, `NewDate` is returned as a date number.

If `StartDate` is a datetime array, then `NewDate` is returned as a datetime array.

References

[1] Stigum, Marcia L. and Franklin Robinson. *Money Market and Bond Calculations.* Richard D. Irwin, 1996, ISBN 1-55623-476-7

See Also

`datetime` | `daysdif`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

daysdif

Days between dates for any for any day-count basis

Syntax

```
NumDays = daysdif(StartDate,EndDate)
NumDays = daysdif(___,Basis)
```

Description

`NumDays = daysdif(StartDate,EndDate)` returns the number of days between dates `StartDate` and `EndDate`. The first date for `StartDate` is not included when determining the number of days between first and last date.

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 array of character vector dates, then`EndDate` must be an *n*-row array of character vector dates or a single date. `NumDays` is then an `N`-by-`1` vector of numbers.

`NumDays = daysdif(___,Basis)` returns the number of days between dates `StartDate` and `EndDate` using the optional argument `Basis` for day-count. The first date for `StartDate` is not included when determining the number of days between first and last date.

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 array of character vector dates, then `EndDate` must be an *n*-row array of character vector dates or a single date. `NumDays` is then an `N`-by-`1` vector of numbers.

Examples

Determine the Number of Days Between `StartDate` and `EndDate`

Determine the `NumDays` using date character vectors for `StartDate` and `EndDate`.

```
NumDays = daysdif('3/1/99', '3/1/00', 1)
```

```
NumDays = 360
```

Determine the `NumDays` using a datetime array for `StartDate`.

```
NumDays = daysdif(datetime('1-Mar-1999','Locale','en_US'), '3/1/00', 1)
```

```
NumDays = 360
```

Determine the `NumDays` using a vector for `EndDate`.

```
MoreDays = ['3/1/2001'; '3/1/2002'; '3/1/2003'];
NumDays = daysdif('3/1/98', MoreDays)
```

```
NumDays =

        1096
        1461
        1826
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis of the instrument, specified as an integer with a value of `0` through `13` or a `N`-by-`1` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

Output Arguments

`NumDays` — Number of days between dates `StartDate` and `EndDate`

integer

Number of days between the `StartDate` and `EndDate`. `NumDays` returns as a double for serial date number, date character vector, and datetime inputs.

The first date for `StartDate` is not included when determining the number of days between first and last date.

References

[1] Stigum, Marcia L. and Franklin Robinson. *Money Market and Bond Calculations.* Richard D. Irwin, 1996, ISBN 1-55623-476-7

See Also

`datetime` | `dec2thirtytwo`

Topics

"Handle and Convert Dates" on page 2-2

Introduced before R2006a

dec2thirtytwo

Decimal to thirty-second quotation

Syntax

```
[OutNumber,Fractions] = dec2thirtytwo(InNumber,Accuracy)
```

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.

Examples

Convert Decimal to Thirty-Second Quotation

This example shows two bonds that are quoted with decimal prices of 101.78 and 102.96. These prices are converted to fractions with a denominator of 32.

```
InNumber  = [101.78; 102.96];
[OutNumber, Fractions] = dec2thirtytwo(InNumber)
```

```
OutNumber =

   101
   102


Fractions =

    25
    31
```

Input Arguments

`InNumber` — Input number
numeric decimal fraction

Input number, specified as an `N`-by-`1` vector of numeric decimal fractions.

Data Types: `double`

`Accuracy` — Rounding
`1` (round down to nearest thirty second) (default) | numeric with values `1`, `2`, `4` or `10`

Rounding, specified as an `N`-by-`1` vector of accuracy desired. with numeric values of `1`, `2`, `4` or `10`. The values are: `1`, round down to nearest thirty second, `2` (nearest half), `4` (nearest quarter), or `10` (nearest decile).

Data Types: `double`

Output Arguments

`OutNumber` — Output number which is `InNumber` rounded downward to closest integer
numeric

Output number which is `InNumber` rounded downward to closest integer, returned as a numeric value.

`Fractions` — Fractional part in units of thirty-second
numeric

Fractional part in units of thirty-second, returned as a numeric value. The `Fractions` output conforms to accuracy as prescribed by the input `Accuracy`.

See Also

`thirtytwo2dec`

Introduced before R2006a

depfixdb

Fixed declining-balance depreciation schedule

Syntax

```
Depreciation = depfixdb(Cost,Salvage,Life,Period,Month)
```

Arguments

`Cost`	Scalar for the initial value of the asset.
`Salvage`	Scalar for the salvage value of the asset.
`Life`	Scalar value for the life of the asset in years.
`Period`	Scalar integer for the number of years to calculate.
`Month`	(Optional) Scalar value for the number of months in the first year of asset life. Default = `12`.

Description

`Depreciation = depfixdb(Cost,Salvage,Life,Period,Month)` calculates the fixed declining-balance depreciation for each period.

Examples

Compute the Fixed Declining-Balance Depreciation

This example shows how to calculate the depreciation for the first five years for a car is purchased for $11,000, with a salvage value of $1500, and a lifetime of eight years.

```
Depreciation = depfixdb(11000, 1500, 8, 5)
```

```
Depreciation =

   1.0e+03 *

    2.4251    1.8904    1.4737    1.1488    0.8955
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`depgendb` | `deprdv` | `depsoyd` | `depstln`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

depgendb

General declining-balance depreciation schedule

Syntax

```
Depreciation = depgendb(Cost,Salvage,Life,Factor)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Estimated salvage value of the asset.
`Life`	Number of periods over which the asset is depreciated.
`Factor`	Depreciation factor. `Factor` = `2` uses the double-declining-balance method.

Description

`Depreciation = depgendb(Cost,Salvage,Life,Factor)` calculates the declining-balance depreciation for each period.

Examples

Calculate the Declining-Balance Depreciation

A car is purchased for $10,000 and is to be depreciated over five years. The estimated salvage value is $1000. Using the double-declining-balance method, the function calculates the depreciation for each year and returns the remaining depreciable value at the end of the life of the car.

Define the depreciation.

```
Life = 5;
Salvage = 0;
Cost = 10000;
Factor=2;
```

Use `depgendb` to calculate the depreciation.

```
Depreciation = depgendb(10000, 1000, 5, 2)
```

```
Depreciation =

   1.0e+03 *

    4.0000    2.4000    1.4400    0.8640    0.2960
```

The large value returned at the final year is the sum of the depreciation over the life time and is equal to the difference between the `Cost` and `Salvage`. The value of the asset in the final year is computed as (`Cost` - `Salvage`) = `Sum_Depreciation_Upto_Final_Year`.

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`depfixdb` | `deprdv` | `depsoyd` | `depstln`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

deprdv

Remaining depreciable value

Syntax

```
Value = deprdv(Cost,Salvage,Accum)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Salvage value of the asset.
`Accum`	Accumulated depreciation of the asset for prior periods.

Description

`Value = deprdv(Cost,Salvage,Accum)` returns the remaining depreciable value for an asset.

Examples

Compute the Remaining Depreciable Value for an Asset

This example shows how to find the remaining depreciable value after six years for the cost of an asset for $13,000 with a life of 10 years. The salvage value is $1000.

```
Accum = depstln(13000, 1000, 10) * 6
```

```
Accum = 7200
```

```
Value = deprdv(13000, 1000, 7200)
```

```
Value = 4800
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`depfixdb` | `depgendb` | `depsoyd` | `depstln`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

depsoyd

Sum of years' digits depreciation

Syntax

```
Sum = depsoyd(Cost,Salvage,Life)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Salvage value of the asset.
`Life`	Depreciable life of the asset in years.

Description

`Sum = depsoyd(Cost,Salvage,Life)` calculates the depreciation for an asset using the sum of years' digits method. `Sum` is a `1`-by-`Life` vector of depreciation values with each element corresponding to a year of the asset's life.

Examples

Compute the Depreciation for an Asset Using the Sum of Years' Digits Method

This example shows how to calculate the depreciation for an asset using the sum of years' digits method where the asset is $13,000 with a life of 10 years. The salvage value of the asset is $1000.

```
Sum = depsoyd(13000, 1000, 10)'
```

```
Sum =
```

```
1.0e+03 *

 2.1818
 1.9636
 1.7455
 1.5273
 1.3091
 1.0909
 0.8727
 0.6545
 0.4364
 0.2182
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`depfixdb` | `depgendb` | `deprdv` | `depstln`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

depstln

Straight-line depreciation schedule

Syntax

```
Depreciation = depstln(Cost,Salvage,Life)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Salvage value of the asset.
`Life`	Depreciable life of the asset in years.

Description

`Depreciation = depstln(Cost,Salvage,Life)` calculates straight-line depreciation for an asset.

Examples

Compute the Straight-Line Depreciation for an Asset

This example shows how to calculate the straight-line depreciation for an asset that costs $13,000 with a life of 10 years. The salvage value of the asset is $1000.

```
Depreciation = depstln(13000, 1000, 10)
```

```
Depreciation = 1200
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`depfixdb` | `depgendb` | `deprdv` | `depsoyd`

Topics

"Analyzing and Computing Cash Flows" on page 2-21

Introduced before R2006a

diff

Differencing

Syntax

```
newfts = diff(oldfts)
```

Description

`diff` computes the differences of the data series in a financial time series object. It returns another time series object containing the difference.

`newfts = diff(oldfts)` computes the difference of all the data in the data series of the object `oldfts` and returns the result in the object `newfts`. `newfts` is a financial time series object containing the same data series (names) as the input `oldfts`.

See Also

`diff`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

diffusion class

Diffusion-rate model component

Description

The `diffusion` constructor specifies the diffusion-rate component of continuous-time stochastic differential equations (SDEs). The diffusion-rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The diffusion-rate specification can be any `NVARS`-by-`NBROWNS` matrix-valued function G of the general form:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

And a diffusion-rate specification is associated with a vector-valued SDE of the form:
$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

- D is an `NVARS`-by-`NVARS` diagonal matrix, in which each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of a.
- V is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.

The diffusion-rate specification is flexible, and provides direct parametric support for static volatilities and state vector exponents. It is also extensible, and provides indirect support for dynamic/nonlinear models via an interface. This enables you to specify virtually any diffusion-rate specification.

Construction

`DiffusionRate = diffusion(Alpha,Sigma)` constructs a default `diffusion` object.

For more information on constructing a `diffusion` object, see `drift`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state $X(t)$ and is invoked with both input arguments.

`Alpha` — `Return` represents the parameter D

array or deterministic function of time

`Alpha` represents the parameter D, specified as an array or deterministic function of time.

If you specify `Alpha` as an array, it represents an `NVARS`-by-`1` column vector of exponents.

As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` matrix.

If you specify it as a function of time and state, `Alpha` must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter V
array or deterministic function of time

`Sigma` represents the parameter V, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

Note Although the `diffusion` constructor enforces no restrictions on the signs of these volatility parameters, each parameter is specified as a positive value.

Properties

`Rate` — Composite diffusion-rate function

value stored from diffusion-rate function (default) | function accessible by (t, X_t)

Composite diffusion-rate function, specified as: $G(t,X_t)$). The function stored in `Rate` fully encapsulates the combined effect of `Alpha` and `Sigma` where:

- `Alpha` is the state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma` is the volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

Attributes:

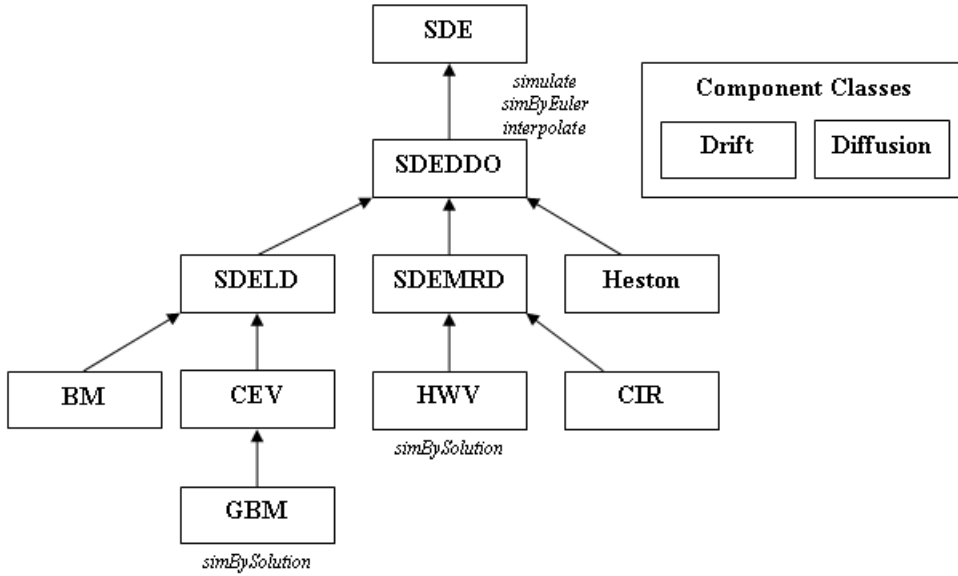
`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

Methods

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a `diffusion` Object

Create a diffusion-rate function `G`:

```
G = diffusion(1, 0.3)  % Diffusion rate function G(t,X)

G =
   Class DIFFUSION: Diffusion Rate Specification
   ---------------------------------------------
       Rate: diffusion rate function G(t,X(t))
```

```
Alpha: 1
Sigma: 0.3
```

The `diffusion` object displays like a MATLAB® structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because the `diffusion` class creates a model component rather than a model. `G` does not contain enough information to characterize the dimensionality of a problem.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the input arguments `Alpha` and `Sigma` as MATLAB arrays, they are associated with a specific parametric form. By contrast, when you specify either `Alpha` or `Sigma` as a function, you can customize virtually any diffusion-rate specification.

Accessing the output diffusion-rate parameters `Alpha` and `Sigma` with no inputs simply returns the original input specification. Thus, when you invoke diffusion-rate parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke diffusion-rate parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters `Alpha` and `Sigma` accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Specifically, parameters `Alpha` and `Sigma` evaluate the corresponding diffusion-rate component. Even if you originally specified an input as an array, `diffusion` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`drift` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

diffusion

Construct diffusion-rate model components

Syntax

```
DiffusionRate = diffusion(Alpha, Sigma)
```

Class

`diffusion`

Description

The `diffusion` constructor specifies the diffusion-rate component of continuous-time stochastic differential equations (SDEs). The diffusion-rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The diffusion-rate specification can be any `NVARS`-by-`NBROWNS` matrix-valued function G of the general form:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

associated with a vector-valued SDE of the form:
$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

- *D* is an `NVARS`-by-`NVARS` diagonal matrix, in which each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of *a*.
- *V* is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.

The diffusion-rate specification is flexible, and provides direct parametric support for static volatilities and state vector exponents. It is also extensible, and provides indirect support for dynamic/nonlinear models via an interface. This enables you to specify virtually any diffusion-rate specification.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Alpha`	`Alpha` determines the format of the parameter *D*. If you specify `Alpha` as an array, it must be an `NVARS`-by-`1` column vector of exponents. As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` column vector. If you specify `Alpha` as a function of time and state, it must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.
`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

Note Although the `diffusion` constructor enforces no restrictions on the signs of these volatility parameters, each parameter is usually specified as a positive value.

Output Arguments

`DiffusionRate`	Object of class `diffusion` that encapsulates the composite diffusion-rate specification, with the following displayed parameters: • `Rate`: The diffusion-rate function, G. `Rate` is the diffusion-rate calculation engine. It accepts the current time t and an `NVARS`-by-`1` state vector X_t as inputs, and returns an `NVARS`-by-`1` diffusion-rate vector. • `Alpha`: Access function for the input argument `Alpha`. • `Sigma`: Access function for the input argument `Sigma`.

Examples

Create a `diffusion` Object

Create a diffusion-rate function `G`:

```
G = diffusion(1, 0.3)  % Diffusion rate function G(t,X)
```

```
G =
   Class DIFFUSION: Diffusion Rate Specification
   ---------------------------------------------
       Rate: diffusion rate function G(t,X(t))
      Alpha: 1
      Sigma: 0.3
```

The `diffusion` object displays like a MATLAB® structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because the `diffusion` class creates a model component rather than a model. `G` does not contain enough information to characterize the dimensionality of a problem.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70

- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the input arguments `Alpha` and `Sigma` as MATLAB arrays, they are associated with a specific parametric form. By contrast, when you specify either `Alpha` or `Sigma` as a function, you can customize virtually any diffusion-rate specification.

Accessing the output diffusion-rate parameters `Alpha` and `Sigma` with no inputs simply returns the original input specification. Thus, when you invoke diffusion-rate parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke diffusion-rate parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters `Alpha` and `Sigma` accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Specifically, parameters `Alpha` and `Sigma` evaluate the corresponding diffusion-rate component. Even if you originally specified an input as an array, `diffusion` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

`drift` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

disc2zero

Zero curve given discount curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Compounding` and `Basis`.

Syntax

```
[ZeroRates,CurveDates] = disc2zero(DiscRates,CurveDates,Settle)
[ZeroRates,CurveDates] = disc2zero(___,Name,Value)
```

Description

`[ZeroRates,CurveDates] = disc2zero(DiscRates,CurveDates,Settle)` returns a zero curve given a discount curve and its maturity dates. If either inputs for `CurveDates` or `Settle` are datetime arrays, the output `CurveDates` is returned as datetime arrays.

`[ZeroRates,CurveDates] = disc2zero(___,Name,Value)` adds optional name-value pair arguments

Examples

Determine the Zero Curve Given a Discount Curve and Maturity Dates

Given the following discount factors `DiscRates` over a set of maturity dates `CurveDates`, and a settlement date `Settle`:

```
DiscRates = [0.9996
             0.9947
             0.9896
             0.9866
```

```
              0.9826
              0.9786
              0.9745
              0.9665
              0.9552
              0.9466];

CurveDates = [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 `disc2zero` which returns the zero curve `ZeroRates` at the maturity dates `CurveDates`.

```
[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates,...
Settle, Compounding, Basis)

ZeroRates =

    0.0487
    0.0510
    0.0523
    0.0524
    0.0530
    0.0526
    0.0530
    0.0532
    0.0549
    0.0536
```

```
CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `DiscRates` and `ZeroRates` are shown here only to the basis point. However, MATLAB® software computed them at full precision. If you enter `DiscRates` as shown, `ZeroRates` may differ due to rounding.

Determine the Zero Curve Given a Discount Curve and Maturity Dates Using datetime Inputs

Given the following discount factors, `DiscRates`, over a set of maturity dates, `CurveDates`, and a settlement date, `Settle`, use `datetime` inputs to return the zero curve, `ZeroRates`, at the maturity dates, `CurveDates`.

```
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');
Compounding = 365;
Basis = 3;

CurveDates = datetime(CurveDates,'ConvertFrom','datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');

[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates,...
Settle, Compounding, Basis)

ZeroRates =

    0.0487
    0.0510
    0.0523
    0.0524
    0.0530
    0.0526
    0.0530
    0.0532
    0.0549
    0.0536


CurveDates = 10x1 datetime array
   06-Nov-2000 00:00:00
   11-Dec-2000 00:00:00
   15-Jan-2001 00:00:00
   05-Feb-2001 00:00:00
   04-Mar-2001 00:00:00
   02-Apr-2001 00:00:00
   30-Apr-2001 00:00:00
   25-Jun-2001 00:00:00
   04-Sep-2001 00:00:00
   12-Nov-2001 00:00:00
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`DiscRates` — Discount factors

decimal fraction

Discount factors, specified as a column vector of decimal fractions. In aggregate, the factors in `DiscRates` constitute a discount curve for the investment horizon represented by `CurveDates`.

Data Types: `double`

`CurveDates` — Maturity dates

serial date number | date character vector | datetime

Maturity dates that correspond to the discount factors in `DiscRates`, specified as a column vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`Settle` — Common settlement date for discount rates in `DiscRates`

serial date number | date character vector | datetime

Common settlement date for the discount rates in `DiscRates`, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[ZeroRates,CurveDates] = disc2zero(DiscRates,CurveDates,Settle,'Compounding',6,'Basis',9)`

`Compounding` — Rate at which output zero rates are compounded when annualized

`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Rate at which the output zero rates are compounded when annualized, specified as a numeric value. Allowed values are:

- `0` — Simple interest (no compounding)
- `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

Data Types: `double`

`Basis` — Day-count basis used for annualizing output zero rates

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count basis used for annualizing the output zero rates, specified as a numeric value. Allowed values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`ZeroRates` — Zero curve for investment horizon represented by `CurveDates`
numeric

Zero curve for the investment horizon represented by `CurveDates`, returned as a column vector of decimal fractions. The zero rates are the yields to maturity on theoretical zero-coupon bonds.

`CurveDates` — Maturity dates that correspond to `ZeroRates`
serial date number | date character vector | datetime

Maturity dates that correspond to the `ZeroRates`, returned as a column vector. This vector is the same as the input vector `CurveDates`, but the output is sorted by ascending maturity. If either inputs for `CurveDates` or `Settle` are datetime arrays, the output `CurveDates` is returned as datetime arrays.

See Also

`datetime` | `fwd2zero` | `prbyzero` | `pyld2zero` | `zbtprice` | `zbtyield` | `zero2disc` | `zero2disc` | `zero2fwd` | `zero2pyld`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

discrate

Bank discount rate of money market security

Syntax

```
DiscRate = discrate(Settle,Maturity,Face,Price,Basis)
```

Arguments

`Settle`	Enter as serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.
`Maturity`	Enter as serial date numbers, date character vectors, or datetime arrays.
`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) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 For more information, see **basis** on page Glossary-0 .

Description

`DiscRate = discrate(Settle,Maturity,Face,Price,Basis)` finds the bank discount rate of a security. The bank discount rate normalizes by the face value of the security (for example, U. S. Treasury Bills) and understates the true yield earned by investors.

Examples

Compute the Bank Discount Rate of a Security

This example shows how to find the bank discount rate of a security.

```
DiscRate = discrate('12-jan-2000', '25-jun-2000', 100, 97.74, 0)
```

```
DiscRate = 0.0501
```

Compute the Bank Discount Rate of a Security Using datetime Inputs

This example shows how to use `datetime` inputs to find the bank discount rate of a security.

```
DiscRate = discrate(datetime('12-jan-2000','Locale','en_US'), datetime('25-jun-2000','I
```

```
DiscRate = 0.0501
```

- "Term Structure of Interest Rates" on page 2-45

References

Mayle. *Standard Securities Calculation Methods.* Volumes I-II, 3rd edition. Formula 1.

See Also

`acrudisc` | `datetime` | `fvdisc` | `prdisc` | `ylddisc`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

drift class

Drift-rate model component

Description

The `drift` constructor specifies the drift-rate component of continuous-time stochastic differential equations (SDEs). The drift-rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The drift-rate specification can be any `NVARS`-by-`1` vector-valued function F of the general form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

And a drift-rate specification is associated with a vector-valued SDE of the form

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.
- A and B are model parameters.

The drift-rate specification is flexible, and provides direct parametric support for static/linear drift models. It is also extensible, and provides indirect support for dynamic/nonlinear models via an interface. This enables you to specify virtually any drift-rate specification.

Construction

`DriftRate = drift(A,B)` constructs a default `drift` object.

For more information on constructing a `drift` object, see `drift`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`A` — `A` represents the parameter *A*

array or deterministic function of time

`A` represents the parameter *A*, specified as an array or deterministic function of time.

If you specify `A` as an array, it must be an `NVARS`-by-`1` column vector of intercepts.

As a deterministic function of time, when `A` is called with a real-valued scalar time `t` as its only input, `A` must produce an `NVARS`-by-`1` column vector. If you specify `A` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of intercepts when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`B` — `B` represents the parameter *B*

array or deterministic function of time

`B` represents the parameter *B*, specified as an array or deterministic function of time.

If you specify `B` as an array, it must be an `NVARS`-by-`NVARS` 2-dimensional matrix of state vector coefficients.

As a deterministic function of time, when `B` is called with a real-valued scalar time `t` as its only input, `B` must produce an `NVARS`-by-`NVARS` matrix. If you specify `B` as a function of time and state, it must generate an `NVARS`-by-`NVARS` matrix of state vector coefficients when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

Properties

`Rate` — Composite drift-rate function

value stored from drift-rate function (default) | function accessible by $F(t,X_t)$

Composite drift-rate function, specified as $F(t,X_t)$. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`, where `A` and `B` are:

The `drift` object's displayed parameters are:

- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

Attributes:

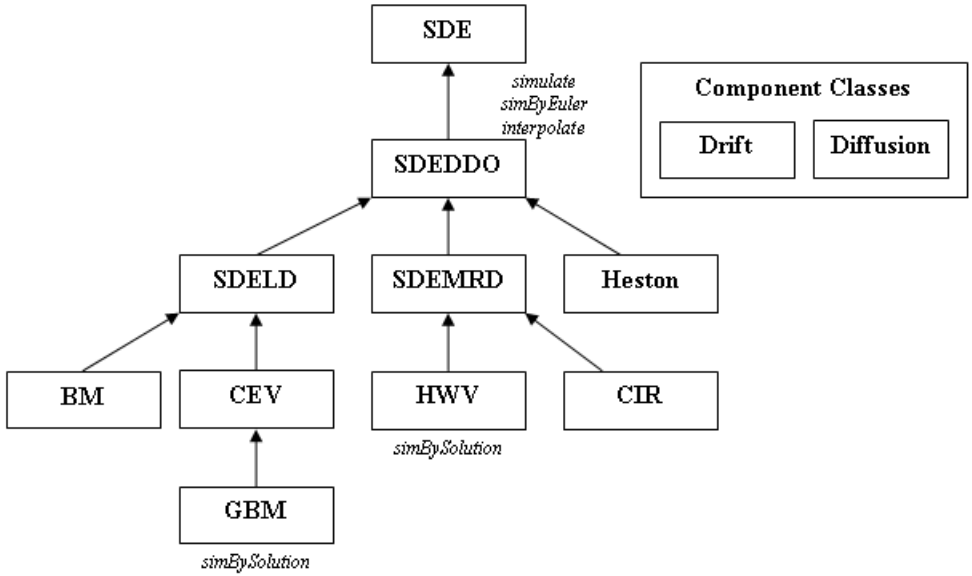
`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

Methods

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a `drift` Object

Create a drift-rate function `F`:

```
F = drift(0, 0.1)   % Drift rate function F(t,X)

F =
   Class DRIFT: Drift Rate Specification
   -------------------------------------
      Rate: drift rate function F(t,X(t))
         A: 0
         B: 0.1
```

The `drift` object displays like a MATLAB® structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because the `drift` class creates a model component rather than a model. `F` does not contain enough information to characterize the dimensionality of a problem.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the input arguments `A` and `B` as MATLAB arrays, they are associated with a linear drift parametric form. By contrast, when you specify either `A` or `B` as a function, you can customize virtually any drift-rate specification.

Accessing the output drift-rate parameters `A` and `B` with no inputs simply returns the original input specification. Thus, when you invoke drift-rate parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs.

function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke drift-rate parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters `A` and `B` accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Specifically, parameters `A` and `B` evaluate the corresponding drift-rate component. Even if you originally specified an input as an array, `drift` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70

“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

drift

Construct drift-rate model components

Syntax

```
DriftRate = drift(A, B)
```

Class

`drift`

Description

This constructor specifies the drift-rate component of continuous-time stochastic differential equations (SDEs). The drift-rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The drift-rate specification can be any `NVARS`-by-`1` vector-valued function F of the general form:

$$F(t, X_t) = A(t) + B(t)X_t$$

associated with a vector-valued SDE of the form

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.
- A and B are model parameters.

The drift-rate specification is flexible, and provides direct parametric support for static/linear drift models. It is also extensible, and provides indirect support for dynamic/nonlinear models via an interface. This enables you to specify virtually any drift-rate specification.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`A`	This argument represents the parameter *A*. If you specify `A` as an array, it must be an `NVARS`-by-`1` column vector. As a deterministic function of time, when `A` is called with a real-valued scalar time `t` as its only input, `A` must produce an `NVARS`-by-`1` column vector. If you specify `A` as a function of time and state, it must return an `NVARS`-by-`1` column vector when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

`B`	This argument represents the parameter B. If you specify `B` as an array, it must be an `NVARS`-by-`NVARS` 2-dimensional matrix. As a deterministic function of time, when `B` is called with a real-valued scalar time `t` as its only input, `B` must produce an `NVARS`-by-`NVARS` matrix. If you specify `B` as a function of time and state, it must return an `NVARS`-by-`NVARS` column vector when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.

Output Arguments

`DriftRate`	Object of class `drift` that encapsulates the composite drift-rate specification, with the following displayed parameters: • `Rate`: The drift-rate function, F. `Rate` is the drift-rate calculation engine. It accepts the current time t and an `NVARS`-by-`1` state vector X_t as inputs, and returns an `NVARS`-by-`1` drift-rate vector. • `A`: Access function for the input argument `A`. • `B`: Access function for the input argument `B`.

Examples

Create a `drift` Object

Create a drift-rate function `F`:

```
F = drift(0, 0.1)    % Drift rate function F(t,X)
```

```
F =
   Class DRIFT: Drift Rate Specification
   -------------------------------------
      Rate: drift rate function F(t,X(t))
         A: 0
         B: 0.1
```

The `drift` object displays like a MATLAB® structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to

the SDE representation, a summary of the dimensionality of the model does not appear, because the `drift` class creates a model component rather than a model. `F` does not contain enough information to characterize the dimensionality of a problem.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the input arguments `A` and `B` as MATLAB arrays, they are associated with a linear drift parametric form. By contrast, when you specify either `A` or `B` as a function, you can customize virtually any drift-rate specification.

Accessing the output drift-rate parameters `A` and `B` with no inputs simply returns the original input specification. Thus, when you invoke drift-rate parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke drift-rate parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters `A` and `B` accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Specifically, parameters `A` and `B` evaluate the corresponding drift-rate component. Even if you originally specified an input as an array, `drift` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

ecmlsrmle

Least-squares regression with missing data

Syntax

```
[Parameters,Covariance,Resid,Info] = ecmlsrmle(Data,Design,MaxIterations,TolParam,TolOb
```

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
	$\|Param_k - Param_{k-1}\| < TolParam \times (1 + \|Param_k\|)$
	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 $\|Obj_k - Obj_{k-1}\| < TolObj \times (1 + \|Obj_k\|)$ 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.
`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. Compute the full covariance matrix. • `'diagonal'` — Force the covariance matrix to be a diagonal matrix.

Description

`[Parameters, Covariance, Resid, Info] = ecmlsrmle(Data, Design, MaxIterations, TolParam, TolObj, Param0, Covar0, CovarFormat)` estimates a least-squares regression model with missing data. The model has the form

$$Data_k \sim N\left(Design_k \times Parameters, Covariance\right)$$

for samples $k = 1, \ldots, \texttt{NUMSAMPLES}$.

`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 contain each value of the objective function at each iteration of the estimation algorithm. The last value in this vector, `Obj(end)`, is the terminal estimate of the objective function. If you do least-squares, the objective function is the least-squares objective function.
- `Info.PrevParameters` — `NUMPARAMS`-by-`1` column vector of estimates for the model parameters from the iteration just prior to the terminal iteration.
- `Info.PrevCovariance` — `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance parameters from the iteration just prior to the terminal iteration.

Notes

If doing covariance-weighted least-squares, `Covar0` should usually be a diagonal matrix. Series with greater influence should have smaller diagonal elements in `Covar0` and series with lesser influence should have larger diagonal elements. Note that if doing CWLS, `Covar0` do not need to be a diagonal matrix even if `CovarFormat = 'diagonal'`.

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES` $\geq$ `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.

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.

Examples

See "Multivariate Normal Regression" on page 9-17, "Least-Squares Regression" on page 9-17, "Covariance-Weighted Least Squares" on page 9-18, "Feasible Generalized Least Squares" on page 9-19, and "Seemingly Unrelated Regression" on page 9-20.

References

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data.* 2nd Edition. John Wiley & Sons, Inc., 2002.

Xiao-Li Meng and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika.* Vol. 80, No. 2, 1993, pp. 267–278.

Joe Sexton and Anders Rygh Swensen. "ECM Algorithms that Converge at the Rate of EM." *Biometrika.* Vol. 87, No. 3, 2000, pp. 651–662.

A. P. Dempster, N.M. Laird, and D. B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society.* Series B, Vol. 39, No. 1, 1977, pp. 1–37.

See Also

`ecmlsrobj` | `ecmmvnrmle` | `ecmmvnrmle`

Topics

"Least-Squares Regression Without Missing Data" on page 9-18
"Covariance-Weighted Least Squares Without Missing Data" on page 9-19

Introduced in R2006a

ecmlsrobj

Log-likelihood function for least-squares regression with missing data

Syntax

```
Objective = ecmlsrobj(Data,Design,Parameters,Covariance)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `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` ≥ `1`.

- If `Design` is a cell array and `NUMSERIES` = `1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES` > `1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Examples

See “Multivariate Normal Regression” on page 9-17, “Least-Squares Regression” on page 9-17, “Covariance-Weighted Least Squares” on page 9-18, “Feasible Generalized Least Squares” on page 9-19, and “Seemingly Unrelated Regression” on page 9-20.

See Also

`ecmlsrmle` | `mvnrmle`

Topics

“Least-Squares Regression With Missing Data” on page 9-18

Introduced in R2006a

ecmmvnrfish

Fisher information matrix for multivariate normal regression model

Syntax

```
Fisher = ecmmvnrfish(Data,Design,Covariance,Method,MatrixFormat,CovarFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `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) Character vector 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) Character vector 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.
`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. The covariance matrix is a full matrix. • `'diagonal'` — The covariance matrix is a diagonal matrix.

Description

`Fisher = ecmmvnrfish(Data,Design,Covariance,Method,MatrixFormat,CovarFormat)` computes a Fisher information matrix based on current maximum likelihood or least-squares parameter estimates that account for missing data.

`Fisher` is a `NUMPARAMS`-by-`NUMPARAMS` Fisher information matrix or Hessian matrix. The size of `NUMPARAMS` depends on `MatrixFormat` and on current parameter estimates. If `MatrixFormat = 'full'`,

```
NUMPARAMS = NUMSERIES * (NUMSERIES + 3)/2
```

If `MatrixFormat = 'paramonly'`,

`NUMPARAMS = NUMSERIES`

Note `ecmmvnrfish` operates slowly if you calculate the full Fisher information matrix.

Examples

See "Multivariate Normal Regression" on page 9-17, "Least-Squares Regression" on page 9-17, "Covariance-Weighted Least Squares" on page 9-18, "Feasible Generalized Least Squares" on page 9-19, and "Seemingly Unrelated Regression" on page 9-20.

See Also

`ecmnmle` | `ecmnstd`

Topics

"Multivariate Normal Regression Without Missing Data" on page 9-17
"Fisher Information" on page 9-6
"Multivariate Normal Linear Regression" on page 9-2

Introduced in R2006a

ecmmvnrmle

Multivariate normal regression with missing data

Syntax

```
[Parameters,Covariance,Resid,Info] = ecmmvnrmle(Data,Design,MaxIterations,TolParam,TolC
```

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 $\|Param_k - Param_{k-1}\| < TolParam \times (1 + \|Param_k\|)$ 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 $\|Obj_k - Obj_{k-1}\| < TolObj \times (1 + \|Obj_k\|)$ 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.

`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. Compute the full covariance matrix. • `'diagonal'` — Force the covariance matrix to be a diagonal matrix.

Description

`[Parameters,Covariance,Resid,Info] = ecmmvnrmle(Data,Design,MaxIterations,TolParam,TolObj,Param0,Covar0,CovarFormat)` estimates a multivariate normal regression model with missing data. The model has the form

$$Data_k \sim N\left(Design_k \times Parameters, Covariance\right)$$

for samples $k = 1, \ldots, $ `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.

Note The covariance estimate `Covariance` cannot be derived from the residuals.

Another output, `Info`, is a structure that contains additional information from the regression. The structure has these fields:

- `Info.Obj` — A variable-extent column vector, with no more than `MaxIterations` elements, that contain 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.

Examples

See “Multivariate Normal Regression” on page 9-17, “Least-Squares Regression” on page 9-17, “Covariance-Weighted Least Squares” on page 9-18, “Feasible Generalized Least Squares” on page 9-19, and “Seemingly Unrelated Regression” on page 9-20.

References

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data.* 2nd Edition. John Wiley & Sons, Inc., 2002.

Xiao-Li Meng and Donald B. Rubin. “Maximum Likelihood Estimation via the ECM Algorithm.” *Biometrika.* Vol. 80, No. 2, 1993, pp. 267–278.

Joe Sexton and Anders Rygh Swensen. “ECM Algorithms that Converge at the Rate of EM.” *Biometrika.* Vol. 87, No. 3, 2000, pp. 651–662.

A. P. Dempster, N.M. Laird, and D. B. Rubin. “Maximum Likelihood from Incomplete Data via the EM Algorithm.” *Journal of the Royal Statistical Society.* Series B, Vol. 39, No. 1, 1977, pp. 1–37.

See Also

`ecmmvnrobj` | `mvnrmle`

Topics

“Multivariate Normal Regression With Missing Data” on page 9-17
“Multivariate Normal Linear Regression” on page 9-2

Introduced in R2006a

ecmmvnrobj

Log-likelihood function for multivariate normal regression with missing data

Syntax

```
Objective = ecmmvnrobj(Data,Design,Parameters,Covariance,CovarFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `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.

`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. The covariance matrix is a full matrix. • `'diagonal'` — The covariance matrix is a diagonal matrix.

Description

`Objective = ecmmvnrobj(Data,Design,Parameters,Covariance,CovarFormat)` computes a log-likelihood function based on current maximum likelihood parameter estimates with missing data. `Objective` is a scalar that contains the least-squares objective function.

Notes

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES` = `1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES` > `1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Examples

See "Multivariate Normal Regression" on page 9-17, "Least-Squares Regression" on page 9-17, "Covariance-Weighted Least Squares" on page 9-18, "Feasible Generalized Least Squares" on page 9-19, and "Seemingly Unrelated Regression" on page 9-20.

See Also

`ecmmvnrmle` | `mvnrmle` | `mvnrobj`

Topics

"Multivariate Normal Regression With Missing Data" on page 9-17
"Portfolios with Missing Data" on page 9-27
"Multivariate Normal Linear Regression" on page 9-2

Introduced in R2006a

ecmmvnrstd

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters, StdCovariance] = ecmmvnrstd(Data,Design,Covariance,Method,CovarFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are represented as `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) Character vector 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.
`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. The covariance matrix is a full matrix. • `'diagonal'` — The covariance matrix is a diagonal matrix.

Description

`[StdParameters,StdCovariance] = ecmmvnrstd(Data,Design,Covariance,Method,CovarFormat)` evaluates standard errors for a multivariate normal regression model with missing data. The model has the form

$$Data_k \sim N\left(Design_k \times Parameters, Covariance\right)$$

for samples $k = 1, \ldots, $ `NUMSAMPLES`.

`ecmmvnrstd` computes two outputs:

- `StdParameters` is a `NUMPARAMS`-by-`1` column vector of standard errors for each element of `Parameters`, the vector of estimated model parameters.
- `StdCovariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of standard errors for each element of `Covariance`, the matrix of estimated covariance parameters.

Note `ecmmvnrstd` operates slowly when you calculate the standard errors associated with the covariance matrix `Covariance`.

Notes

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES` = `1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES` > `1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Examples

See “Multivariate Normal Regression” on page 9-17, “Least-Squares Regression” on page 9-17, “Covariance-Weighted Least Squares” on page 9-18, “Feasible Generalized Least Squares” on page 9-19, and “Seemingly Unrelated Regression” on page 9-20.

References

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data. 2nd edition*, John Wiley & Sons, Inc., 2002.

See Also

`ecmmvnrmle` | `ecmmvnrstd`

Topics

“Multivariate Normal Regression With Missing Data” on page 9-17
“Multivariate Normal Linear Regression” on page 9-2

Introduced in R2006a

ecmnfish

Fisher information matrix

Syntax

```
Fisher = ecmnfish(Data,Covariance,InvCovariance,MatrixFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix of observed multivariate normal data
`Covariance`	`NUMSERIES`-by-`NUMSERIES` matrix with covariance estimate of `Data`
`InvCovariance`	(Optional) Inverse of covariance matrix: `inv(Covariance)`
`MatrixFormat`	(Optional) Character vector 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'`.

The data matrix has `NaNs` for missing observations. The multivariate normal model has

```
NUMPARAMS = NUMSERIES + NUMSERIES*(NUMSERIES + 1)/2
```

distinct parameters. Therefore, the full Fisher information matrix is of size `NUMPARAMS`-by-`NUMPARAMS`. The first `NUMSERIES` parameters are estimates for the mean of the data in `Mean` and the remaining `NUMSERIES*(NUMSERIES + 1)/2` parameters are estimates for the lower-triangular portion of the covariance of the data in `Covariance`, in row-major order.

If `MatrixFormat = 'meanonly'`, the number of parameters is reduced to `NUMPARAMS = NUMSERIES`, where the Fisher information matrix is computed for the mean parameters only. In this format, the routine executes fastest.

This routine expects the inverse of the covariance matrix as an input. If you do not pass in the inverse, the routine computes it. You can obtain an approximation for the lower-bound standard errors of estimation of the parameters from

```
Stderr = (1.0/sqrt(NumSamples)) .* sqrt(diag(inv(Fisher)));
```

Because of missing information, these standard errors can be smaller than the estimated standard errors derived from the expected Hessian matrix. To see the difference, compare to standard errors calculated with `ecmnhess`.

See Also

`ecmnhess` | `ecmnmle`

Topics

“Multivariate Normal Regression With Missing Data” on page 9-17
“Fisher Information” on page 9-6

Introduced before R2006a

ecmnhess

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) Character vector 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 on current parameter estimates, where

```
NUMPARAMS = NUMSERIES*(NUMSERIES + 3)/2
```

if `MatrixFormat = 'full'` and

```
NUMPARAMS = NUMSERIES
```

if `MatrixFormat = 'meanonly'`.

This routine is 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 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 can be larger than the estimated standard errors derived from the Fisher information matrix. To see the difference, compare to standard errors calculated from `ecmnfish`.

See Also

`ecmnfish` | `ecmnmle`

Topics

“Maximum Likelihood Estimation” on page 9-3

Introduced before R2006a

ecmninit

Initial mean and covariance

Syntax

```
[Mean,Covariance] = ecmninit(Data,InitMethod)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.
`InitMethod`	(Optional) Character vector 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 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`.

Algorithms

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 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`

Topics

"Mean and Covariance Estimation" on page 9-5

Introduced before R2006a

ecmnmle

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) Character vector 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` ≤ `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`.

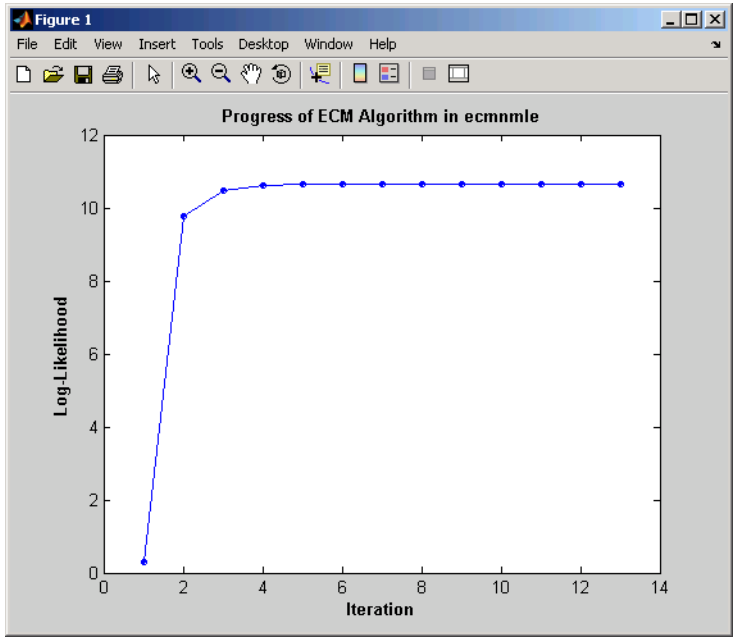
Description

`[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 program `ecmguidemo`.

Algorithms

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 robust but tends to result in slower convergence of the ECM algorithm.

diagonal

The `diagonal` method is a worst-case approach that deals with problematic data, such as disjoint series and excessive missing data (more than 33% of data missing). Of the three initialization methods, this method causes the slowest convergence of the ECM algorithm. If problems occur with this method, use display mode to examine convergence and modify either `MaxIterations` or `Tolerance`, or try alternative initial estimates with `Mean0` and `Covar0`. If all else fails, try

```
Mean0 = zeros(NumSeries);
Covar0 = eye(NumSeries,NumSeries);
```

Given estimates for mean and covariance from this routine, you can estimate standard errors with the companion routine `ecmnstd`.

Convergence

The ECM algorithm does not work for all patterns of missing values. Although it works in most cases, it can fail to converge if the covariance becomes singular. If this occurs,

plots of the log-likelihood function tend to have a constant upward slope over many iterations as the log of the negative determinant of the covariance goes to zero. In some cases, the objective fails to converge due to machine precision errors. No general theory of missing data patterns exists to determine these cases. An example of a known failure occurs when two time series are proportional wherever both series contain nonmissing values.

References

[1] Little, Roderick J. A. and Donald B. Rubin. *Statistical Analysis with Missing Data.* 2nd Edition. John Wiley & Sons, Inc., 2002.

[2] Meng, Xiao-Li and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika.* Vol. 80, No. 2, 1993, pp. 267–278.

[3] Sexton, Joe and Anders Rygh Swensen. "ECM Algorithms that Converge at the Rate of EM." *Biometrika.* Vol. 87, No. 3, 2000, pp. 651–662.

[4] Dempster, A. P., N. M. Laird, and Donald B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society.* Series B, Vol. 39, No. 1, 1977, pp. 1–37.

See Also

`ecmnfish` | `ecmnhess` | `ecmninit` | `ecmnobj` | `ecmnstd`

Topics

"Multivariate Normal Regression With Missing Data" on page 9-17
"Mean and Covariance Estimation" on page 9-5

Introduced before R2006a

ecmnobj

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`

Topics

"Multivariate Normal Regression Without Missing Data" on page 9-17
"Multivariate Normal Regression With Missing Data" on page 9-17

Introduced before R2006a

ecmnstd

Standard errors for mean and covariance of incomplete data

Syntax

```
[StdMean,StdCovariance] = ecmnstd(Data,Mean,Covariance,Method)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `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) Character vector 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

$$\theta - E[\theta] \sim N\left(0, I^{-1}(\theta)\right),$$

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 slow for `NUMSERIES > 10` or `NUMSAMPLES > 1000`.

See Also

`ecmnmle`

Topics

“Mean and Covariance Estimation” on page 9-5

Introduced before R2006a

effrr

Effective rate of return

Syntax

```
Return = effrr(Rate,NumPeriods)
```

Arguments

`Rate`	Annual percentage rate. Enter as a decimal fraction.
`NumPeriods`	Number of compounding periods per year, an integer.

Description

`Return = effrr(Rate,NumPeriods)` calculates the annual effective rate of return. Compounding continuously returns `Return` equivalent to `(e^Rate-1)`.

Examples

Compute the Annual Effective Rate of Return

This example shows how to find the effective annual rate of return based on an annual percentage rate of 9% compounded monthly.

```
Return = effrr(0.09, 12)
```

```
Return = 0.0938
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`nomrr`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

elpm

Compute expected lower partial moments for normal asset returns

Syntax

```
elpm(Mean,Sigma)

elpm(Mean,Sigma,MAR)

elpm(Mean,Sigma,MAR,Order)

Moment = elpm(Mean,Sigma,MAR,Order)
```

Arguments

`Mean`	`NUMSERIES` vector with mean returns for a collection of `NUMSERIES` assets.
`Sigma`	`NUMSERIES` vector with standard deviation of returns for a collection of `NUMSERIES` assets.
`MAR`	(Optional) Scalar minimum acceptable return (default `MAR` = `0`). This is a cutoff level of return such that all returns above `MAR` contribute nothing to the lower partial moment.
`Order`	(Optional) Either a scalar or a `NUMORDERS` vector of nonnegative integer moment orders. If no order specified, default `Order` = `0`, which is the shortfall probability. This function will not work for negative or noninteger orders.

Description

Given `NUMSERIES` asset returns with a vector of mean returns in a `NUMSERIES` vector `Mean`, a vector of standard deviations of returns in a `NUMSERIES` vector `Sigma`, a scalar minimum acceptable return `MAR`, and one or more nonnegative integer moment orders in

a NUMORDERS vector Order, compute expected lower partial moments (elpm) relative to MAR for each asset in a NUMORDERS-by-NUMSERIES matrix Moment.

The output, Moment, is a NUMORDERS-by-NUMSERIES matrix of expected lower partial moments with NUMORDERS Orders and NUMSERIES series, that is, each row contains expected lower partial moments for a given order.

Note To compute upper partial moments, reverse the signs of both the input Mean and MAR (do not reverse the signs of either Sigma or the output). This function computes expected lower partial moments with the mean and standard deviation of normally distributed asset returns. To compute sample lower partial moments from asset returns which have no distributional assumptions, use lpm.

Examples

See “Expected Lower Partial Moments” on page 7-16.

References

Vijay S. Bawa. "Safety-First, Stochastic Dominance, and Optimal Portfolio Choice." *Journal of Financial and Quantitative Analysis.* Vol. 13, No. 2, June 1978, pp. 255–271.

W. V. Harlow. "Asset Allocation in a Downside-Risk Framework." *Financial Analysts Journal.* Vol. 47, No. 5, September/October 1991, pp. 28–40.

W. V. Harlow and K. S. Rao. "Asset Pricing in a Generalized Mean-Lower Partial Moment Framework: Theory and Evidence." *Journal of Financial and Quantitative Analysis.* Vol. 24, No. 3, September 1989, pp. 285–311.

Frank A. Sortino and Robert van der Meer. "Downside Risk." *Journal of Portfolio Management.* Vol. 17, No. 5, Spring 1991, pp. 27–31.

See Also

lpm

Topics

"Expected Lower Partial Moments" on page 7-16
"Performance Metrics Overview" on page 7-2

Introduced in R2006b

emaxdrawdown

Compute expected maximum drawdown for Brownian motion

Syntax

```
EDD = emaxdrawdown(Mu,Sigma,T)
```

Arguments

`Mu`	Scalar. Drift term of a Brownian motion with drift.
`Sigma`	Scalar. Diffusion term of a Brownian motion with drift.
`T`	A time period of interest or a vector of times.

Description

`EDD = emaxdrawdown(Mu,Sigma,T)` computes the expected maximum drawdown for a Brownian motion for each time period in `T` using the following equation:

$dX(t) = \mu dt + \sigma dW(t)$.

If the Brownian motion is geometric with the stochastic differential equation

$dS(t) = \mu_0 S(t) dt + \sigma_0 S(t) dW(t)$

then use Ito's lemma with $X(t) = \log(S(t))$ such that

$\mu = \mu_0 - 0.5\sigma_0^2$,

$\sigma = \sigma_0$

converts it to the form used here.

The output argument `ExpDrawdown` is computed using an interpolation method. Values are accurate to a fraction of a basis point. Maximum drawdown is nonnegative since it is the change from a peak to a trough.

Note To compare the actual results from `maxdrawdown` with the expected results of `emaxdrawdown`, set the `Format` input argument of `maxdrawdown` to either of the nondefault values (`'arithmetic'` or `'geometric'`). These are the only two formats `emaxdrawdown` supports.

Examples

See "Expected Maximum Drawdown" on page 7-21.

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*. Vol. 41, Number 1, March 2004, pp. 147–161.

See Also

`maxdrawdown`

Topics

"Expected Maximum Drawdown" on page 7-21
"Performance Metrics Overview" on page 7-2

Introduced in R2006b

end

Last date entry

Syntax

```
end
```

Description

`end` returns the index to the last date entry in a financial time series object.

Examples

Consider a financial time series object called `MyFts`:

```
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]
```

Use `end` to return the last date entry in the financial time series object `myFts`.

```
myFts(end)

ans =

    desc:  My first FINTS
    freq:  Daily (1)

    'dates:  (1)'    'times:  (1)'    'Data1:  (1)'
    '03-Jan-2001'    '12:00'          [          6]
```

See Also

subsasgn | subsref

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

eomdate

Last date of month

Syntax

```
DayMonth = eomdate(Date)
DayMonth = eomdate(Year,Month,outputType)
```

Description

`DayMonth = eomdate(Date)` returns the serial date number of the last date of the month for the given `Date`.

`DayMonth = eomdate(Year,Month,outputType)` returns the serial date number of the last date of the month for the given year and month. However, if `outputType` is `'datetime'`, then `DayMonth` is a datetime array. By default, `outputType` is `'datenum'`.

Examples

Determine the Last Day of the Month for Various Dates

Find the last day of the month using `Year` and `Month`.

```
DayMonth = eomdate(2001, 2)
```

```
DayMonth = 730910
```

```
datestr(DayMonth)
```

```
ans =
'28-Feb-2001'
```

Find the last day of the month using multiples values for `Year` and a single `Month`.

```
Year = [2002 2003 2004 2005];
DayMonth = eomdate(Year, 2);
datestr(DayMonth)

ans = 4x11 char array
    '28-Feb-2002'
    '28-Feb-2003'
    '29-Feb-2004'
    '28-Feb-2005'
```

Find the last day of the month using a datetime array for `Date`.

```
DayMonth = eomdate(datetime('1-Jan-2015','Locale','en_US'))

DayMonth = datetime
   31-Jan-2015
```

Find the last day of the month using an `outputType` for `'datetime'`.

```
DayMonth = eomdate(2001, 2,'datetime')

DayMonth = datetime
   28-Feb-2001
```

- “Financial Time Series Operations” on page 12-8
- “Using Time Series to Predict Equity Return” on page 12-25

Input Arguments

`Date` — Date to determine last day of month

serial date number | date character vector | datetime array

Date to determine last day of month, specified as a serial date number, date character vector, or datetime array.

If `Date` is a serial date number or a date character vector, `DayMonth` is returned as a serial date number. If `Date` is a datetime array, then `DayMonth` is returned as a datetime array.

Use the function `datestr` to convert serial date numbers to formatted date character vectors or `datenum` to convert date and time to a serial date number.

Data Types: `single` | `double` | `char` | `datetime`

`Year` — Year to determine last date of month

four-digit nonnegative integer | vector of four-digit nonnegative integers

Year to determine last date of month, specified as a four-digit nonnegative integer.

Either input argument for `Year` or `Month` 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` output is then a `1`-by-*n* vector of date numbers.

Data Types: `single` | `double`

`Month` — Month to determine last date of month

integer from `1` through `12` | vector of integers from `1` through `12`

Month to determine last date of month, specified as an integer from `1` through `12`.

Either input argument for `Year` or `Month` 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` output is then a `1`-by-*n* vector of date numbers.

Data Types: `single` | `double`

`outputType` — Output date format

`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

Output date format, specified as a character vector with values `'datenum'` or `'datetime'`. If `outputType` is `'datenum'`, then `DayMonth` is a serial date number. However, if `outputType` is `'datetime'`, then `DayMonth` is a datetime array.

Data Types: `char`

Output Arguments

`DayMonth` — Last day of month
serial date number | datetime array

Last day of the month, returned as a serial date number or datetime array.

See Also

`datetime` | `day` | `eomday` | `lbusdate` | `month` | `year`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

eq (fts)

Multiple financial times series object equality

Syntax

```
tsobj_1 == tsobj_2

iseq = eq(tsobj_1,tsobj_2)
```

Arguments

`tsobj_1`	Financial time series object.
`tsobj_2`	Financial time series object.

Description

`tsobj_1 == tsobj_2` returns True (`1`) if both financial time series objects have the same dates, frequencies, data series names, and data values. Otherwise, `eq` returns False (`0`).

Note The data series names are case-sensitive, but do not have to be in the same order within each object.

Examples

Determine Multiple Financial Times Series Object Equality

This example shows how to determine if multiple financial times series objects are equal.

```
load disney
dis == dis
```

```
ans = logical
   1
```

- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`isequal`

Topics

“Using Time Series to Predict Equity Return” on page 12-25
“What Is the Financial Time Series App?” on page 13-2

Introduced before R2006a

estimateAssetMoments

Estimate mean and covariance of asset returns from data

Use the `estimateAssetMoments` function with a `Portfolio` object to estimate mean and covariance of asset returns from data.

For details on the workflow, see "Portfolio Object Workflow" on page 4-21.

Syntax

```
obj = estimateAssetMoments(obj,AssetReturns)
obj] = estimateAssetMoments(obj,AssetReturns,Name,Value)
```

Description

`obj = estimateAssetMoments(obj,AssetReturns)` estimates mean and covariance of asset returns from data for a `Portfolio` object.

`obj] = estimateAssetMoments(obj,AssetReturns,Name,Value)` estimates mean and covariance of asset returns from data for a Portfolio object with additional options for one or more `Name,Value` pair arguments.

Examples

Estimate Mean and Covariance of Asset Returns from Data for a Portfolio Object

To illustrate using the `estimateAssetMoments` function, generate random samples of 120 observations of asset returns for four assets from the mean and covariance of asset returns in the variables `m` and `C` with the `portsim` function. The default behavior `portsim` creates simulated data with estimated mean and covariance identical to the input moments `m` and `C`. In addition to a return series created by the `portsim` function in the variable `X`, a price series is created in the variable `Y`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
X = portsim(m', C, 120);
Y = ret2tick(X);
```

Given asset returns and prices in the variables `X` and `Y` from above, the following examples demonstrate equivalent ways to estimate asset moments for the Portfolio object. A Portfolio object is created in `p` with the moments of asset returns set directly in the `Portfolio` function and a second Portfolio object is created in `q` to obtain the mean and covariance of asset returns from asset return data in `X` using the `estimateAssetMoments` function.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
p = Portfolio('mean',m,'covar',C);
q = Portfolio;
q = estimateAssetMoments(q, X);

[passetmean, passetcovar] = getAssetMoments(p)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150


passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
```

```
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

```
[qassetmean, qassetcovar] = getAssetMoments(q)
```

```
qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150


qassetcovar =

    0.0005    0.0003    0.0002   -0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
   -0.0000    0.0010    0.0028    0.0102
```

Notice how either approach yields the same moments. The default behavior of the `estimateAssetMoments` function is to work with asset returns. If, instead, you have asset prices, such as in the variable `Y`, the `estimateAssetMoments` function accepts a parameter name `'DataFormat'` with a corresponding value set to `'prices'` to indicate that the input to the method is in the form of asset prices and not returns (the default parameter value for `'DataFormat'` is `'returns'`). The following example compares direct assignment of moments in the Portfolio object `p` with estimated moments from asset price data in `Y` in the Portfolio object `q`:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

X = portsim(m', C, 120);
Y = ret2tick(X);

p = Portfolio('mean',m,'covar',C);
```

```
q = Portfolio;
q = estimateAssetMoments(q, Y, 'dataformat', 'prices');

[passetmean, passetcovar] = getAssetMoments(p)

passetmean =

    0.0042
    0.0083
    0.0100
    0.0150


passetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102

[qassetmean, qassetcovar] = getAssetMoments(q)

qassetmean =

    0.0042
    0.0083
    0.0100
    0.0150


qassetcovar =

    0.0005    0.0003    0.0002   -0.0000
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
   -0.0000    0.0010    0.0028    0.0102
```

- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-48
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object. For more information on creating a portfolio object, see

- `Portfolio`

`AssetReturns` — Matrix or `fints` object that contains asset price data that can be converted to asset returns

matrix | `fints` object

Matrix or `fints` object that contains asset price data that can be converted to asset returns, specified by a `fints` object or `NumSamples`-by-`NumAssets` matrix for asset returns. Use the optional `'DataFormat'` argument to convert `AssetReturns` input data that is asset prices into asset returns. Be careful when using asset price data because portfolio optimization usually requires total returns and not simply price returns.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p= estimateAssetMoments(p, Y, 'dataformat', 'prices')`

`DataFormat` — Flag to convert input data as prices into returns

`'Returns'` (default) | character vector with values `'Returns'` or `'Prices'`

Flag to convert input data as prices into returns, specified using a character vector with the values:

- `'Returns'` — Data in `AssetReturns` contains asset total returns.
- `'Prices'` — Data in `AssetReturns` contains asset total return prices.

Data Types: `char`

`MissingData` — Flag indicating whether to use ECM algorithm or exclude samples with `NaN` values

`false` (default) | logical with value `true` or `false`

Flag indicating whether to use ECM algorithm or excludes samples with `NaN` values, specified as a logical with a value of `true` or `false`.

To handle time series with missing data (indicated with `NaN` values), the `MissingData` flag either uses the ECM algorithm to obtain maximum likelihood estimates in the presences of `NaN` values or excludes samples with `NaN` values. Since the default is `false`, it is necessary to specify `MissingData` as `true` to use the ECM algorithm.

Acceptable values for `MissingData` are:

- `false` — Do not use ECM algorithm to handle `NaN` values (exclude `NaN` values).
- `true` — Use ECM algorithm to handle `NaN` values.

For more information on the ECM algorithm, see `ecmnmle` and "Multivariate Normal Regression" on page 9-2.

Data Types: `logical`

`GetAssetList` — Flag indicating which asset names to use for asset list

`false` (default) | logical with value `true` or `false`

Flag indicating which asset names to use for the asset list, specified as a logical with a value of `true` or `false`. Acceptable values for `GetAssetList` are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this function and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is `'Asset'`.

If the `GetAssetList` flag is `false`, no action occurs, which is the default behavior.

Data Types: `logical`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio` object. For more information on creating a portfolio object, see

- `Portfolio`

Tips

You can also use dot notation to estimate the mean and covariance of asset returns from data.

```
obj = obj.estimateAssetMoments(AssetReturns);
```

See Also

`Portfolio` | `estimateBounds` | `portsim`

Topics

"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-48
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimateBounds

Estimate global lower and upper bounds for set of portfolios

Use the `estimateBounds` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate global lower and upper bounds for a set of portfolios.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[glb,gub,isbounded] = estimateBounds(obj)
[glb,gub,isbounded] = estimateBounds(obj,obtainExactBounds)
```

Description

`[glb,gub,isbounded] = estimateBounds(obj)` estimates global lower and upper bounds for set of portfolios.

`[glb,gub,isbounded] = estimateBounds(obj,obtainExactBounds)` estimates global lower and upper bounds for set of portfolios with an additional option specified for `obtainExactBounds`.

Examples

Create an Unbounded Portfolio for a Portfolio Object

Create an unbounded portfolio set.

```
p = Portfolio('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)
```

```
lb =
```

```
  -Inf
  -Inf


ub =

   1.0e-08 *

   -0.3712
       Inf


isbounded = logical
   0
```

The `estimateBounds` function returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Create an Unbounded Portfolio for a PortfolioCVaR Object

Create an unbounded portfolio set.

```
p = PortfolioCVaR('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

  -Inf
  -Inf


ub =

   1.0e-08 *

   -0.3712
       Inf
```

```
isbounded = logical
   0
```

The `estimateBounds` function returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

Create an Unbounded Portfolio for a PortfolioMAD Object

Create an unbounded portfolio set.

```
p = PortfolioMAD('AInequality', [1 -1; 1 1 ], 'bInequality', 0);
[lb, ub, isbounded] = estimateBounds(p)

lb =

  -Inf
  -Inf


ub =

   1.0e-08 *

   -0.3712
       Inf


isbounded = logical
   0
```

The `estimateBounds` function returns (possibly infinite) bounds and sets the `isbounded` flag to `false`. The result shows which assets are unbounded so that you can apply bound constraints as necessary.

- “Validate the Portfolio Problem for Portfolio Object” on page 4-104
- “Validate the CVaR Portfolio Problem” on page 5-96

- “Validate the MAD Portfolio Problem” on page 6-91
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`obtainExactBounds` — Flag to specify whether to solve for all bounds or to accept specified bounds whenever available

`true` (default) | logical

Flag to specify whether to solve for all bounds or to accept specified bounds whenever available, specified as a logical with values of `true` or `false`. If bounds are known, set `obtainExactBounds` to `false` to accept known bounds. The default for `obtainExactBounds` is `true`.

Data Types: `logical`

Output Arguments

`glb` — Global lower bounds for portfolio set

vector

Global lower bounds for portfolio set, returned as vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`gub` — Global upper bounds for portfolio set

vector

Global upper bounds for portfolio set, returned as vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`isbounded` — Indicator for whether portfolio set is empty, bounded, or unbounded
logical

Indicator for whether portfolio set is empty (`[]`), bounded (`true`), or unbounded (`false`), returned as a logical.

Note By definition, any portfolio set must be nonempty and bounded:

- If the set is empty, `isbounded = [ ]`.
- If the set is nonempty and unbounded, `isbounded = false`.
- If the set is nonempty and bounded, `isbounded = true`.
- If the set is empty, `glb` and `gub` are set to `NaN` vectors.

An `isbounded` value is returned for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

- You can also use dot notation to estimate the global lower and upper bounds for a given set of portfolios.

  ```
  [glb, gub, isbounded] = obj.estimateBounds;
  ```

- Estimated bounds are accurate in most cases to within `1.0e-8`. If you intend to use these bounds directly in a portfolio object, ensure that if you impose such bound constraints, a lower bound of `0` is probably preferable to a lower bound of, for example, `1.0e-10` for portfolio weights.

See Also

`checkFeasibility`

Topics

"Validate the Portfolio Problem for Portfolio Object" on page 4-104
"Validate the CVaR Portfolio Problem" on page 5-96
"Validate the MAD Portfolio Problem" on page 6-91
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimateFrontier

Estimate specified number of optimal portfolios on the efficient frontier

Use the `estimateFrontier` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate specified number of optimal portfolios on the efficient frontier.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[pwgt,pbuy,psell] = estimateFrontier(obj)
[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)
```

Description

`[pwgt,pbuy,psell] = estimateFrontier(obj)` estimates the specified number of optimal portfolios on the efficient frontier. When no value is specified for the optional input argument `NumPorts`, the default value of `10` is obtained from the hidden property `defaultNumPorts`.

`[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)` estimates the specified number of optimal portfolios on the efficient frontier with an additional option specified for `NumPorts`.

Examples

Obtain the Default Number of Efficient Portfolios for a Portfolio Object

Obtain the default number of efficient portfolios over the entire range of the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontier(p);
disp(pwgt);
```

```
  Columns 1 through 7

    0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630

  Columns 8 through 10

         0         0         0
    0.2314    0.0579         0
    0.1394    0.1468         0
    0.6292    0.7953    1.0000
```

Obtain Purchases and Sales for Portfolios on the Efficient Frontier for a Portfolio Object

Starting from the initial portfolio, the `estimateFrontier` function returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. Given an initial portfolio in `pwgt0`, you can obtain purchases and sales.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
```

```
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);

pwgt =

  Columns 1 through 7

    0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630

  Columns 8 through 10

         0         0         0
    0.2314    0.0579         0
    0.1394    0.1468         0
    0.6292    0.7953    1.0000


display(pbuy);

pbuy =

  Columns 1 through 7

    0.5891    0.4215    0.2540    0.0865         0         0         0
         0         0         0    0.0129    0.1049    0.1969    0.1049
         0         0         0         0         0         0         0
         0         0    0.0521    0.1113    0.1705    0.2297    0.3630

  Columns 8 through 10

         0         0         0
         0         0         0
         0         0         0
    0.5292    0.6953    0.9000


display(psell);

psell =

  Columns 1 through 7
```

```
         0         0         0         0    0.0810    0.2485    0.3000
    0.2631    0.1711    0.0791         0         0         0         0
    0.1596    0.1433    0.1270    0.1107    0.0944    0.0781    0.0680
    0.0664    0.0071         0         0         0         0         0

  Columns 8 through 10

    0.3000    0.3000    0.3000
    0.0686    0.2421    0.3000
    0.0606    0.0532    0.2000
         0         0         0
```

Obtain the Default Number of Efficient Portfolios for a PortfolioCVaR Object

Obtain the default number of efficient portfolios over the entire range of the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontier(p);

disp(pwgt);

  Columns 1 through 7
```

```
    0.8454    0.6847    0.5151    0.3541    0.1902    0.0314    0.0000
    0.0599    0.1427    0.2302    0.3165    0.3980    0.4733    0.3513
    0.0462    0.0639    0.0945    0.1079    0.1345    0.1583    0.1756
    0.0485    0.1087    0.1602    0.2215    0.2773    0.3371    0.4731

  Columns 8 through 10

    0.0000    0.0000    0.0000
    0.1806    0.0000    0.0000
    0.1916    0.2212    0.0000
    0.6278    0.7788    1.0000
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Purchases and Sales for Portfolios on the Efficient Frontier for a PortfolioCVaR Object

Starting from the initial portfolio, the `estimateFrontier` function returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. Given an initial portfolio in `pwgt0`, you can obtain purchases and sales.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);
p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);
```

```
display(pwgt);

pwgt =

  Columns 1 through 7

    0.8454    0.6847    0.5151    0.3541    0.1902    0.0314    0.0000
    0.0599    0.1427    0.2302    0.3165    0.3980    0.4733    0.3513
    0.0462    0.0639    0.0945    0.1079    0.1345    0.1583    0.1756
    0.0485    0.1087    0.1602    0.2215    0.2773    0.3371    0.4731

  Columns 8 through 10

    0.0000    0.0000    0.0000
    0.1806    0.0000    0.0000
    0.1916    0.2212    0.0000
    0.6278    0.7788    1.0000


display(pbuy);

pbuy =

  Columns 1 through 7

    0.5454    0.3847    0.2151    0.0541         0         0         0
         0         0         0    0.0165    0.0980    0.1733    0.0513
         0         0         0         0         0         0         0
         0    0.0087    0.0602    0.1215    0.1773    0.2371    0.3731

  Columns 8 through 10

         0         0         0
         0         0         0
         0    0.0212         0
    0.5278    0.6788    0.9000


display(psell);

psell =

  Columns 1 through 7

         0         0         0         0    0.1098    0.2686    0.3000
```

```
    0.2401    0.1573    0.0698         0         0         0         0
    0.1538    0.1361    0.1055    0.0921    0.0655    0.0417    0.0244
    0.0515         0         0         0         0         0         0

  Columns 8 through 10

    0.3000    0.3000    0.3000
    0.1194    0.3000    0.3000
    0.0084         0    0.2000
         0         0         0
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain the Default Number of Efficient Portfolios for a PortfolioMAD Object

Obtain the default number of efficient portfolios over the entire range of the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontier(p);

disp(pwgt);

  Columns 1 through 7
```

```
    0.8815    0.7151    0.5487    0.3817    0.2170    0.0499         0
    0.0431    0.1282    0.2128    0.2981    0.3825    0.4662    0.3609
    0.0389    0.0605    0.0826    0.1053    0.1241    0.1492    0.1786
    0.0365    0.0963    0.1559    0.2149    0.2764    0.3348    0.4605

  Columns 8 through 10

         0    0.0000    0.0000
    0.1755         0    0.0000
    0.2095    0.2266    0.0000
    0.6150    0.7734    1.0000
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Purchases and Sales for Portfolios on the Efficient Frontier for a PortfolioMAD Object

Starting from the initial portfolio, the `estimateFrontier` function returns purchases and sales to get from your initial portfolio to each efficient portfolio on the efficient frontier. Given an initial portfolio in `pwgt0`, you can obtain purchases and sales.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);
p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateFrontier(p);

display(pwgt);
```

```
pwgt =

  Columns 1 through 7

    0.8815    0.7151    0.5487    0.3817    0.2170    0.0499         0
    0.0431    0.1282    0.2128    0.2981    0.3825    0.4662    0.3609
    0.0389    0.0605    0.0826    0.1053    0.1241    0.1492    0.1786
    0.0365    0.0963    0.1559    0.2149    0.2764    0.3348    0.4605

  Columns 8 through 10

         0    0.0000    0.0000
    0.1755         0    0.0000
    0.2095    0.2266    0.0000
    0.6150    0.7734    1.0000


display(pbuy);

pbuy =

  Columns 1 through 7

    0.5815    0.4151    0.2487    0.0817         0         0         0
         0         0         0         0    0.0825    0.1662    0.0609
         0         0         0         0         0         0         0
         0         0    0.0559    0.1149    0.1764    0.2348    0.3605

  Columns 8 through 10

         0         0         0
         0         0         0
    0.0095    0.0266         0
    0.5150    0.6734    0.9000


display(psell);

psell =

  Columns 1 through 7

         0         0         0         0    0.0830    0.2501    0.3000
    0.2569    0.1718    0.0872    0.0019         0         0         0
    0.1611    0.1395    0.1174    0.0947    0.0759    0.0508    0.0214
```

```
    0.0635    0.0037         0         0         0         0         0

  Columns 8 through 10

    0.3000    0.3000    0.3000
    0.1245    0.3000    0.3000
         0         0    0.2000
         0         0         0
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`NumPorts` — Number of points to obtain on efficient frontier
value from hidden property `defaultNumPorts` (default value is `10`) (default) | scalar integer

Number of points to obtain on the efficient frontier, specified as a scalar integer.

Note If no value is specified for `NumPorts`, the default value is obtained from the hidden property `defaultNumPorts` (default value is `10`). If `NumPorts = 1`, this function returns the portfolio specified by the hidden property `defaultFrontierLimit` (current default value is `'min'`).

Data Types: `double`

Output Arguments

`pwgt` — Optimal portfolios on efficient frontier with specified number of portfolios spaced equally from minimum to maximum portfolio return
matrix

Optimal portfolios on the efficient frontier with specified number of portfolios spaced equally from minimum to maximum portfolio return, returned as a `NumAssets`-by-`NumPorts` matrix. `pwgt` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`pbuy` — Purchases relative to initial portfolio for optimal portfolios on efficient frontier
matrix

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`pbuy` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`psell` — Sales relative to initial portfolio for optimal portfolios on efficient frontier
matrix

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as a `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`psell` is returned for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

- You can also use dot notation to estimate the specified number of optimal portfolios over the entire efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontier(NumPorts);
```

- When introducing transaction costs and turnover constraints to the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object, the portfolio optimization objective contains a term with an absolute value. For more information on how Financial Toolbox handles such cases algorithmically, see “References” on page 18-728.

References

[1] Cornuejols, G., and R. Tutuncu. *Optimization Methods in Finance.* Cambridge University Press, 2007.

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109

"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimateFrontierByReturn

Estimate optimal portfolios with targeted portfolio returns

Use the `estimateFrontierByReturn` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate optimal portfolios with targeted portfolio returns.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierByReturn(obj,TargetReturn)
```

Description

`[pwgt,pbuy,psell] = estimateFrontierByReturn(obj,TargetReturn)` estimates optimal portfolios with targeted portfolio returns.

Examples

Obtain the Portfolio for Targeted Portfolio Returns for a Portfolio Object

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` function accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 6%, 9%, and 12%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);

display(pwgt);

pwgt =

    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022
```

Obtain the Portfolio for Targeted Portfolio Returns for a PortfolioCVaR Object

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` function accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 13%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioCVaR;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierByReturn(p, [0.07 0.10, 0.13]);

display(pwgt);
```

```
pwgt =

    0.7371    0.3071         0
    0.1504    0.3919    0.4396
    0.0286    0.1011    0.1360
    0.0839    0.1999    0.4244
```

The function `rng(`*seed*`)` is used to reset the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain the Portfolio for Targeted Portfolio Returns for a PortfolioMAD Object

To obtain efficient portfolios that have targeted portfolio returns, the `estimateFrontierByReturn` function accepts one or more target portfolio returns and obtains efficient portfolios with the specified returns. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio returns of 7%, 10%, and 13%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioMAD;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByReturn(p, [0.07 0.10, 0.13]);

display(pwgt);

pwgt =

    0.7436    0.3147    0.0000
    0.1357    0.3835    0.4422
    0.0328    0.0939    0.1324
```

```
    0.0879    0.2079    0.4254
```

The function `rng(`*seed*`)` is used to reset the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`TargetReturn` — Target values for portfolio return

vector

Target values for portfolio return, specified as a `NumPorts` vector.

Note `TargetReturn` specifies target returns for portfolios on the efficient frontier. If any `TargetReturn` values are outside the range of returns for efficient portfolios, the

`TargetReturn` is replaced with the minimum or maximum efficient portfolio return, depending upon whether the target return is below or above the range of efficient portfolio returns.

Data Types: `double`

Output Arguments

`pwgt` — Optimal portfolios on efficient frontier with specified target returns

matrix

Optimal portfolios on the efficient frontier with specified target returns from `TargetReturn`, returned as a `NumAssets`-by-`NumPorts` matrix. `pwgt` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`pbuy` — Purchases relative to initial portfolio for optimal portfolios on efficient frontier

matrix

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`pbuy` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`psell` — Sales relative to initial portfolio for optimal portfolios on efficient frontier

matrix

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as a `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`psell` is returned for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio returns.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByReturn(TargetReturn);
```

See Also

`estimateFrontier` | `estimateFrontierByRisk` | `estimateFrontierLimits`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimateFrontierByRisk

Estimate optimal portfolios with targeted portfolio risks

Use the `estimateFrontierByRisk` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate optimal portfolios with targeted portfolio risks.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)
```

Description

`[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)` estimates optimal portfolios with targeted portfolio risks.

Examples

Obtain Portfolios with Targeted Portfolio Risks for a Portfolio Object

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` function accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 14%, and 16%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
 p = Portfolio;
 p = setAssetMoments(p, m, C);
 p = setDefaultConstraints(p);
 pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

 display(pwgt);
```

```
pwgt =

    0.3984    0.2659    0.1416
    0.3064    0.3791    0.4474
    0.0882    0.1010    0.1131
    0.2071    0.2540    0.2979
```

Obtain Portfolios with Targeted Portfolio Risks for a PortfolioCVaR Object

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` function accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 20%, and 30%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioCVaR;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierByRisk(p, [0.12, 0.20, 0.30]);

display(pwgt);
```

```
pwgt =

    0.5363    0.1387         0
    0.2655    0.4990    0.3830
    0.0568    0.1240    0.1461
    0.1413    0.2382    0.4709
```

The function `rng`(*seed*) resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Portfolios with Targeted Portfolio Risks for a PortfolioMAD Object

To obtain efficient portfolios that have targeted portfolio risks, the `estimateFrontierByRisk` function accepts one or more target portfolio risks and obtains efficient portfolios with the specified risks. Assume you have a universe of four assets where you want to obtain efficient portfolios with target portfolio risks of 12%, 20%, and 25%.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

rng(11);

p = PortfolioMAD;
p = simulateNormalScenariosByMoments(p, m, C, 2000);
p = setDefaultConstraints(p);

pwgt = estimateFrontierByRisk(p, [0.12, 0.20, 0.25]);

display(pwgt);

pwgt =

    0.1613    0.0000    0.0000
    0.4777    0.2139    0.0037
    0.1118    0.1381    0.1214
```

```
    0.2492    0.6480    0.8749
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`TargetRisk` — Target values for portfolio risk

vector

Target values for portfolio risk, specified as a `NumPorts` vector.

Note If any `TargetRisk` values are outside the range of risks for efficient portfolios, the target risk is replaced with the minimum or maximum efficient portfolio risk, depending on whether the target risk is below or above the range of efficient portfolio risks.

Data Types: `double`

Output Arguments

`pwgt` — Optimal portfolios on efficient frontier with specified target risks

matrix

Optimal portfolios on the efficient frontier with specified target returns from `TargetRisk`, returned as a `NumAssets`-by-`NumPorts` matrix. `pwgt` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`pbuy` — Purchases relative to initial portfolio for optimal portfolios on efficient frontier

matrix

Purchases relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`pbuy` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`psell` — Sales relative to initial portfolio for optimal portfolios on efficient frontier

matrix

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as a `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`psell` is returned for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to estimate optimal portfolios with targeted portfolio risks.

```
[pwgt, pbuy, psell] = obj.estimateFrontierByRisk(TargetRisk);
```

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierLimits` | `rng`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimateFrontierLimits

Estimate optimal portfolios at endpoints of efficient frontier

Use the `estimateFrontierLimits` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate optimal portfolios at endpoints of efficient frontier.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierLimits(obj)
[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice)
```

Description

`[pwgt,pbuy,psell] = estimateFrontierLimits(obj)` estimates optimal portfolios at endpoints of efficient frontier.

`[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice)` estimates optimal portfolios at endpoints of efficient frontier with an additional option specified for the `Choice` argument.

Examples

Obtain Endpoint Portfolios for a Portfolio Object

Given portfolio `p`, the `estimateFrontierLimits` function obtains the endpoint portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

```
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);

disp(pwgt);
```

```
    0.8891         0
    0.0369         0
    0.0404         0
    0.0336    1.0000
```

Obtain Endpoint Portfolios for a PortfolioCVaR Object

Given portfolio `p`, the `estimateFrontierLimits` function obtains the endpoint portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);


pwgt = estimateFrontierLimits(p);

disp(pwgt);
```

```
    0.8454    0.0000
    0.0599    0.0000
    0.0462    0.0000
    0.0485    1.0000
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Endpoint Portfolios for a PortfolioMAD Object

Given portfolio `p`, the `estimateFrontierLimits` function obtains the endpoint portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierLimits(p);

disp(pwgt);
```

```
    0.8815    0.0000
    0.0431    0.0000
    0.0389    0.0000
    0.0365    1.0000
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`Choice` — Indicator for which portfolios to obtain at extreme ends of efficient frontier

`[]` (default) | character vector with values `[]`, `'Both'`, `'Min'`, `'Max'`

Indicator which portfolios to obtain at the extreme ends of the efficient frontier, specified as a character vector with values `[]`, `'Both'`, `'Min'`, or `'Max'`. `Choice` specifies various actions with default value `[]`. The options for a `Choice` action are:

- `[]` — Compute both minimum-risk and maximum-return portfolios.
- `'Both'` — Compute both minimum-risk and maximum-return portfolios.
- `'Min'` — Compute minimum-risk portfolio only.
- `'Max'` — Compute maximum-return portfolio only.

Data Types: `char`

Output Arguments

`pwgt` — Optimal portfolios at endpoints of efficient frontier
matrix

Optimal portfolios at the endpoints of the efficient frontier `TargetReturn`, returned as a `NumAssets`-by-`NumPorts` matrix. `pwgt` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`pbuy` — Purchases relative to an initial portfolio for optimal portfolios at endpoints of efficient frontier
matrix

Purchases relative to an initial portfolio for optimal portfolios at the endpoints of the efficient frontier, returned as `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`pbuy` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`psell` — Sales relative to an initial portfolio for optimal portfolios at endpoints of efficient frontier
matrix

Sales relative to an initial portfolio for optimal portfolios on the efficient frontier, returned as a `NumAssets`-by-`NumPorts` matrix.

Note If no initial portfolio is specified in `obj.InitPort`, that value is assumed to be `0` such that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`.

`psell` is returned for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to estimate the optimal portfolios at the endpoints of the efficient frontier.

```
[pwgt, pbuy, psell] = obj.estimateFrontierLimits(Choice);
```

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `rng`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimateMaxSharpeRatio

Estimate efficient portfolio to maximize Sharpe ratio for Portfolio object

Use the `estimateMaxSharpeRatio` function with a `Portfolio` object to estimate moments of portfolio returns.

For details on the workflow, see “Portfolio Object Workflow” on page 4-21.

Syntax

```
[pwgt,pbuy,psell] = estimateMaxSharpeRatio(obj)
```

Description

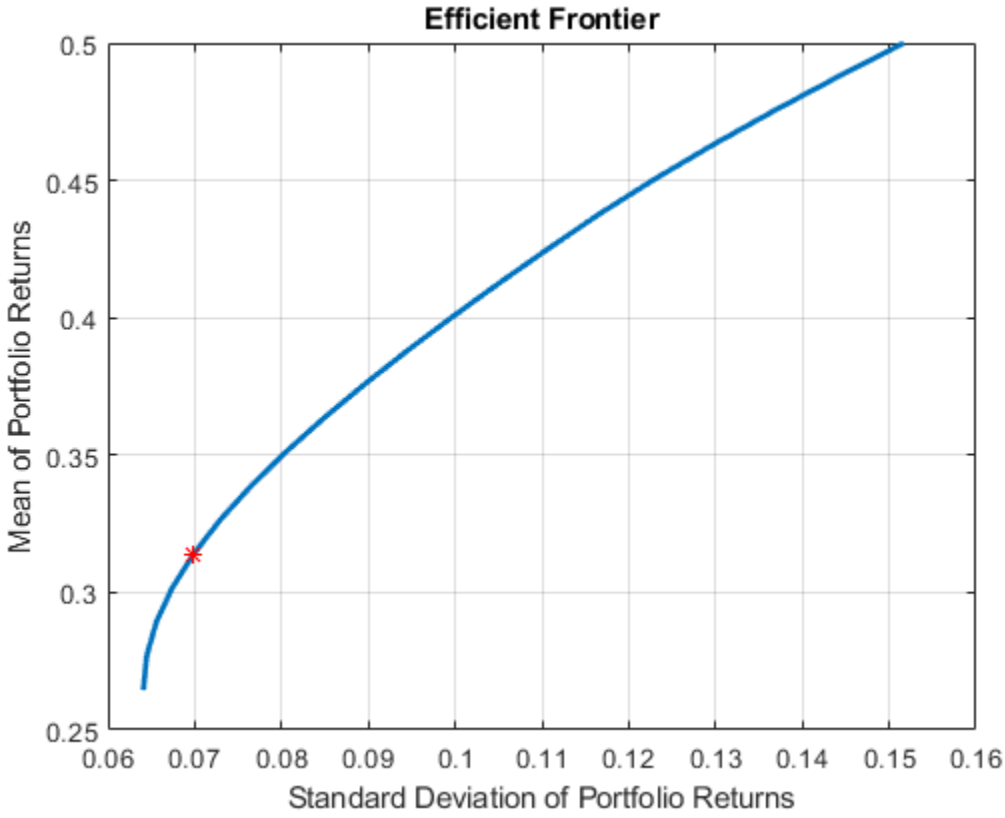
`[pwgt,pbuy,psell] = estimateMaxSharpeRatio(obj)` estimates efficient portfolio to maximize Sharpe ratio for `Portfolio` object.

Examples

Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object

Estimate the efficient portfolio that maximizes the Sharpe ratio.

```
p = Portfolio('AssetMean',[0.3, 0.1, 0.5], 'AssetCovar',...
[0.01, -0.010,  0.004; -0.010,  0.040, -0.002;  0.004, -0.002,  0.023]);
p = setDefaultConstraints(p);
plotFrontier(p, 20);
weights = estimateMaxSharpeRatio(p);
[risk, ret] = estimatePortMoments(p, weights);
hold on
plot(risk,ret,'*r');
```



- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object.

Note The risk-free rate is obtained from the property `RiskFreeRate` in the Portfolio object. If you leave the `RiskFreeRate` unset, it is assumed to be `0`.

For more information on creating a portfolio object, see

- `Portfolio`

Output Arguments

`pwgt` — Portfolio on efficient frontier with maximum Sharpe ratio
vector

Portfolio on the efficient frontier with a maximum Sharpe ratio, returned as a `NumAssets` vector.

`pbuy` — Purchases relative to initial portfolio for portfolio on efficient frontier with maximum Sharpe ratio
vector

Purchases relative to an initial portfolio for a portfolio on the efficient frontier with a maximum Sharpe ratio, returned as a `NumAssets` vector.

`pbuy` is returned for a `Portfolio` input object (`obj`).

`psell` — Sales relative to initial portfolio for portfolio on efficient frontier with maximum Sharpe ratio
vector

Sales relative to an initial portfolio for a portfolio on the efficient frontier with maximum Sharpe ratio, returned as a `NumAssets` vector.

`psell` is returned for a `Portfolio` input object (`obj`).

Definitions

Sharpe Ratio

The Sharpe ratio is the ratio of the difference between the mean of portfolio returns and the risk-free rate divided by the standard deviation of portfolio returns.

The `estimateMaxSharpeRation` function maximizes the Sharpe ratio among portfolios on the efficient frontier.

Tips

You can also use dot notation to estimate an efficient portfolio that maximizes the Sharpe ratio.

```
[pwgt,pbuy,psell] = obj.estimateMaxSharpeRatio;
```

Algorithms

The maximization of the Sharpe ratio is accomplished by a one-dimensional optimization using `fminbnd` to find the portfolio that minimizes the negative of the Sharpe ratio. The `estimateMaxSharpeRation` function takes only a fully qualified Portfolio object as its input and uses all information in the object to solve the problem.

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimatePortMoments

Estimate moments of portfolio returns for Portfolio object

Use the `estimatePortMoments` function with a `Portfolio` object to estimate moments of portfolio returns.

For details on the workflow, see "Portfolio Object Workflow" on page 4-21.

Syntax

```
[prsk,pret] = estimatePortMoments(obj,pwgt)
```

Description

`[prsk,pret] = estimatePortMoments(obj,pwgt)` estimate moments of portfolio returns for a `Portfolio` object.

The estimate of port moments is specific to mean-variance portfolio optimization and computes the mean and standard deviation (which is the square-root of variance) of portfolio returns.

Examples

Identify the Range of Risks and Returns for Efficient Portfolios for a Portfolio Object

Given portfolio `p`, use the `estimatePortMoments` function to show the range of risks and returns for efficient portfolios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
```

```
p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);

[prsk, pret] = estimatePortMoments(p, pwgt);
disp([prsk, pret]);

    0.0769    0.0590
    0.3500    0.1800
```

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object. For more information on creating a portfolio object, see

- `Portfolio`

`pwgt` — Collection of portfolios

matrix

Collection of portfolios, specified as a `NumAssets`-by-`NumPorts` matrix where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Data Types: `double`

Output Arguments

`prsk` — Estimates for standard deviations of portfolio returns for each portfolio in `pwgt`

vector

Estimates for standard deviations of portfolio returns for each portfolio in `pwgt`, returned as a `NumPorts` vector.

`prsk` is returned for a `Portfolio` input object (`obj`).

`pret` — Estimates for means of portfolio returns for each portfolio in `pwgt`
vector

Estimates for means of portfolio returns for each portfolio in `pwgt`, returned as a `NumPorts` vector.

`pret` is returned for a `Portfolio` input object (`obj`).

Tips

You can also use dot notation to estimate the moments of portfolio returns.

```
[prsk, pret] = obj.estimatePortMoments(pwgt);
```

See Also

`estimatePortReturn` | `estimatePortRisk`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimatePortReturn

Estimate mean of portfolio returns

Use the `estimatePortReturn` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate mean of portfolio returns.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
pret = estimatePortReturn(obj,pwgt)
```

Description

`pret = estimatePortReturn(obj,pwgt)` estimates the mean of portfolio returns (as the proxy for portfolio return).

Note Depending on whether costs have been set, the portfolio return is either gross or net portfolio returns.

Examples

Estimate the Mean of Portfolio Returns for a Portfolio Object

Given portfolio `p`, use the `estimatePortReturn` function to estimate the mean of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
```

```
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
pret = estimatePortReturn(p, pwgt);
disp(pret)
```

```
    0.0590
    0.1800
```

Estimate the Mean of Portfolio Returns for a PortfolioCVaR Object

Given portfolio `p`, use the `estimatePortReturn` function to estimate the mean of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierLimits(p);
pret = estimatePortReturn(p, pwgt);
disp(pret)
```

```
    0.0050
    0.0154
```

The function `rng(`*`seed`*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Estimate the Mean of Portfolio Returns for a PortfolioMAD Object

Given portfolio `p`, use the `estimatePortReturn` function to estimate the mean of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierLimits(p);
pret = estimatePortReturn(p, pwgt);
disp(pret)
```

```
    0.0048
    0.0154
```

The function `rng(`*`seed`*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-109
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-129
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101

- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`pwgt` — Collection of portfolios

matrix

Collection of portfolios, specified as a `NumAssets`-by-`NumPorts` matrix, where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Data Types: `double`

Output Arguments

`pret` — Estimates for means of portfolio returns for each portfolio in `pwgt`

vector

Estimates for means of portfolio returns for each portfolio in `pwgt`, returned as a `NumPorts` vector.

`pret` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to estimate the mean of portfolio returns (as the proxy for portfolio return).

```
pret = obj.estimatePortReturn(pwgt);
```

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `rng`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimatePortRisk

Estimate portfolio risk according to risk proxy associated with corresponding object

Use the `estimatePortRisk` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to estimate portfolio risk according to the risk proxy associated with the corresponding object.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
prsk = estimatePortRisk(obj,pwgt)
```

Description

`prsk = estimatePortRisk(obj,pwgt)` estimates portfolio risk according to the risk proxy associated with the corresponding object (`obj`).

Examples

Standard Deviation of Portfolio Returns as the Proxy for Portfolio Risk for a Portfolio Object

Given portfolio `p`, use the `estimatePortRisk` function to show the standard deviation of portfolio returns for each portfolio in `pwgt`.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];

p = Portfolio;
```

```
p = setAssetMoments(p, m, C);
p = setDefaultConstraints(p);
pwgt = estimateFrontierLimits(p);
prsk = estimatePortRisk(p, pwgt);
disp(prsk)
```

```
    0.0769
    0.3500
```

Conditional Value-at-Risk of Portfolio Returns as the Proxy for Portfolio Risk for a PortfolioCVaR Object

Given a portfolio `pwgt`, use the `estimatePortRisk` function to show the conditional value-at-risk (CVaR) of portfolio returns for each portfolio.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierLimits(p);
prsk = estimatePortRisk(p, pwgt);
disp(prsk)
```

```
    0.0407
    0.1911
```

The function `rng(`*`seed`*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Mean-Absolute Deviation Returns as the Proxy for Portfolio Risk for a PortfolioMAD Object

Given a portfolio `pwgt`, use the `estimatePortRisk` function to show the mean-absolute deviation of portfolio returns for each portfolio.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);


pwgt = estimateFrontierLimits(p);
prsk = estimatePortRisk(p, pwgt);
disp(prsk)

    0.0177
    0.0809
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119

- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`pwgt` — Collection of portfolios

matrix

Collection of portfolios, specified as a `NumAssets`-by-`NumPorts` matrix, where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Data Types: `double`

Output Arguments

`prsk` — Estimates for portfolio risk according to the risk proxy associated with the corresponding object (`obj`) for each portfolio in `pwgt`

vector

Estimates for portfolio risk according to the risk proxy associated with the corresponding object (`obj`) for each portfolio in `pwgt`, returned as a `NumPorts` vector.

`prsk` is returned for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to estimate portfolio risk according to the risk proxy associated with the corresponding object (`obj`).

```
prsk = obj.estimatePortRisk(pwgt);
```

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortStd` | `estimatePortVaR` | `rng`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

estimatePortStd

Estimate standard deviation of portfolio returns

Use the `estimatePortStd` function with a `PortfolioCVaR` or `PortfolioMAD` objects to estimate standard deviation of portfolio returns.

For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-20 and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
pstd = estimatePortStd(obj,pwgt)
```

Description

`pstd = estimatePortStd(obj,pwgt)` estimate standard deviation of portfolio returns for `PortfolioCVaR` or `PortfolioMAD` objects.

Examples

Estimate Standard Deviations for Portfolio Returns for a PortfolioCVaR Object

Given a portfolio `pwgt`, use the `estimatePortStd` function to show the standard deviation of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);
```

```
AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierLimits(p);

pstd = estimatePortStd(p, pwgt);
disp(pstd)

    0.0223
    0.1010
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Estimate Standard Deviations for Portfolio Returns for a PortfolioMAD Object

Given a portfolio `pwgt`, use the `estimatePortStd` function to show the standard deviation of portfolio returns.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

pwgt = estimateFrontierLimits(p);
```

```
pstd = estimatePortStd(p, pwgt);
disp(pstd)
```

```
    0.0222
    0.1010
```

The function `rng(`*`seed`*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` or `PortfolioMAD`object.

For more information on creating a `PortfolioCVaR` or `PortfolioMAD` object, see

- `PortfolioCVaR`
- `PortfolioMAD`

`pwgt` — Collection of portfolios

matrix

Collection of portfolios, specified as a `NumAssets`-by-`NumPorts` matrix, where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Data Types: `double`

Output Arguments

`pstd` — Estimates for standard deviations of portfolio returns for each portfolio in `pwgt`

vector

Estimates for standard deviations of portfolio returns for each portfolio in `pwgt`, returned as a `NumPorts` vector.

Tips

You can also use dot notation to estimate the standard deviation of portfolio returns.

```
pstd = obj.estimatePortStd(pwgt);
```

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortReturn` | `estimatePortVaR` | `rng`

Topics

“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
“Portfolio Optimization Theory” on page 4-3

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

estimatePortVaR

Estimate value-at-risk for PortfolioCVaR object

Use the `estimatePortVaR` function with a `PortfolioCVaR` object to estimate value-at-risk.

For details on the workflow, see "PortfolioCVaR Object Workflow" on page 5-20.

Syntax

```
pvar = estimatePortVaR(obj,pwgt)
```

Description

`pvar = estimatePortVaR(obj,pwgt)` estimates value-at-risk for a `PortfolioCVaR` object where the probability level used is from the `PortfolioCVaR` property `ProbabilityLevel`.

Examples

Estimate Value-at-Risk for a PortfolioCVaR Object

Given a portfolio `pwgt`, use the `estimatePortVaR` function to estimate the value-at-risk of portfolio.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

rng(11);
```

```
AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

pwgt = estimateFrontierLimits(p);

pvar = estimatePortVaR(p, pwgt);
disp(pvar)

    0.0314
    0.1483
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` object.

For more information on creating a `PortfolioCVaR` object, see

- `PortfolioCVaR`

`pwgt` — Collection of portfolios

matrix

Collection of portfolios, specified as a `NumAssets`-by-`NumPorts` matrix, where `NumAssets` is the number of assets in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

Data Types: `double`

Output Arguments

`pvar` — Estimates for value-at-risk of portfolio returns for each portfolio in `pwgt`

vector

Estimates for value-at-risk of portfolio returns for each portfolio in `pwgt`, returned as a `NumPorts` vector.

Tips

You can also use dot notation to estimate the value-at-risk of PortfolioCVaR object.

```
pvar = obj.estimatePortVaR(pwgt);
```

See Also

`estimatePortStd` | `rng` | `setProbabilityLevel`

Topics

"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
"Conditional Value-at-Risk" on page 5-6

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

estimateScenarioMoments

Estimate mean and covariance of asset return scenarios

Use the `estimateScenarioMoments` function with a `PortfolioCVaR` or `PortfolioMAD` objects to estimate mean and covariance of asset return scenarios.

For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[ScenarioMean,ScenarioCovar] = estimateScenarioMoments(obj)
```

Description

`[ScenarioMean,ScenarioCovar] = estimateScenarioMoments(obj)` estimates mean and covariance of asset return scenarios for `PortfolioCVaR` or `PortfolioMAD` objects.

Examples

Estimate Mean and Covariance of Asset Return Scenarios for a PortfolioCVaR Object

Given PortfolioCVaR object `p`, use the `estimatePortRisk` function to estimate mean and covariance of asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
```

```
rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(p)

ScenarioMean =

    0.0039
    0.0082
    0.0102
    0.0154


ScenarioCovar =

    0.0005    0.0003    0.0001   -0.0001
    0.0003    0.0024    0.0017    0.0010
    0.0001    0.0017    0.0048    0.0028
   -0.0001    0.0010    0.0028    0.0102
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Estimate Mean and Covariance of Asset Return Scenarios for a PortfolioMAD Object

Given PortfolioMAD object `p`, use the `estimatePortRisk` function to estimate mean and covariance of asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
```

```
C = C/12;

rng(11);

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

[ScenarioMean, ScenarioCovar] = estimateScenarioMoments(p)

ScenarioMean =

    0.0039
    0.0082
    0.0102
    0.0154


ScenarioCovar =

    0.0005    0.0003    0.0001   -0.0001
    0.0003    0.0024    0.0017    0.0010
    0.0001    0.0017    0.0048    0.0028
   -0.0001    0.0010    0.0028    0.0102
```

The function `rng`(*seed*) resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` or `PortfolioMAD` object.

For more information on creating a `PortfolioCVaR` or `PortfolioMAD` object, see

- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`ScenarioMean` — Estimate for mean of scenarios

`[]` (default) | vector

Estimate for mean of scenarios, returned as a `NumPorts` vector or `[]`.

Note If no scenarios are associated with the specified object, both `ScenarioMean` and `ScenarioCovar` are set to empty `[]`.

`ScenarioCovar` — Estimate for covariance of scenarios

`[]` (default) | matrix

Estimate for covariance of scenarios, returned as a `NumAssets`-by-`NumAssets` matrix or `[]`.

Note If no scenarios are associated with the specified object, both `ScenarioMean` and `ScenarioCovar` are set to empty `[]`.

Tips

You can also use dot notation to estimate the mean and covariance of asset return scenarios for a portfolio.

```
[ScenarioMean, ScenarioCovar] = obj.estimateScenarioMoments
```

See Also

`estimatePortRisk` | `rng` | `setScenarios` | `simulateNormalScenariosByMoments`

Topics

“Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

ewstats

Expected return and covariance from return time series

Syntax

```
[ExpReturn,ExpCovariance,NumEffObs] = ewstats(RetSeries)
[ExpReturn,ExpCovariance,NumEffObs] = ewstats(___,DecayFactor,
WindowLength)
```

Description

`[ExpReturn,ExpCovariance,NumEffObs] = ewstats(RetSeries)` computes estimated expected returns (`ExpReturn`), estimated covariance matrix (`ExpCovariance`), and the number of effective observations (`NumEffObs`). These outputs are maximum likelihood estimates which are biased.

`[ExpReturn,ExpCovariance,NumEffObs] = ewstats(___,DecayFactor, WindowLength)` adds optional input arguments for `DecayFactor` and `WindowLength`.

Examples

Compute Estimated Expected Returns and Estimated Covariance Matrix

This example shows how to compute the estimated expected returns and the estimated covariance matrix.

```
RetSeries = [ 0.24 0.08
              0.15 0.13
              0.27 0.06
              0.14 0.13 ];

DecayFactor = 0.98;

[ExpReturn, ExpCovariance] = ewstats(RetSeries, DecayFactor)
```

```
ExpReturn =

    0.1995    0.1002


ExpCovariance =

    0.0032   -0.0017
   -0.0017    0.0010
```

Input Arguments

`RetSeries` — Return series

matrix

Return series, specified the 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.

Data Types: `double`

`DecayFactor` — Controls how much less each observation is weighted than its successor

`1` (default) | numeric

(Optional) Controls how much less each observation is weighted than its successor, specified as a numeric value. The kth observation back in time has weight $\texttt{DecayFactor}^k$. `DecayFactor` must lie in the range: `0 < DecayFactor <= 1`.

The default value of `1` is the equally weighted linear moving average model (BIS).

Data Types: `double`

`WindowLength` — Number of recent observations in computation

`NUMOBS` (default) | numeric

(Optional) Number of recent observations in the computation, specified as a numeric value.

Data Types: `double`

Output Arguments

`ExpReturn` — Estimated expected returns

vector

Estimated expected returns, returned as a `1`-by-`NASSETS` vector.

`ExpCovariance` — Estimated covariance matrix

matrix

Estimated covariance matrix, returned as a `NASSETS`-by-`NASSETS` matrix.

The standard deviations of the asset return processes are defined as

```
STDVec = sqrt(diag(ExpCovariance))
```

The correlation matrix is

```
CorrMat = ExpCovariance./( STDVec*STDVec' )
```

`NumEffObs` — Number of effective observations

numeric

`NumEffObs` is the number of effective observations where

$$NumEffObs = \frac{1 - DecayFactor^{WindowLength}}{1 - DecayFactor}$$

A smaller `DecayFactor` or `WindowLength` emphasizes recent data more strongly but uses less of the available data set.

Algorithms

For a return series $r(1),\ldots,r(n)$, where (n) is the most recent observation, and w is the decay factor, the expected returns (`ExpReturn`) are calculated by

$$E(r) = \frac{(r(n) + wr(n-1) + w^2 r(n-2) + \ldots + w^{n-1} r(1))}{NumEffObs}$$

where the number of effective observations `NumEffObs` is defined as

$$NumEffObs = 1 + w + w^2 + \ldots + w^{n-1} = \frac{1 - w^n}{1 - w}$$

$E(r)$ is the weighed average of $r(n),\ldots,r(1)$. The unnormalized weights are w, w^2, …, $w^{(n-1)}$. The unnormalized weights do not sum up to `1`, so `NumEffObs` rescales the unnormalized weights. After rescaling, the normalized weights (which sum up to `1`) are used for averaging. When w = `1`, then `NumEffObs` = n, which is the number of observations. When w < `1`, `NumEffObs` is still interpreted as the sample size, but it is less than n due to the down-weight on the observations of the remote past.

Note There is no relationship between `ewstats` function and the RiskMetrics® approach for determing the expected return and covariance from a return time series.

See Also

`cov` | `cov2corr` | `mean`

Topics

"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

exp

Exponential values

Syntax

```
newfts = exp(tsobj)
```

Description

`newfts = exp(tsobj)` calculates the natural exponential (base e) of all the data in the data series of the financial time series object `tsobj` and returns the result in the object `newfts`.

See Also

`log` | `log10` | `log2`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

extfield

Data series extraction

Syntax

```
ftse = extfield(tsobj,fieldnames)
```

Arguments

`tsobj`	Financial time series object
`fieldnames`	Data series to be extracted. A cell array of character vectors if a list of data series names (`fieldnames`) is supplied. A character vector 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`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

fanplot

Plot combined historical and forecast data to visualize possible outcomes

Syntax

```
fanplot(historical,forecast)
fanplot(___,Name,Value)

h = fanplot(historical,forecast)
h = fanplot(___,Name,Value)
```

Description

`fanplot(historical,forecast)` generates a fan chart. In time series analysis, a fan chart is a chart that joins a simple line chart for observed past data with ranges for possible values of future data. The historical data and possible future data are joined with a line showing a central estimate or most likely value for the future outcomes.

`fanplot` supports three plotting scenarios:

- Matching — This scenario occurs when the time period perfectly matches for `historical` and `forecast` data.
- Backtest — This scenario occurs when there are overlaps between `historical` and `forecast` data.
- Gap — This scenario occurs when there are `NaN` values in the `historical` or `forecast` data.

`fanplot(___,Name,Value)` generates a fan chart using optional name-value pair arguments.

`h = fanplot(historical,forecast)` generates a fan chart and returns the figure handle `h`. In time series analysis, a fan chart is a chart that joins a simple line chart for observed past data with ranges for possible values of future data. The historical data and possible future data are joined with a line showing a central estimate or most likely value for the future outcomes.

`fanplot` supports three plotting scenarios:

- Matching — This scenario occurs when the time period perfectly matches for `historical` and `forecast` data.
- Backtest — This scenario occurs when there are overlaps between `historical` and `forecast` data.
- Gap — This scenario occurs when there are `NaN` values in the `historical` or `forecast` data.

`h = fanplot(___,Name,Value)` generates a fan chart and returns the figure handle `h` using optional name-value pair arguments.

Examples

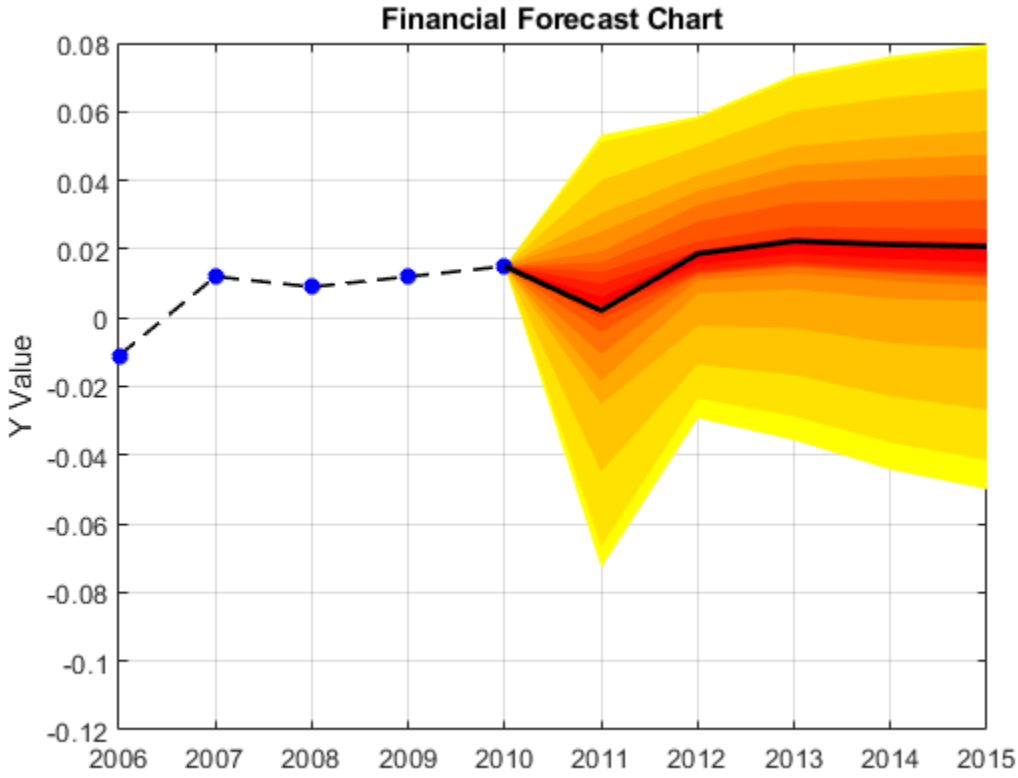
Create a Fan Plot Using Cell Array Data

Define the data inputs for `historical` as a 5-by-2 cell array and `forecast` as a 5-by-21 cell array with 20 observations.

```
historical =  {[2006]    [-0.0110]
               [2007]    [ 0.0120]
               [2008]    [ 0.0090]
               [2009]    [ 0.0120]
               [2010]    [ 0.0150]};

forecast =    {[2011]  [0.0203]      [-0.0155]    [0.0311]     [-0.0026]     [0.0035]
                       [0.0533]      [0.0139]     [0.0037]     [-0.0727]     [-0.0291]
               [2012]  [0.0430]      [-0.0094]    [0.0587]     [ 0.0095]     [0.0185]
                       [0.0141]      [0.0337]     [0.0187]     [0.0132]      [-0.0292]
               [2013]  [0.0518]      [-0.0116]    [0.0708]     [0.0112]      [0.0221]
                       [0.0168]      [0.0405]     [0.0224]     [0.0157]      [-0.0356]
               [2014]  [0.0546]      [-0.0171]    [0.0762]     [0.0088]      [0.0210]
                       [0.0151]      [0.0419]     [0.0214]     [0.0139]      [-0.0442]
               [2015]  [0.0565]      [-0.0207]    [0.0797]     [0.0072]      [0.0203]
                       [0.0139]      [0.0428]     [0.0207]     [0.0126]      [-0.0499]
```

Generate the fan plot.

```
fanplot (historical, forecast);
```



The dotted points are the historical lines and the filled lines indicate the mean for the forecasts. This fanplot represents a matching scenario where the time period perfectly matches for the historical and forcast data.

Create a Fan Plot Using Matrix Data

Define the data inputs for `historical` as a 5-by-2 matrix and `forecast` as a 5-by-21 matrix with 20 observations.

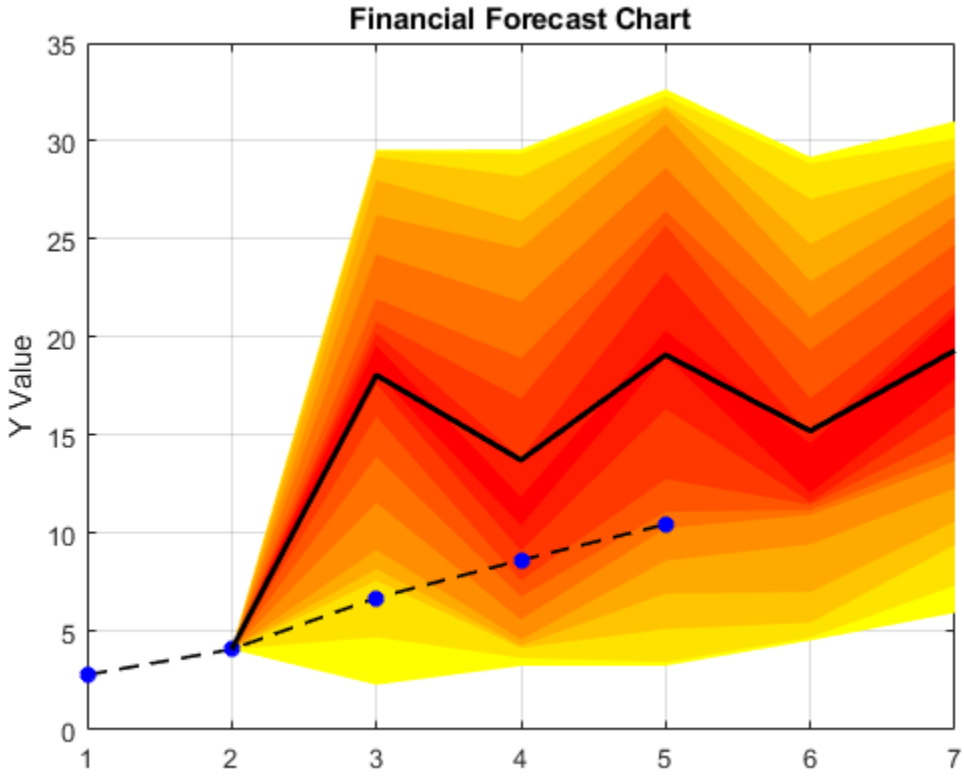
```
historical =  [ 1.0000    2.8046 ;
                2.0000    4.1040 ;
```

```
                3.0000    6.7292 ;
                4.0000    8.6486 ;
                5.0000   10.4747 ];



forecast  =   [ 3.0000   28.9874   18.3958   19.6376   29.5627    8.3462    7.1502   25
                         20.8557   27.0691   23.0803   20.7885   18.0205   17.2294   10

                4.0000    4.8933   27.2659    7.2206   24.4703   10.5895   15.0212   29
                          8.0007   18.7114   19.1691   24.5963    4.2835    4.0676    3

                5.0000   20.9732   19.7069   11.6862   25.7018   31.8940    7.2664   19
                         31.7996    3.6419    3.2695   27.1422   10.5487   32.6529   18

                6.0000   11.0069   29.1965    4.5551   20.2627   10.9209   15.2675   28
                         23.9532   18.4804   25.5484    4.8747    8.0036   11.5329   11

                7.0000    5.9699   11.1486   26.0449   13.4619   21.1196   28.8068   26
                         21.2390   29.2396   18.4828   28.3945   21.9342   14.4642   17
```

Generate the fan plot and return the figure handle.

```
h = fanplot(historical, forecast)
```

```
h =
  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [360 502 560 420]
       Units: 'pixels'

  Show all properties
```



The dotted points are the historical lines and the filled lines indicate the mean for the forecasts. This fanplot represents a backtest scenario where there is an overlap between the historical and forecast data.

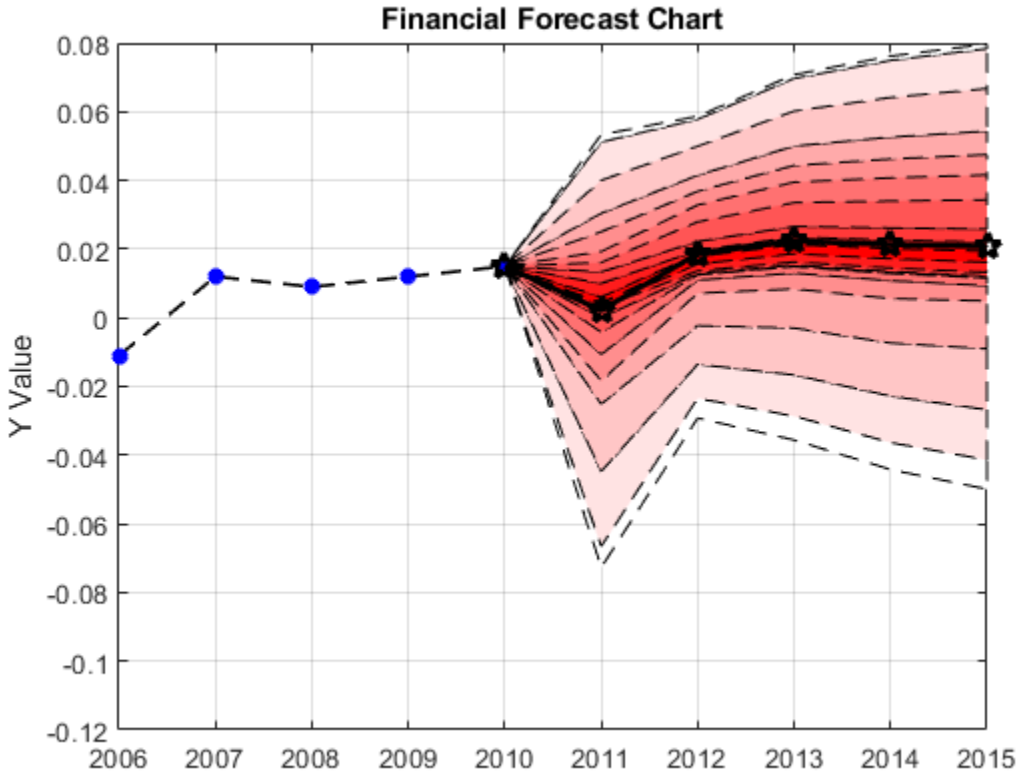
Create a Fan Plot Using Cell Array Data and Customize the Plot With Name-Value Pair Arguments

Define the data inputs for `historical` as a 5-by-2 cell array and `forecast` as a 5-by-21 cell array with 20 observations.

```
historical =  {[2006]    [-0.0110]
               [2007]    [ 0.0120]
               [2008]    [ 0.0090]
               [2009]    [ 0.0120]
               [2010]    [ 0.0150]};

forecast =    {[2011]  [0.0203]      [-0.0155]    [0.0311]     [-0.0026]     [0.0035]
                       [0.0533]      [0.0139]     [0.0037]     [-0.0727]     [-0.0291]
               [2012]  [0.0430]      [-0.0094]    [0.0587]     [ 0.0095]     [0.0185]
                       [0.0141]      [0.0337]     [0.0187]     [0.0132]      [-0.0292]
               [2013]  [0.0518]      [-0.0116]    [0.0708]     [0.0112]      [0.0221]
                       [0.0168]      [0.0405]     [0.0224]     [0.0157]      [-0.0356]
               [2014]  [0.0546]      [-0.0171]    [0.0762]     [0.0088]      [0.0210]
                       [0.0151]      [0.0419]     [0.0214]     [0.0139]      [-0.0442]
               [2015]  [0.0565]      [-0.0207]    [0.0797]     [0.0072]      [0.0203]
                       [0.0139]      [0.0428]     [0.0207]     [0.0126]      [-0.0499]
```

Generate the fan plot using name-value pair arguments to customize the presentation.

```
fanplot(historical,forecast,'FanFaceColor',[1 1 1;1 0 0],'FanLineStyle','--','ForecastM
```



Create a Fan Plot Using Table Data

Create table of historical dates and data.

```
historicalDates = datetime(2006:2010,1,1)';
historicalData = [-0.0110;0.0120;0.0090;0.0120;0.0150];
historical = table(historicalDates,historicalData,'VariableNames',{'Dates','Data'});
```

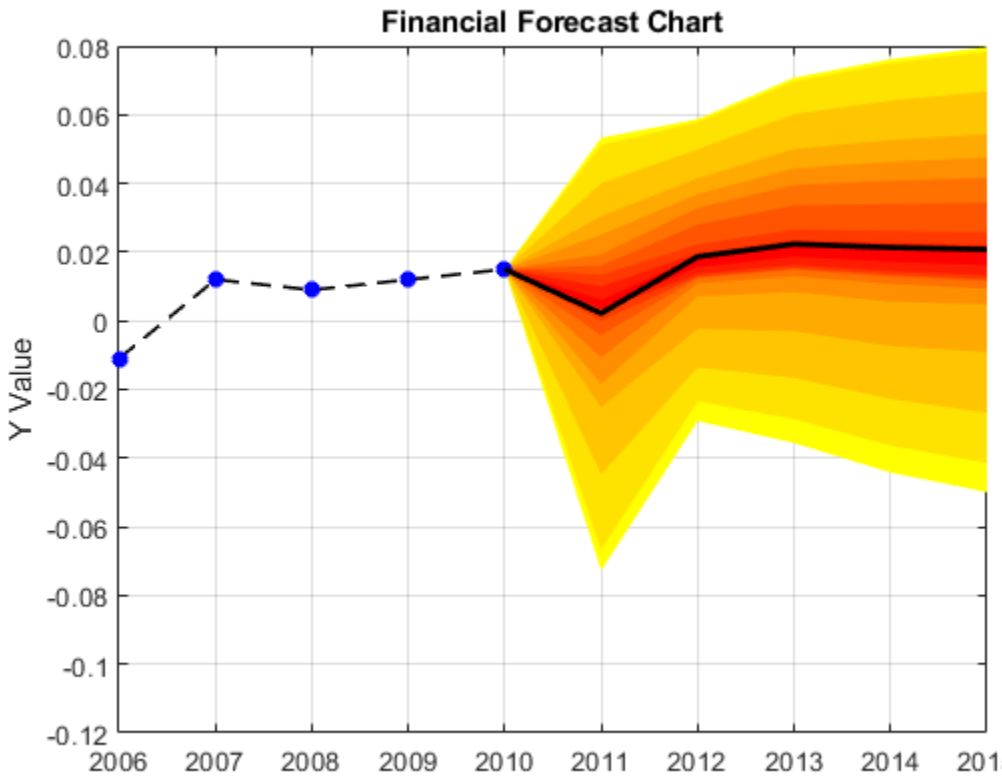
Create table of forecast dates and data.

```
forecastDates = datetime(2011:2015,1,1)';
forecastData =      [0.0203      -0.0155    0.0311     -0.0026     0.0035    0.0049
```

```
    0.0533    0.0139    0.0037   -0.0727   -0.0291   -0.0058
    0.0430   -0.0094    0.0587    0.0095    0.0185    0.0205
    0.0141    0.0337    0.0187    0.0132   -0.0292    0.0048
    0.0518   -0.0116    0.0708    0.0112    0.0221    0.0246
    0.0168    0.0405    0.0224    0.0157   -0.0356    0.0056
    0.0546   -0.0171    0.0762    0.0088    0.0210    0.0239
    0.0151    0.0419    0.0214    0.0139   -0.0442    0.0024
    0.0565   -0.0207    0.0797    0.0072    0.0203    0.0234
    0.0139    0.0428    0.0207    0.0126   -0.0499    0.0026
forecast = [table(forecastDates,'VariableName',{'Dates'}),array2table(forecastData)];
```

Plot the data using `fanplot`.

```
fanplot(historical,forecast);
```



- “Charting Financial Data” on page 2-14

Input Arguments

`historical` — Historical dates and data

matrix | cell array | table

Historical dates and data, specified as an `N`-by-`2` matrix, cell array, or table, where the first column is the date, and the second column is the data associated for that date. `N` indicates the number of dates. By using the cell array format for the input, you can make the first column datetime and produce the same plot as would serial date numbers or date character vectors. For example:

```
historical(:,1) = num2cell(datetime(2006:2010,1,1));
forecast(:,1) = num2cell(datetime(2011:2015,1,1));
fanplot (historical, forecast);
```

Data Types: `cell` | `double` | `table`

`forecast` — Forecast dates and data

matrix | cell array of character vectors | table

Forecast dates and data, specified as an `N`-by-`M` matrix, cell array, or table, where the first column is the date, and the second to the last columns are the data observations. `N` indicates the number of the dates and (`M` – `1`) is the number for data observations. By using the cell array format for the input, you can make the first column datetime and produce the same plot as would serial date numbers or date character vectors. For example:

```
historical(:,1) = num2cell(datetime(2006:2010,1,1));
forecast(:,1) = num2cell(datetime(2011:2015,1,1));
fanplot (historical, forecast);
```

Data Types: `cell` | `double` | `table`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: fanplot(historical,forecast,'NumQuantiles', 14,'FanLineColor','blue','HistoricalLineWidth', 1.8,'ForecastLineColor','red')

NumQuantiles — Number of quantiles to display
20 (default) | positive integer

Number of quantiles to display in fan chart, specified as a positive integer.

Data Types: double

FanLineStyle — Style of the lines separating fans
'none' (default) | character vector

Style of the lines separating fans, specified as a character vector. For more information on supported character vectors for line styles, see Primitive Line.

Data Types: char

FanLineColor — Color of lines separating fans
'black' (default) | character vector for color or RGB triplet

Color of lines separating fans, specified as a character vector for color or an RGB triplet. For more information on supported color character vectors, see Primitive Line.

Data Types: double | char

FanFaceColor — Color of each fan
[1 1 0;1 0 0] (yellow to red) (default) | matrix

Color of each fan, specified as an N-by-3 matrix controlling the color of each fan, where each row is an RGB triplet. There are three possible values of N:

- When N = NumQuantiles, the color of each fan is specified by the corresponding row in the matrix.
- When N = ceil(NumQuantiles/2), the specified colors represent the bottom half of the fans. The colors of the top half are determined by reversing the order of these colors. For more information, see ceil.
- When N = 2, the colors in the bottom half of the fan are a linear interpolation between the two specified colors. The pattern is reversed for the top half.

Data Types: `double`

`HistoricalMarker` — Marker symbol of historical line
`'o'` (default) | character vector

Marker symbol of historical line, specified as a character vector. For more information on supported character vectors for markers, see Primitive Line.

Data Types: `char`

`HistoricalMarkerSize` — Marker size of historical line
`5` (default) | positive value in point units

Marker size of historical line, specified as a positive value in point units.

Data Types: `double` | `char`

`HistoricalMarkerFaceColor` — Marker fill color of historical line
`'blue'` (default) | character vector with a value of `'none'`, `'auto'`, color identifier, or RGB triplet

Marker fill color of historical line, specified as a character vector with a value of `'none'`, `'auto'`, a character vector for color, or an RGB triplet. For more information on supported character vectors for color, see Primitive Line.

Data Types: `double` | `char`

`HistoricalMarkerEdgeColor` — Marker outline color of historical line
`'blue'` (default) | character vector with a value of `'none'`, `'auto'`, color identifier, or RGB triplet

Marker outline color of historical line, specified as a character vector with a value of `'none'`, `'auto'`, a character vector for color, or an RGB triplet. For more information on supported character vectors for color, see Primitive Line.

Data Types: `double` | `char`

`HistoricalLineColor` — Color of historical line
`'black'` (default) | character vector with a value of `'none'`, color identifier, or RGB triplet

Color of historical line, specified as a character vector with a value of `'none'`, a character vector for color, or an RGB triplet. For more information on supported character vectors for color, see Primitive Line.

Data Types: `double` | `char`

`HistoricalLineStyle` — Style of historical line
`'--'` (default) | character vector

Style of historical line, specified as a character vector. For more information on supported character vectors for line styles, see Primitive Line.

Data Types: `char`

`HistoricalLineWidth` — Width of historical line
`1.5` (default) | positive value in point units

Width of historical line, specified as a positive value in point units.

Data Types: `double`

`ForcecastMarker` — Marker symbol of forecast line
`'none'` (default) | character vector

Marker symbol of forecast line, specified as a character vector. For more information on supported character vectors for marker symbols, see Primitive Line.

Data Types: `char`

`ForecastMarkerSize` — Marker size of forecast line
`5` (default) | positive value in point units

Marker size of forecast line, specified as a positive value in point units.

Data Types: `double`

`ForecastMarkerFaceColor` — Marker fill color of forecast line
`'none'` (default) | character vector with a value of `'none'`, `'auto'`, color identifier, or RGB triplet

Marker fill color of forecast line, specified as a character vector with a value of `'none'`, `'auto'`, a character vector for color, or an RGB triplet. For more information on supported character vectors for color, see Primitive Line.

Data Types: `double` | `char`

`ForecastMarkerEdgeColor` — Marker outline color of forecast line
`'auto'` (default) | character vector with a value of `'none'`, `'auto'`, color identifier, or RGB triplet

Marker outline color of forecast line, specified as a character vector with a value of `'none'`, `'auto'`, character vector for color, or an RGB triplet. For more information on supported character vectors for color, see Primitive Line.

Data Types: `double` | `char`

`ForecastLineColor` — Color of forecast line
`'black'` (default) | character vector with a value of `'none'`, color identifier, or RGB triplet

Color of forecast line, specified as a character vector with a value of `'none'`, a character vector for color, or an RGB triplet. For more information on supported character vectors for color, see Primitive Line.

Data Types: `double` | `char`

`ForecastLineStyle` — Style of forecast line
`'-'` (default) | character vector

Style of forecast line, specified as a character vector. For more information on supported character vectors for line styles, see Primitive Line.

Data Types: `char`

`ForecastLineWidth` — Width of forecast line
`2` (default) | positive value in point units

Width of forecast line, specified as a positive value in point units.

Data Types: `double`

Output Arguments

`h` — Figure handle for fanplot
handle object

Figure handle for the fanplot, returned as handle object.

See Also

`bolling` | `candle` | `ceil` | `datetime` | `highlow` | `linebreak` | `movavg` | `pointfig` | `priceandvol` | `renko` | `volarea`

Topics

"Charting Financial Data" on page 2-14

Introduced in R2014b

fbusdate

First business date of month

Syntax

```
Date = fbusdate(Year,Month)
Date = fbusdate(___,Holiday,Weekend,outputType)
```

Description

`Date = fbusdate(Year,Month)` returns the serial date number for the first business date of the given year and month.

`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 character vectors.

`Date = fbusdate(___,Holiday,Weekend,outputType)` returns the serial date number for the first business date of the given year and month using optional input arguments. The optional argument `Holiday` specifies nontrading days.

If neither `Holiday` nor `outputType` are specified, `Date` is returned as a serial date number. If `Holiday` is specified, but not `outputType`, then the type of the holiday variable controls the type of date. If `Holiday` is a serial date number or date character vector, then `Date` is returned as a serial date number.

Examples

Return a Serial Date Number for the First Business Date

This example shows how to return serial date numbers for the first business date, given year and month.

```
Date = fbusdate(2001, 11)

Date = 731156

datestr(Date)

ans =
'01-Nov-2001'


Year = [2002 2003 2004];
Date = fbusdate(Year, 11)

Date =

      731521      731888      732252


datestr(Date)

ans = 3x11 char array
    '01-Nov-2002'
    '03-Nov-2003'
    '01-Nov-2004'
```

Return a Serial Date Number for the First Business Date Using the Weekend Argument

This example shows how to return serial date numbers for the first business date, given year and month, and also indicate that Saturday is a business day by setting the `Weekend` argument. March 1, 2003, is a Saturday. Use `fbusdate` to check that this Saturday is actually the first business day of the month.

```
Weekend = [1 0 0 0 0 0 0];
Date = datestr(fbusdate(2003, 3, [], Weekend))

Date =
'01-Mar-2003'
```

Return a datetime array for `Date` for the First Business Date Using the `outputType` Argument

This example shows how to return a datetime array for `Date` using an `outputType` of `'datetime'`.

```
Date = fbusdate(2001, 11,[],[],'datetime')
```

```
Date = datetime
   01-Nov-2001
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Year` — Year to determine occurrence of weekday

4-digit integer | vector of 4-digit integers

Year to determine occurrence of weekday, specified as a 4-digit integer or vector of 4-digit integers.

Data Types: `single` | `double`

`Month` — Month to determine occurrence of weekday

integer with value `1` through `12` | vector of integers with values `1` through `12`

Month to determine occurrence of weekday, specified as an integer or vector of integers with values `1` through `12`.

Data Types: `single` | `double`

`Holiday` — Holidays and nontrading-day dates

non-trading day vector is determined by the routine `holidays` (default) | serial date number | date character vector | datetime array

Holidays and nontrading-day dates, specified as vector.

All dates in `Holiday` must be the same format: either serial date numbers, or date character vectors, or datetime arrays. (Using serial date numbers improves performance.) The `holidays` function supplies the default vector.

If `Holiday` is a datetime array, then `Date` is returned as a datetime array. If `outputType` is specified, then its value determines the output type of `Date`. This overrides any influence of `Holiday`.

Data Types: `double` | `char` | `datetime`

`Weekend` — Weekend days
`[1 0 0 0 0 0 1]` (Saturday and Sunday form the weekend) (default) | vector of length 7, containing `0` and `1`, where `1` indicates weekend days

Weekend days, specified as a vector of length 7, containing `0` and `1`, where `1` indicates weekend days and the first element of this vector corresponds to Sunday.

Data Types: `double`

`outputType` — Year to determine days
`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

A character vector specified as either `'datenum'` or `'datetime'`. The output `Date` is in serial date format if `'datenum'` is specified, or datetime format if `'datetime'` is specified. By default the output `Date` is in serial date format, or match the format of `Holiday`, if specified.

Data Types: `char`

Output Arguments

`Date` — Date for the first business date of given year and month
serial date number | date character vector | datetime array

Date for the first business date of a given year and month, returned as a serial date number, date character vector, or datetime array.

If neither `Holiday` nor `outputType` are specified, `Date` is returned as a serial date number. If `Holiday` is specified, but not `outputType`, then the type of the holiday variable controls the type of date:

- If `Holiday` is a serial date number or date character vector, then `Date` is returned as a serial date number
- If `Holiday` is a datetime array, then `Date` is returned as a datetime array.

.

See Also

`busdate` | `datetime` | `eomdate` | `holidays` | `isbusday` | `lbusdate`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

fetch

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 returns:

```
ans =

    desc:  My first FINTS
    freq:  Daily (1)

    'dates:  (4)'    'times:  (4)'    'Data1:  (4)'
    '01-Jan-2001'    '11:00'          [          1]
    '     "     '    '12:00'          [          2]
    '03-Jan-2001'    '11:00'          [          5]
    '     "     '    '12:00'          [          6]
```

Example 2. Create a financial time series object with time intervals of less than 1 hour:

```
dates2 = ['01-Jan-2001';'01-Jan-2001'; '01-Jan-2001';...
'02-Jan-2001'; '02-Jan-2001';'02-Jan-2001'];
times2 = ['11:00';'11:05';'11:06';'12:00';'12:05';'12:06'];
dates_times2 = cellstr([dates2, repmat(' ',size(dates2,1),1),...
times2]);
myFts2 = fints(dates_times2,(1:6)',{'Data1'},1,'My second
FINTS')
```

```
myFts2 =

    desc:  My second FINTS
    freq:  Daily (1)

    'dates:  (6)'    'times:  (6)'    'Data1:  (6)'
    '01-Jan-2001'    '11:00'          [          1]
    '     "     '    '11:05'          [          2]
    '     "     '    '11:06'          [          3]
    '02-Jan-2001'    '12:00'          [          4]
    '     "     '    '12:05'          [          5]
    '     "     '    '12:06'          [          6]
```

Use `fetch` to extract data from this time series object at 5-minute intervals for each day starting at 11:00 o'clock on January 1, 2001.

```
fetch(myFts2,'01-Jan-2001',[],'02-Jan-2001',[],1,'d',5)
```

```
    desc:  My second FINTS
    freq:  Daily (1)

    'dates:  (4)'    'times:  (4)'    'Data1:  (4)'
    '01-Jan-2001'    '11:00'          [          1]
    '     "     '    '11:05'          [          2]
    '02-Jan-2001'    '12:00'          [          4]
    '     "     '    '12:05'          [          5]
```

You can use this version of `fetch` to extract data at specific times. For example, to fetch data only at 11:06 and 12:06 from `myFts2`, enter

```
fetch(myFts2,'01-Jan-2001',[],'02-Jan-2001',[],1,'d',...
{'11:06';'12:06'})

ans =

    desc:  My second FINTS
    freq:  Daily (1)

    'dates:  (2)'    'times:  (2)'    'Data1:  (2)'
    '01-Jan-2001'    '11:06'          [          3]
    '02-Jan-2001'    '12:06'          [          6]
```

See Also

`extfield` | `ftsbound` | `getfield` | `subsref`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

fieldnames

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 character vectors, 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`

Topics

“Using Time Series to Predict Equity Return” on page 12-25
“What Is the Financial Time Series App?” on page 13-2

Introduced before R2006a

fillts

Fill missing values in time series

Syntax

```
newfts = fillts(oldfts,fill_method)

newfts = fillts(oldfts,fill_method,newdates)

newfts = fillts(oldfts,fill_method,newdates,{'T1','T2',...})

newfts = fillts(oldfts,fill_method,newdates,'SPAN',{'TS','TE'},delta)

newfts = fillts(... sortmode)
```

Arguments

`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 character vector, or a column cell array of character vector dates. 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

Example 1. Create a financial time series object with missing data in the fourth and fifth rows.

```
dates = ['01-Jan-2001';'01-Jan-2001'; '02-Jan-2001';...
         '02-Jan-2001'; '03-Jan-2001';'03-Jan-2001'];
times = ['11:00';'12:00';'11:00';'12:00';'11:00';'12:00'];
dates_times = cellstr([dates, repmat(' ',size(dates,1),1),...
                       times]);
OpenFts = fints(dates_times,[(1:3)'; nan; nan; 6],{'Data1'},1,...
'Open Financial Time Series');
```

`OpenFts` looks like this:

```
OpenFts =

    desc:  Open Financial Time Series
    freq:  Daily (1)

    'dates:  (6)'    'times:  (6)'    'Data1:  (6)'
    '01-Jan-2001'    '11:00'          [          1]
    '     "     '    '12:00'          [          2]
    '02-Jan-2001'    '11:00'          [          3]
    '     "     '    '12:00'          [        NaN]
    '03-Jan-2001'    '11:00'          [        NaN]
    '     "     '    '12:00'          [          6]
```

Example 2. Fill the missing data in `OpenFts` using cubic interpolation.

```
FilledFts = 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 3. 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 4. 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 5. Use a spanning time interval to add an additional day to `OpenFts`.

```
SpanFts = fillts(OpenFts,'c','04-Jan-2001','span',...
              {'11:00';'12:00'},60,0)

SpanFts =

       desc:  Filled Open Financial Time Series
       freq:  Unknown (0)

       'dates:  (8)'    'times:  (8)'    'Data1:  (8)'
       '01-Jan-2001'    '11:00'          [          1]
       '     "     '    '12:00'          [          2]
       '02-Jan-2001'    '11:00'          [          3]
       '     "     '    '12:00'          [     3.0663]
```

```
'03-Jan-2001'    '11:00'          [     5.8411]
'     "     '    '12:00'          [     6.0000]
'04-Jan-2001'    '11:00'          [     9.8404]
'     "     '    '12:00'          [     9.9994]
```

See Also

`interp1`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

filter

Linear filtering

Syntax

```
newfts = filter(B,A,oldfts)
```

Description

`filter` filters an entire financial time series object with certain filter specifications. The filter is specified in a transfer function expression.

`newfts = filter(B,A,oldfts)` filters the data in the financial time series object `oldfts` with the filter described by vectors `A` and `B` to create the new financial time series object `newfts`. The filter is a "Direct Form II Transposed" implementation of the standard difference equation. `newfts` is a financial time series object containing the same data series (names) as the input `oldfts`.

See Also

`filter2`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

fints

Construct financial time series object

Syntax

```
tsobj = fints(dates_and_data)

tsobj = fints(dates,data)

tsobj = fints(dates,data,datanames)

tsobj = fints(dates,data,datanames,freq)

tsobj = fints(dates,data,datanames,freq,desc)
```

Arguments

`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 MATLAB function `today` to enter date information or the MATLAB function `now` to enter date with time information.

`dates`	Column vector of dates. Dates can be date character vectors 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 character vector format, the entry must be a column-oriented cell array of character vector dates and times when multiple entries are present. Valid date and time character vector 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 help with 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`, and so on. **Note** Not all character vectors are accepted as `datanames` parameters. Supported data series names cannot start with a number and must contain only these characters: • Lowercase Latin alphabet, `a` to `z` • Uppercase Latin alphabet, `A` to `Z` • Underscore, _

`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`	Character vector providing descriptive name for financial time series object. Default = `''`.

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, ..., series*n*, 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)` also allows you to rename the data series. The names are specified in the `datanames` cell array. The number of character vectors 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)` also 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 (`desc`) specified as a character vector for the financial time series object.

Note `fints` only supports hourly and minute time series. Seconds are not supported and will be disregarded when the `fints` object is created (that is, 01-jan-2001 12:00:01 will be considered as 01-jan-2001 12:00). If there are duplicate dates and times, the `fints` constructor 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`.

Examples

Create a Financial Time Series Object Containing Days and Data

Define the data:

```
data = [1:6]'
```

```
data =

     1
     2
     3
     4
     5
     6
```

Define the dates:

```
dates = [today:today+5]'
```

```
dates =

      736939
      736940
      736941
      736942
      736943
      736944
```

Create the financial times series object:

```
tsobjkt = fints(dates, data)
```

```
tsobjkt =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (6)'    'series1:  (6)'
    '01-Sep-2017'    [            1]
    '02-Sep-2017'    [            2]
    '03-Sep-2017'    [            3]
    '04-Sep-2017'    [            4]
```

```
    '05-Sep-2017'    [            5]
    '06-Sep-2017'    [            6]
```

Create a Financial Time Series Object Containing Days, Time of Day, and Data

Define the data:

```
data = [1:6]'
```

```
data =

     1
     2
     3
     4
     5
     6
```

Define the dates:

```
dates = [now:now+5]'
```

```
dates =

   1.0e+05 *

    7.3694
    7.3694
    7.3694
    7.3694
    7.3694
    7.3694
```

Create the financial times series object:

```
tsobjkt = fints(dates, data)
```

```
tsobjkt =

    desc:  (none)
```

```
    freq:  Unknown (0)

    'dates:  (6)'    'times:  (6)'    'series1:  (6)'
    '01-Sep-2017'    '16:30'          [            1]
    '02-Sep-2017'    '16:30'          [            2]
    '03-Sep-2017'    '16:30'          [            3]
    '04-Sep-2017'    '16:30'          [            4]
    '05-Sep-2017'    '16:30'          [            5]
    '06-Sep-2017'    '16:30'          [            6]
```

Create a Financial Time Series Object From a Single Input for Dates and Times

Define the dates and times:

```
dates_and_times = (now:now+5)'
```

```
dates_and_times =

   1.0e+05 *

    7.3694
    7.3694
    7.3694
    7.3694
    7.3694
    7.3694
```

Create the financial times series object:

```
 f = fints(dates_and_times, randn(6,1))
```

```
f =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (6)'    'times:  (6)'    'series1:  (6)'
    '01-Sep-2017'    '16:50'          [       0.5377]
    '02-Sep-2017'    '16:50'          [       1.8339]
    '03-Sep-2017'    '16:50'          [      -2.2588]
```

```
'04-Sep-2017'    '16:50'          [       0.8622]
'05-Sep-2017'    '16:50'          [       0.3188]
'06-Sep-2017'    '16:50'          [      -1.3077]
```

This generates a financial time series object, `f`, and obtains the dates and data from the matrix `dates_and_times`. The dates and times in the input matrix must be oriented column-wise (i.e. the date series and each time series are columns in the input matrix). In addition, the dates entered must be in the serial date format (i.e. 01-Jan-2001 is 730852). You can also use the function `now` to enter in date information. The names of the series will default to `'series1'`, ..., `'|seriesN'|` where `N` is the total number of columns in `dates_and_times` less 1 (that is the number of data columns). The default contents of the `desc` and `freq` fields are`|''|` and `| 'Unknown'|` (0), respectively.

- "Data Transformation and Frequency Conversion" on page 12-12
- "Using Time Series to Predict Equity Return" on page 12-25

See Also

`datenum` | `datestr` | `ftsgui` | `ftstool`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

floatdiscmargin

Discount margin for floating-rate bond

Syntax

```
Margin = floatdiscmargin(Price,SpreadSettle,Maturity,RateInfo,
LatestFloatingRate)
Margin = floatdiscmargin(___,Name,Value)
```

Description

`Margin = floatdiscmargin(Price,SpreadSettle,Maturity,RateInfo, LatestFloatingRate)` calculates the discount margin or zero discount margin for a floating-rate bond.

The input `RateInfo` determines whether the discount margin or the zero discount margin is calculated. Principal schedules are supported using `Principal`.

`Margin = floatdiscmargin(___,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Zero Discount Margin Using a Yield Curve

Use `floatdiscmargin` to compute the discount margin and zero discount margin for a floating-rate note.

Define data for the floating-rate note.

```
Price = 99.99;
Spread = 50;
Settle = '20-Jan-2011';
Maturity = '15-Jan-2012';
```

```
LatestFloatingRate = 0.05;
StubRate = 0.049;
SpotRate = 0.05;
Reset = 4;
Basis = 2;
```

Compute the discount margin.

```
dMargin = floatdiscmargin(Price, Spread, Settle, Maturity, ...
[StubRate, SpotRate], LatestFloatingRate,'Reset', Reset, 'Basis', Basis, ...
'AdjustCashFlowsBasis', true)
```

```
dMargin = 48.4810
```

Usually you want to set `AdjustCashFlowsBasis` to `true`, so cash flows are calculated with adjustments on accrual amounts.

Create an annualized zero-rate term structure to calculate the zero discount margin.

```
Rates = [0.0500;
         0.0505;
         0.0510;
         0.0520];
StartDates = ['20-Jan-2011';
              '15-Apr-2011';
              '15-Jul-2011';
              '15-Oct-2011'];
EndDates =   ['15-Apr-2011';
              '15-Jul-2011';
              '15-Oct-2011';
              '15-Jan-2012'];
ValuationDate = '20-Jan-2011';
RateSpec = intenvset('Compounding', Reset, 'Rates', Rates,...
'StartDates', StartDates, 'EndDates', EndDates,...
'ValuationDate', ValuationDate, 'Basis', Basis);
```

Calculate the zero discount margin using the previous yield curve.

```
dMargin = floatdiscmargin(Price, Spread, Settle, Maturity, ...
RateSpec, LatestFloatingRate,'Reset', Reset, 'Basis', Basis, ...
'AdjustCashFlowsBasis', true)
```

```
dMargin = 46.0688
```

Compute the Zero Discount Margin Using a Yield Curve With datetime Inputs

Use `floatdiscmargin` to compute the discount margin and zero discount margin for a floating-rate note using `datetime` inputs.

```
Price = 99.99;
Spread = 50;
Settle = '20-Jan-2011';
Maturity = '15-Jan-2012';
LatestFloatingRate = 0.05;
StubRate = 0.049;
SpotRate = 0.05;
Reset = 4;
Basis = 2;

Settle = datetime(Settle,'Locale','en_US');
Maturity = datetime(Maturity,'Locale','en_US');
dMargin = floatdiscmargin(Price, Spread, Settle, Maturity, ...
[StubRate, SpotRate], LatestFloatingRate,'Reset', Reset, 'Basis', Basis, ...
'AdjustCashFlowsBasis', true)

dMargin = 48.4810
```

Input Arguments

`Price` — Bond prices where discount margin is to be computed

matrix

Bond prices where discount margin is to be computed, specified as a `NINST`-by-`1` matrix.

Note The spread is calculated against the clean price (the function internally does not add the accrued interest to the price specified by the `Price` input). If the spread is required against the dirty price, you must supply the dirty price for the `Price` input.

Data Types: `double`

`Spread` — Number of basis points over the reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` matrix.

Data Types: `double`

`Settle` — Settlement date of the floating-rate bonds
serial date number | date character vector | datetime

Settlement date of the floating-rate bonds, specified as serial date number, date character vector, or datetime array. If supplied as a `NINST`-by-`1` vector of dates, all settlement dates must be the same (only a single settlement date is supported)

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of the floating-rate bond
serial date number | date character vector | datetime

Maturity date of the floating-rate bond, specified as serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`RateInfo` — Interest-rate information
numeric

interest-rate information, specified as `NINST`-by-`2` vector where the:

- First column is the stub rate between the settlement date and the first coupon rate.
- Second column is the reference rate for the term of the floating coupons (for example, the 3-month LIBOR from settlement date for a bond with a `Reset` of `4`).

Note If the `RateInfo` argument is an annualized zero-rate term structure created by `intenvset`, the zero discount margin is calculated.

Data Types: `double`

`LatestFloatingRate` — Rate for next floating payment set at last reset date
numeric

Rate for the next floating payment set at the last reset date, specified as `NINST`-by-`1` vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Margin = floatdiscmargin(Price,Spread,Settle,Maturity,RateInfo,LatestFloatingRate,'Reset',2,'Basis',5)`

`Reset` — Frequency of payments per year

`1` (default) | numeric

Frequency of payments per year, specified as `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis used for time factor calculations

`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Day-count basis used for time factor calculations, specified as a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`Principal` — Notional principal amounts
`100` (default) | numeric

Notional principal amounts, specified as a `NINST`-by-`1` vector or a `NINST`-by-`1` cell array where each element is a `NUMDATES`-by-`2` cell array where the first column is dates and the second column is the associated principal amount. The date indicates the last day that the principal value is valid.

Data Types: `double` | `cell`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a `NINST`-by-`1` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`AdjustCashFlowsBasis` — Adjust cash flows according to accrual amount
`0` (not in effect) (default) | nonnegative integer `0` or `1`

Adjusts cash flows according to the accrual amount, specified as a `NINST`-by-`1` vector of logicals.

Note Usually you want to set `AdjustCashFlowsBasis` to `1`, so cash flows are calculated with adjustments on accrual amounts. The default is set to `0` to be consistent with `floatbyzero`.

Data Types: `logical`

`Holidays` — Dates for holidays
`holidays.m` used (default)

Dates for holidays, specified as `NHOLIDAYS`-by-`1` vector of MATLAB dates using serial date numbers, date character vectors, or datetime arrays. Holidays are used in computing business days.

Data Types: `double` | `char` | `datetime`

`BusinessDayConvention` — Business day conventions
`'actual'` (default) | character vector with values `'actual'`, `'follow'`, `'modifiedfollow'`, `'previous'` or `'modifiedprevious'`

Business day conventions, specified as a `NINST`-by-`1` cell array of character vectors of business day conventions to be used in computing payment dates. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `'actual'` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Margin` — Discount margin
numeric

Discount margin, returned as a `NINST`-by-`1` vector of the discount margin if `RateInfo` is specified as a `NINST`-by-`2` vector of stub and spot rates.

If `RateInfo` is specified as an annualized zero rate term structure created by `intenvset`, `Margin` is returned as a `NINST`-by-`NCURVES` matrix of the zero discount margin.

References

[1] Fabozzi, Frank J., Mann, Steven V. *Floating-Rate Securities.* John Wiley and Sons, New York, 2000.

[2] Fabozzi, Frank J., Mann, Steven V. *Introduction to Fixed Income Analytics: Relative Value Analysis, Risk Measures and Valuation.* John Wiley and Sons, New York, 2010.

[3] O'Kane, Dominic, Sen, Saurav. *"Credit Spreads Explained."* Lehman Brothers Fixed Income Quantitative Research, March 2004.

See Also

`bndspread` | `datetime` | `floatbyzero` | `floatmargin` | `intenvset`

Topics

"Fixed-Income Terminology" on page 2-25

Introduced in R2012b

floatmargin

Margin measures for floating-rate bond

Syntax

```
[Margin,AdjPrice] = floatmargin(Price,SpreadSettle,Maturity)
[Margin,AdjPrice] = floatmargin(___,Name,Value)
```

Description

`[Margin,AdjPrice] = floatmargin(Price,SpreadSettle,Maturity)` calculates margin measures for a floating-rate bond.

Use `floatmargin` to calculate the following types of margin measures for a floating-rate bond:

- Spread for life
- Adjusted simple margin
- Adjusted total margin

To calculate the discount margin or zero discount margin, see `floatdiscmargin`.

`[Margin,AdjPrice] = floatmargin(___,Name,Value)` adds optional name-value pair arguments.

Examples

Compute Margin Measures for a Floating-Rate Bond

Use `floatmargin` to compute margin measures for `spreadforlife`, `adjustedsimple`, and `adjustedtotal` for a floating-rate note.

Define data for the floating-rate note.

```
Price = 99.99;
Spread = 50;
Settle = '20-Jan-2011';
Maturity = '15-Jan-2012';
LatestFloatingRate = 0.05;
StubRate = 0.049;
SpotRate = 0.05;
Reset = 4;
Basis = 2;
```

Calculate `spreadforlife`.

```
Margin = floatmargin(Price, Spread, Settle, Maturity, 'Reset', ...
Reset, 'Basis', Basis)
```

```
Margin = 51.0051
```

Calculate `adjustedsimple` margin.

```
[Margin, AdjPrice] = floatmargin(Price, Spread, Settle, Maturity, ...
'SpreadType', 'adjustedsimple', 'RateInfo', [StubRate, SpotRate], ...
'LatestFloatingRate', LatestFloatingRate, 'Reset', Reset, 'Basis', Basis)
```

```
Margin = 53.2830
```

```
AdjPrice = 99.9673
```

Calculate `adjustedtotal` margin.

```
[Margin, AdjPrice] = floatmargin(Price, Spread, Settle, Maturity, ...
'SpreadType', 'adjustedtotal', 'RateInfo', [StubRate, SpotRate], ...
'LatestFloatingRate', LatestFloatingRate, 'Reset', Reset, 'Basis', Basis)
```

```
Margin = 53.4463
```

```
AdjPrice = 99.9673
```

Compute Margin Measures for a Floating-Rate Bond Using datetime Inputs

Use `floatmargin` to calculate margin measures for `spreadforlife`, `adjustedsimple`, and `adjustedtotal` for a floating-rate note using `datetime` inputs.

```
Price = 99.99;
Spread = 50;
```

```
Settle = '20-Jan-2011';
Maturity = '15-Jan-2012';
LatestFloatingRate = 0.05;
StubRate = 0.049;
SpotRate = 0.05;
Reset = 4;
Basis = 2;

Settle = datetime(Settle,'Locale','en_US');
Maturity = datetime(Maturity,'Locale','en_US');
[Margin, AdjPrice] = floatmargin(Price, Spread, Settle, Maturity, ...
'SpreadType', 'adjustedsimple', 'RateInfo', [StubRate, SpotRate], ...
'LatestFloatingRate', LatestFloatingRate, 'Reset', Reset, 'Basis', Basis)

Margin = 53.2830

AdjPrice = 99.9673
```

Input Arguments

`Price` — Bond prices where spreads are to be computed

matrix

Bond prices where spreads are to be computed, specified as a `NINST`-by-`1` matrix.

Data Types: `double`

`Spread` — Number of basis points over the reference rate

numeric

Number of basis points over the reference rate, specified as a `NINST`-by-`1` matrix.

Data Types: `double`

`Settle` — Settlement date of the floating-rate bonds

serial date number | date character vector | datetime

Settlement date of the floating-rate bonds, specified as serial date number, date character vector, or datetime array. If supplied as a `NINST`-by-`1` vector of dates, all settlement dates must be the same (only a single settlement date is supported)

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of the floating-rate bond
serial date number | date character vector | datetime

Maturity date of the floating-rate bond, specified as serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[Margin,AdjPrice] = floatmargin(Price,Spread,Settle,Maturity, 'SpreadType','adjustedtotal','RateInfo', [StubRate,SpotRate],'LatestFloatingRate',.0445,'Reset',2,'Basis',5)`

`SpreadType` — Type of spread to calculate
spreadforlife (default) | `adjustedsimpleadjustedtotal`

Type of spread to calculates, specified by type, specified as `spreadforlife`,`adjustedsimple`, or `adjustedtotal`.

Note If the `SpreadType` is `spreadforlife` (default), then the name-value arguments `LatestFloatingRate` and `RateInfo` are not used. If the `SpreadType` is `adjustedsimple` or `adjustedtotal`, then the name-value arguments `LatestFloatingRate` and `RateInfo` must be specified.

Data Types: `double`

`LatestFloatingRate` — Rate for next floating payment set at last reset date
numeric

Rate for the next floating payment set at the last reset date, specified as `NINST`-by-`1` vector.

Note This rate must be specified for a `SpreadType` of `adjustedsimple` and `adjustedtotal`.

Data Types: `double`

`RateInfo` — Interest-rate information
numeric

interest-rate information, specified as `NINST`-by-`2` vector where the:

- First column is the stub rate between the settlement date and the first coupon rate.
- Second column is the reference rate for the term of the floating coupons (for example, the 3-month LIBOR from settlement date for a bond with a `Reset` of `4`).

Note The `RateInfo` must be specified for `SpreadType` of `adjustedsimple` and `adjustedtotal`.

Data Types: `double`

`Reset` — Frequency of payments per year
`1` (default) | numeric

Frequency of payments per year, specified as `NINST`-by-`1` vector.

Data Types: `double`

`Basis` — Day-count basis used for time factor calculations
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Day-count basis used for time factor calculations, specified as a `NINST`-by-`1` vector. Values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365

- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

`Principal` — Notional principal amounts
`100` (default) | numeric

Notional principal amounts, specified as `NINST`-by-`1` vector.

Data Types: `double`

`EndMonthRule` — End-of-month rule flag
`1` (in effect) (default) | nonnegative integer `0` or `1`

End-of-month rule flag, specified as a `NINST`-by-`1` vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days.

- `0` = Ignore rule, meaning that a bond coupon payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a bond coupon payment date is always the last actual day of the month.

Data Types: `logical`

`Holidays` — Dates for holidays
`holidays.m` used (default)

Dates for holidays, specified as `NHOLIDAYS`-by-`1` vector of MATLAB dates using serial date numbers, date character vectors, or datetime arrays. Holidays are used in computing business days.

Data Types: `double` | `char` | `datetime`

`BusinessDayConvention` — Business day conventions
`'actual'` (default) | character vector with values `'actual'`, `'follow'`, `'modifiedfollow'`, `'previous'` or `'modifiedprevious'`

Business day conventions, specified as a `NINST`-by-`1` cell array of character vectors of business day conventions to be used in computing payment dates. The selection for business day convention determines how nonbusiness days are treated. Nonbusiness days are defined as weekends plus any other date that businesses are not open (for example, statutory holidays). Values are:

- `'actual'` — Nonbusiness days are effectively ignored. Cash flows that fall on non-business days are assumed to be distributed on the actual date.
- `'follow'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the following business day.
- `'modifiedfollow'` — Cash flows that fall on a non-business day are assumed to be distributed on the following business day. However if the following business day is in a different month, the previous business day is adopted instead.
- `'previous'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day.
- `'modifiedprevious'` — Cash flows that fall on a nonbusiness day are assumed to be distributed on the previous business day. However if the previous business day is in a different month, the following business day is adopted instead.

Data Types: `char` | `cell`

Output Arguments

`Margin` — Spreads for floating-rate bond
numeric

Spreads for the floating-rate bond, returned as a `NINST`-by-`1` vector.

`AdjPrice` — Adjusted price used to calculate spreads for `SpreadType` of `adjustedsimple` and `adjustedtotal`
numeric

Adjusted price used to calculate spreads for `SpreadType` of `adjustedsimple` and `adjustedtotal`, returned as a `NINST`-by-`1` vector.

References

[1] Fabozzi, Frank J., Mann, Steven V. *Floating-Rate Securities.* John Wiley and Sons, New York, 2000.

[2] Fabozzi, Frank J., Mann, Steven V. *Introduction to Fixed Income Analytics: Relative Value Analysis, Risk Measures and Valuation.* John Wiley and Sons, New York, 2010.

See Also

`bndspread` | `datetime` | `floatbyzero` | `floatdiscmargin`

Topics

"Fixed-Income Terminology" on page 2-25

Introduced in R2012b

fpctkd

Fast stochastics

Syntax

```
[pctk,pctd] = fpctkd(highp,lowp,closep)

[pctk,pctd] = fpctkd([highp lowp closep])

[pctk,pctd] = fpctkd(highp,lowp,closep,kperiods,dperiods, dmamethod)

[pctk,pctd] = fpctkd([highp lowp closep],kperiods,dperiods,dmamethod)

pkdts = fpctkd(tsobj,kperiods,dperiods,dmamethod)

pkdts = fpctkd(tsobj,kperiods,dperiods,dmamethod,'ParameterName',ParameterValue, ...)
```

Arguments

`highp`	High price (vector).
`lowp`	Low price (vector).
`closep`	Closing price (vector).
`kperiods`	(Optional) %K periods. Default = `10`.
`dperiods`	(Optional) %D periods. Default = `3`.
`damethod`	(Optional) %D moving average method. Default = `'e'` (exponential).
`tsobj`	Financial time series object.
`'ParameterName'`	Valid parameter names are: • `HighName`: high prices series name • `LowName`: low prices series name • `CloseName`: closing prices series name

`ParameterValue`	Parameter values are the character vectors that represent the valid parameter names.

Description

`fpctkd` calculates the stochastic oscillator.

`[pctk,pctd] = fpctkd(highp,lowp,closep)` calculates the fast stochastics F%K and F%D from the stock price data `highp` (high prices), `lowp` (low prices), and `closep` (closing prices).

`[pctk,pctd] = fpctkd([highp lowp closep])` accepts a three-column matrix of high (`highp`), low (`lowp`), and closing prices (`closep`), in that order.

`[pctk,pctd] = fpctkd(highp,lowp,closep,kperiods,dperiods,dmamethod)` calculates the fast stochastics F%K and F%D from the stock price data `highp` (high prices), `lowp` (low prices), and `closep` (closing prices). `kperiods` sets the %K period. `dperiods` sets the %D period.

`damethod` specifies the %D moving average method. Valid moving average methods for %D are Exponential (`'e'`) and Triangular (`'t'`). See `tsmovavg` for explanations of these methods.

`[pctk,pctd]= fpctkd([highp lowp closep],kperiods,dperiods,dmamethod)` accepts a three-column matrix of high (`highp`), low (`lowp`), and closing prices (`closep`), in that order.

`pkdts = fpctkd(tsobj,kperiods,dperiods,dmamethod)` calculates the fast stochastics F%K and F%D from the stock price data in the financial time series object `tsobj`. `tsobj` must minimally contain the series `High` (high prices), `Low` (low prices), and `Close` (closing prices). `pkdts` is a financial time series object with similar dates to `tsobj` and two data series named `PercentK` and `PercentD`.

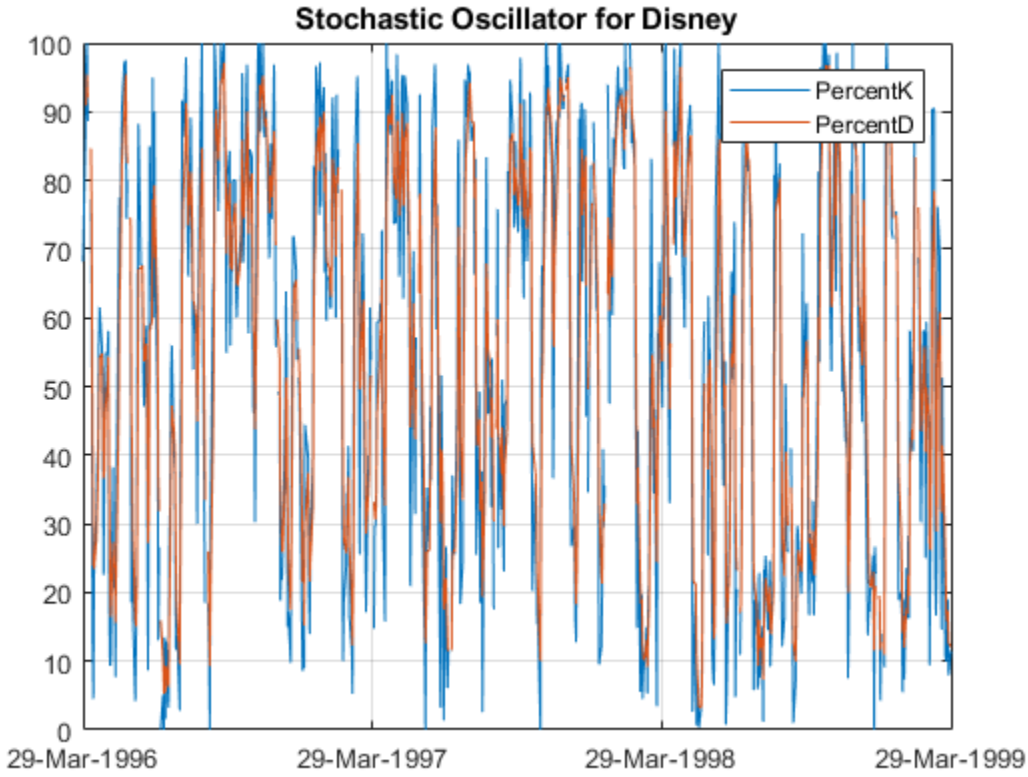
`pkdts = fpctkd(tsobj,kperiods,dperiods,dmamethod,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Stochastic Oscillator

This example shows how to compute the stochastic oscillator for Disney stock and plot the results.

```
load disney.mat
dis_FastStoc = fpctkd(dis);
plot(dis_FastStoc)
title('Stochastic Oscillator for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 268–271.

See Also

`spctkd` | `stochosc` | `tsmovavg`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

frac2cur

Fractional currency value to decimal value

Syntax

```
Decimal = frac2cur(Fraction,Denominator)
```

Description

`Decimal = frac2cur(Fraction,Denominator)` converts a fractional currency value to a decimal value. `Fraction` is the fractional currency value input as a character vector, and `Denominator` is the denominator of the fraction.

Examples

Convert a Fractional Currency Value to a Decimal Value

This example shows how to convert a fractional currency value to a decimal value.

```
Decimal = frac2cur('12.1', 8)
```

```
Decimal = 12.1250
```

Input Arguments

`Fraction` — Fractional currency values

character vector | cell array of character vectors

Fractional currency values, specified as a character vector or cell array of character vectors.

Data Types: `char` | `cell`

`Denominator` — Denominator of the fractions

numeric

Denominator of the fractions, specified as a scalar or vector using numeric values for the denominator.

Data Types: `double`

Output Arguments

`Decimal` — Decimal currency value

numeric decimal

Decimal currency value, returned as a scalar or vector with numeric decimal values.

Data Types: `double`

See Also

`cur2frac` | `cur2str`

Introduced before R2006a

freqnum

Convert character vector 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 character vector frequency indicator into a numeric value.

Character Vector 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

Character Vector Frequency Indicator	Numeric Representation
`QUARTERLY`, `ly`, `quarterly`, `Q`, `q`	`4`
`SEMIANNUAL`, `Semiannual`, `semiannual`, `S`, `s`	`5`
`ANNUAL`, `Annual`, `annual`, `A`, `a`	`6`

See Also

`freqstr`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

freqstr

Convert numeric frequency indicator to character vector 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 character vector representation.

Numeric Frequency Indicator	Character Vector Representation
0	Unknown
1	Daily
2	Weekly
3	Monthly

Numeric Frequency Indicator	Character Vector Representation
4	Quarterly
5	Semiannual
6	Annual

See Also

freqnum

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

frontier

Rolling efficient frontier

Syntax

```
[PortWts,AllMean,AllCovariance] = frontier(Universe,Window,Offset,NumPorts,ActiveMap,Co
```

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` | `portstats`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Selection and Risk Aversion" on page 3-9
"Active Returns and Tracking Error Efficient Frontier" on page 3-43
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

fts2ascii

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 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` is written to the file just before the data.
`desc`	(Optional) Description text, which is 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`).

Examples

Use fts2ascii to Write a Time Series to an ASCII File

Create a data file with time information.

```
dates = ['01-Jan-2001';'01-Jan-2001'; '02-Jan-2001'; ...
'02-Jan-2001'; '03-Jan-2001';'03-Jan-2001'];
times = ['11:00';'12:00';'11:00';'12:00';'11:00';'12:00'];
serial_dates_times = [datenum(dates), datenum(times)];
data = round(10*rand(6,2));
```

Use `fts2ascii` to write the time series to an ascii file.

```
stat = fts2ascii('myfts_file2.txt',serial_dates_times,data, ...
{'dates';'times';'Data1';'Data2'},'My FTS with Time')
```

```
stat = 1
```

Read the data file back and create a financial time series object using `ascii2fts`.

```
MyFts = ascii2fts('myfts_file2.txt','t',1,2,1)
```

```
MyFts =

    desc:  My FTS with Time
    freq:  Unknown (0)

    'dates:  (6)'    'times:  (6)'    'Data1:  (6)'    'Data2:  (6)'
    '01-Jan-2001'    '11:00'          [          8]    [          3]
    '     "     '    '12:00'          [          9]    [          5]
    '02-Jan-2001'    '11:00'          [          1]    [         10]
    '     "     '    '12:00'          [          9]    [         10]
    '03-Jan-2001'    '11:00'          [          6]    [          2]
    '     "     '    '12:00'          [          1]    [         10]
```

- “Working with Financial Time Series Objects” on page 12-3
- “Creating a Financial Time Series Object” on page 13-11

See Also

`ascii2fts`

Topics

“Working with Financial Time Series Objects” on page 12-3
“Creating a Financial Time Series Object” on page 13-11
“What Is the Financial Time Series App?” on page 13-2

Introduced before R2006a

fts2mat

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 character vectors.

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 you want the dates vector included. The dates vector is 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 character vectors.

`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`

Topics

"Working with Financial Time Series Objects" on page 12-3
"Creating a Financial Time Series Object" on page 13-11
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

ftsbound

Start and end dates

Syntax

```
datesbound = ftsbound(tsobj)

datesbound = ftsbound(tsobj,dateform)
```

Arguments

`tsobj`	Financial time series object
`dateform`	`dateform` is an integer representing the format of a date character vector. 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` also 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 character vectors 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`

Topics

“Using Time Series to Predict Equity Return” on page 12-25
“What Is the Financial Time Series App?” on page 13-2

Introduced before R2006a

ftsgui

Financial time series GUI

Syntax

```
ftsgui
```

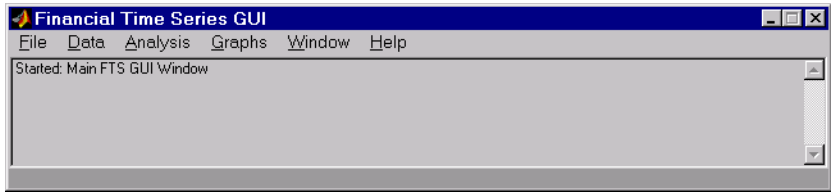
Description

`ftsgui` displays the financial time series graphical user interface (GUI) main window.

The use of the financial time series GUI is described in “Using the Financial Time Series GUI” on page 14-7.

Examples

```
ftsgui
```



See Also

`ftstool`

Topics

“Using the Financial Time Series App” on page 13-11
“Using the Financial Time Series App with GUIs” on page 13-19
“What Is the Financial Time Series App?” on page 13-2

Introduced before R2006a

ftsinfo

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.

Field	Contents
nseries	Number of columns of time series data.

Examples

Convert the supplied file `disney.dat` into a financial time series object named `dis`:

```
dis = ascii2fts('disney.dat', 1, 3);
```

Now use `ftsinfo` to obtain information about `dis`:

```
ftsinfo(dis)
```

```
 FINTS version:   2.0
   Description:  Walt Disney Company (DIS)
     Frequency:  Unknown
    Start date:  29-Mar-1996
      End date:  29-Mar-1999
  Series names:  OPEN
                 HIGH
                 LOW
                 CLOSE
                 VOLUME
     # of data:  782
   # of series:  5
```

Then, executing

```
infodis = ftsinfo(dis)
```

creates the structure `infodis` containing the values

```
infodis =

            ver: '2.0'
           desc: 'Walt Disney Company (DIS)'
           freq: 0
      startdate: '29-Mar-1996'
        enddate: '29-Mar-1999'
    seriesnames: {5x1 cell}
          ndata: 782
        nseries: 5
```

See Also

`fints` | `freqnum` | `freqstr` | `ftsbound`

Topics

"Using the Financial Time Series App" on page 13-11
"Using the Financial Time Series App with GUIs" on page 13-19
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

Financial Time Series

Create and manage Financial Time Series

Description

The **Financial Time Series** app enables you to create and manage Financial Time Series (`fints`) objects.

Open the Financial Time Series App

- MATLAB Toolstrip: On the **Apps** tab, under **Financial Time Series**, click the app icon.
- MATLAB command prompt: Enter `ftstool`.

Examples

- "Loading Data with the Financial Time Series App" on page 13-7
- "Using the Financial Time Series App" on page 13-11
- "Using the Financial Time Series App with GUIs" on page 13-19
- "Getting Started with the Financial Time Series App" on page 13-4

Programmatic Use

```
ftstool
```

See Also

Functions

`ftsgui`

Topics

Introduced in R2006b

ftstool

Financial Time Series app

Syntax

```
ftstool
```

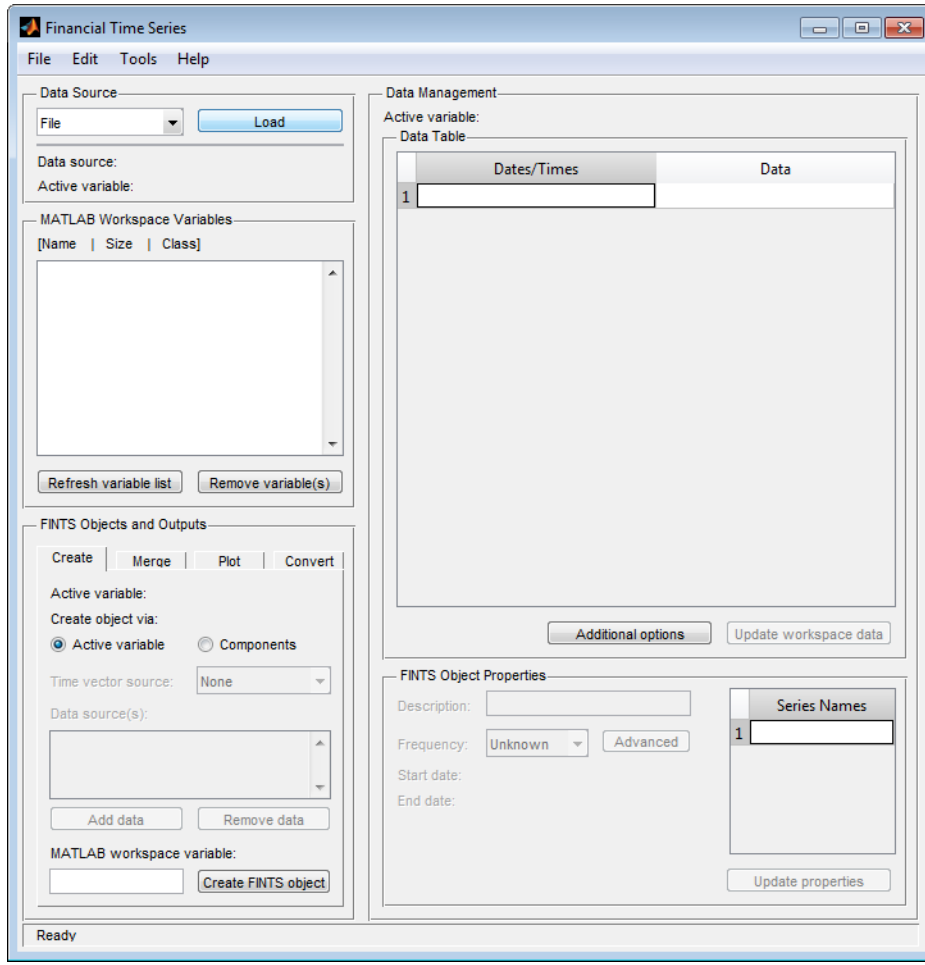
Description

`ftstool` creates and manages Financial Time Series objects. `ftstool` allows the creation and management of Financial Time Series objects via the Financial Time Series app. `ftstool` can interface with `ftsgui`, meaning Line Plots generated with `ftstool` can be analyzed with `ftsgui`. However, `ftsgui` must be running prior to the generation of any Line Plots.

The use of the Financial Time Series app is described in “Getting Started with the Financial Time Series App” on page 13-4.

Examples

```
ftstool
```



Alternatively, on the MATLAB desktop toolstrip, click the **Apps** tab and in the apps gallery, under **Computational Finance**, click **Financial Time Series**. The Financial Time Series app opens. For an overview of the Financial Time Series app, see “What Is the Financial Time Series App?” on page 13-2.

See Also

`ftsgui`

Topics

“Using the Financial Time Series App” on page 13-11
“Using the Financial Time Series App with GUIs” on page 13-19
“What Is the Financial Time Series App?” on page 13-2

Introduced in R2006b

ftsuniq

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 character vectors 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`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

fvdisc

Future value of discounted security

Syntax

```
FutureVal = fvdisc(Settle,Maturity,Price,Discount,Basis)
```

Arguments

`Settle`	Settlement date. Enter as serial date number, date character vector, or datetime array. `Settle` must be earlier than `Maturity`.
`Maturity`	Maturity date. Enter as serial date number, date character vector, or datetime array.
`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) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/365 (ISDA) • 13 = BUS/252 For more information, see **basis** on page Glossary-0 .

Description

`FutureVal = fvdisc(Settle,Maturity,Price,Discount,Basis)` finds the amount received at maturity for a fully vested security.

Examples

Find the Amount Received at Maturity for a Fully-Vested Security

This example shows how to find the amount received at maturity for a fully-vested security, using the following data.

```
Settle = '02/15/2001';
Maturity = '05/15/2001';
```

```
Price = 100;
Discount = 0.0575;
Basis = 2;

FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)

FutureVal = 101.4420
```

Find the Amount Received at Maturity for a Fully-Vested Security Using datetime Inputs

This example shows how to use `datetime` inputs to find the amount received at maturity for a fully-vested security, using the following data.

```
Settle = datetime('02/15/2001','Locale','en_US');
Maturity = datetime('05/15/2001','Locale','en_US');
Price = 100;
Discount = 0.0575;
Basis = 2;

FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)

FutureVal = 101.4420
```

- “Analyzing and Computing Cash Flows” on page 2-21

References

Mayle. *Standard Securities Calculation Methods.* Volumes I-II, 3rd edition.

See Also

`acrudisc` | `datetime` | `discrate` | `prdisc` | `ylddisc`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

fvfix

Future value with fixed periodic payments

Syntax

```
FutureVal = fvfix(Rate,NumPeriods,Payment,PresentVal,Due)
```

Arguments

`Rate`	Periodic interest rate, as a decimal fraction.
`NumPeriods`	Number of periods.
`Payment`	Periodic payment.
`PresentVal`	(Optional) Initial value. Default = `0`.
`Due`	(Optional) When payments are due or made: `0` = end of period (default), or `1` = beginning of period.

Description

`FutureVal = fvfix(Rate,NumPeriods,Payment,PresentVal,Due)` returns the future value of a series of equal payments.

Examples

Return the Future Value of a Series of Equal Payments

This example shows how to compute the future value of a series of equal payments using a savings account that has a starting balance of $1500. $200 is added at the end of each month for 10 years and the account pays 9% interest compounded monthly.

```
FutureVal = fvfix(0.09/12, 12*10, 200, 1500, 0)
```

```
FutureVal = 4.2380e+04
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`fvvar` | `pvfix` | `pvvar`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

fvvar

Future value of varying cash flow

Syntax

```
FutureVal = fvvar(CashFlow,Rate,CFDates)
```

Arguments

`CashFlow`	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
`Rate`	Periodic interest rate. Enter as a decimal fraction.
`CFDates`	(Optional) For irregular (nonperiodic) cash flows, a vector of dates on which the cash flows occur. Enter dates as serial date numbers, date character vectors, or datetime arrays. Default assumes `CashFlow` contains regular (periodic) cash flows.

Description

`FutureVal = fvvar(CashFlow,Rate,CFDates)` returns the future value of a varying cash flow.

Examples

This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 8%.

Year 1	$2000
Year 2	$1500
Year 3	$3000

Year 4	$3800
Year 5	$5000

For the future value of this regular (periodic) cash flow

```
FutureVal = fvvar([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
FutureVal =

            2520.47
```

An investment of $10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

Cash Flow	Dates
($10000)	January 12, 2000
$2500	February 14, 2001
$2000	March 3, 2001
$3000	June 14, 2001
$4000	December 1, 2001

To calculate the future value of this irregular (nonperiodic) cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];

CFDates = ['01/12/2000'
            '02/14/2001'
            '03/03/2001'
            '06/14/2001'
            '12/01/2001'];

FutureVal = fvvar(CashFlow, 0.09, CFDates)
```

returns

```
FutureVal =

            170.66
```

See Also

`datetime` | `fvfix` | `irr` | `payuni` | `pvfix` | `pvvar`

Topics

"Analyzing and Computing Cash Flows" on page 2-21

Introduced before R2006a

fwd2zero

Zero curve given forward curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the new optional name-value pair inputs: `InputCompounding`, `InputBasis`, `OutputCompounding`, and `OutputBasis`.

Syntax

```
[ZeroRates,CurveDates] = fwd2zero(ForwardRates,CurveDates,Settle)
[ZeroRates,CurveDates] = fwd2zero(___,Name,Value)
```

Description

`[ZeroRates,CurveDates] = fwd2zero(ForwardRates,CurveDates,Settle)` returns a zero curve given an implied forward rate curve and its maturity dates. If both inputs for `CurveDates` and `Settle` are serial date numbers or date character vectors, `CurveDates` is returned as serial date numbers. However, if either of the inputs for `CurveDates` and `Settle` are a datetime array, `CurveDates` is returned as a datetime array.

`[ZeroRates,CurveDates] = fwd2zero(___,Name,Value)` adds optional name-value pair arguments

Examples

Compute the Zero Curve Given the Forward Curve

This example shows how to compute the zero curve, given an implied forward rate curve over a set of maturity dates, a settlement date, and a compounding rate.

```
ForwardRates = [0.0469
                0.0519
                0.0549
                0.0535
                0.0558
                0.0508
                0.0560
                0.0545
                0.0615
                0.0486];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');

InputCompounding = 1;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;
```

Execute the function `fwd2zero` to return the zero-rate curve `ZeroRates` at the maturity dates `CurveDates`.

```
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,...
Settle,'InputCompounding',1,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)
```

```
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
```

Compute the Zero Curve Given the Forward Curve Using datetime Inputs

This example shows how to use `datetime` inputs compute the zero curve, given an implied forward rate curve over a set of maturity dates, a settlement date, and a compounding rate.

```
ForwardRates = [0.0469
                0.0519
                0.0549
                0.0535
                0.0558
                0.0508
                0.0560
                0.0545
                0.0615
                0.0486];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
```

```
               datenum('25-Jun-2001')
               datenum('04-Sep-2001')
               datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');

InputCompounding = 1;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;CurveDates = datetime(CurveDates,'ConvertFrom','datenum','Locale','en_U
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,...
Settle,'InputCompounding',1,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)

ZeroRates =

    0.0469
    0.0515
    0.0531
    0.0532
    0.0538
    0.0532
    0.0536
    0.0539
    0.0556
    0.0543


CurveDates = 10x1 datetime array
   06-Nov-2000 00:00:00
   11-Dec-2000 00:00:00
   15-Jan-2001 00:00:00
   05-Feb-2001 00:00:00
   04-Mar-2001 00:00:00
   02-Apr-2001 00:00:00
   30-Apr-2001 00:00:00
   25-Jun-2001 00:00:00
   04-Sep-2001 00:00:00
   12-Nov-2001 00:00:00
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`ForwardRates` — Annualized implied forward rates
decimal fraction

Annualized implied forward rates, specified as a (`NUMBONDS`)-by-`1` vector using 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.

Data Types: `double`

`CurveDates` — Maturity dates
serial date number | date character vector | datetime

Maturity dates, specified as a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays, that correspond to the `ForwardRates`.

Data Types: `double` | `datetime` | `char`

`Settle` — Common settlement date for `ForwardRates`
serial date number | date character vector | datetime

Common settlement date for `ForwardRates`, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
[ZeroRates,CurveDates] =
fwd2zero(ForwardRates,CurveDates,Settle,'InputCompounding',
3,'InputBasis',5,'OutputCompounding',4,'OutputBasis',5)
```

`InputCompounding` — Compounding frequency of input forward rates
`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of input forward rates, specified with allowed values:

- `0` — Simple interest (no compounding)
- `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

Note If `InputCompounding` is not specified, then `InputCompounding` is assigned the value specified for `OutputCompounding`. If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2`

Data Types: `double`

`InputBasis` — Day-count basis of input forward rates

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of input forward rates, specified as a numeric value. Allowed values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `InputBasis` is not specified, then `InputBasis` is assigned the value specified for `OutputBasis`. If either `InputBasis` or `Outputbasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

`OutputCompounding` — Compounding frequency of output zero rates
`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of output zero rates, specified with the allowed values:

- `0` — Simple interest (no compounding)
- `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

Note If `OutputCompounding` is not specified, then `OutputCompounding` is assigned the value specified for `InputCompounding`. If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`OutputBasis` — Day-count basis of output zero rates
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of output zero rates, specified as a numeric value. Allowed values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `OutputBasis` is not specified, then `OutputBasis` is assigned the value specified for `InputBasis`. If either `InputBasis` or `OutputBasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

Output Arguments

`ZeroRates` — Zero curve for investment horizon represented by `CurveDates`
numeric

Zero curve for the investment horizon represented by `CurveDates`, returned as 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` — Maturity dates that correspond to `ZeroRates`
serial date number | date character vector | datetime

Maturity dates that correspond to the `ZeroRates`, returned as a `NUMBONDS`-by-`1` vector of maturity dates that correspond to the zero rates in `ZeroRates`. This vector is the same as the input vector `CurveDates`, but is sorted by ascending maturity.

If both inputs for `CurveDates` and `Settle` are serial date numbers or date character vectors, `CurveDates` is returned as serial date numbers. However, if either of the inputs for `CurveDates` and `Settle` are a datetime array, `CurveDates` is returned as a datetime array.

See Also

`prbyzero` | `pyld2zero` | `zbtprice` | `zbtyield` | `zero2disc` | `zero2fwd` | `zero2fwd` | `zero2pyld`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

gbm class

Geometric Brownian motion model

Description

Geometric Brownian motion (GBM) models allow you to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time GBM stochastic processes. Specifically, this model allows the simulation of vector-valued GBM processes of the form

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`NVARS` generalized expected instantaneous rate of return matrix.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector X_t.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Construction

`GBM = gbm(Return,Sigma)` constructs a default `gbm` object.

`GBM = gbm(Return,Sigma,Name,Value)` constructs a `gbm` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `gbm` object, see `gbm`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Return` — `Return` represents the parameter μ

array or deterministic function of time or deterministic function of time and state

`Return` represents the parameter μ, specified as an array or deterministic function of time.

If you specify `Return` as an array, it must be an `NVARS`-by-`NVARS` matrix representing the expected (mean) instantaneous rate of return.

As a deterministic function of time, when `Return` is called with a real-valued scalar time `t` as its only input, `Return` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Return` as a function of time and state, it must return an `NVARS`-by-`NVARS` matrix when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter V

array or deterministic function of time or deterministic function of time and state

`Sigma` represents the parameter V, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Although the `gbm` constructor enforces no restrictions on the sign of `Sigma` volatilities, they are specified as positive values.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `gbm`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)

value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express`drift` and`diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A`

and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the`gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the `gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`Simulation` — User-defined simulation function or SDE simulation method
if you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `function_handle`

Methods

simBySolution	Simulate approximate solution of diagonal-drift GBM processes

Inherited Methods

The following methods are inherited from the `sde` class.

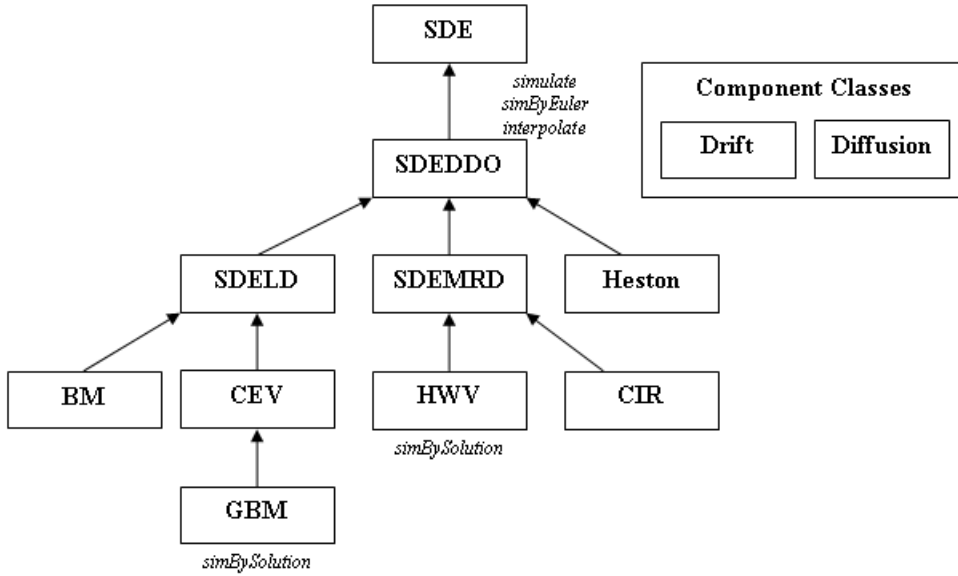
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a gbm Object

Create a univariate `gbm` object to represent the model: $dX_t = 0.25X_t dt + 0.3X_t dW_t$.

```
obj = gbm(0.25, 0.3)  % (B = Return, Sigma)

obj =
   Class GBM: Generalized Geometric Brownian Motion
   -----------------------------------------------
```

```
  Dimensions: State = 1, Brownian = 1
-----------------------------------------------
   StartTime: 0
  StartState: 1
 Correlation: 1
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
      Return: 0.25
       Sigma: 0.3
```

`gbm` objects display the parameter `B` as the more familiar `Return`

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally

specified an input as an array, `gbm` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`bm` | `cev` | `diffusion` | `drift` | `interpolate` | `simByEuler` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"SDEs" on page 17-2

"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

gbm

Construct GBM model

Syntax

```
GBM = gbm(Return, Sigma)

GBM = gbm(Return, Sigma, Name, Value)
```

Class

`gbm`

Description

This function creates and displays geometric Brownian motion (GBM) models, which derive from the `cev` (constant elasticity of variance) class. Use GBM models to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time GBM stochastic processes.

This function allows simulation of vector-valued GBM processes of the form:
$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`NVARS` generalized expected instantaneous rate of return matrix.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector X_t.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Return`	`Return` represents the parameter μ. If you specify `Return` as an array and it must be an `NVARS`-by-`NVARS` matrix representing the expected (mean) instantaneous rate of return. As a deterministic function of time, when `Return` is called with a real-valued scalar time `t` as its only input, `Return` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Return` as a function of time and state, it must return an `NVARS`-by-`NVARS` matrix when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t. Although the `gbm` constructor enforces no restrictions on the sign of `Sigma` volatilities, they are specified as positive values.

Optional Input Arguments

Specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character vector, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.

`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `gbm` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `gbm` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `gbm` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`GBM`	Geometric Brownian motion model with the following displayed parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at `StartTime` • `Correlation`: Access function for the `Correlation` input, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method • `Return`: Access function for the input argument `Return`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Examples

- “Creating Geometric Brownian Motion (GBM) Models” on page 17-27
- “Representing Market Models Using SDELD, CEV, and GBM Objects” on page 17-37

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, the `gbm` constructor treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`cev` | `diffusion` | `drift`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23

"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

simBySolution

Class: gbm

Simulate approximate solution of diagonal-drift GBM processes

Syntax

```
[Paths,Times,Z] = simBySolution(MDL,NPERIODS)
[Paths,Times,Z] = simBySolution(MDL,NPERIODS,Name,Value)
```

Description

`[Paths,Times,Z] = simBySolution(MDL,NPERIODS)` simulates approximate solution of diagonal-drift for geometric Brownian motion (GBM) processes.

`[Paths,Times,Z] = simBySolution(MDL,NPERIODS,Name,Value)` simulates approximate solution of diagonal-drift for GBM processes with additional options specified by one or more `Name,Value` pair arguments.

The `simBySolution` method simulates `NTRIALS` sample paths of `NVARS` correlated state variables, driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time GBM short-rate models by an approximation of the closed-form solution.

Consider a separable, vector-valued GBM model of the form:

$$dX_t = \mu(t)X_t dt + D(t, X_t)V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- μ is an `NVARS`-by-`NVARS` generalized expected instantaneous rate of return matrix.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

The `simBySolution` method simulates the state vector X_t using an approximation of the closed-form solution of diagonal-drift models.

When evaluating the expressions, `simBySolution` assumes that all model parameters are piecewise-constant over each simulation period.

In general, this is *not* the exact solution to the models, because the probability distributions of the simulated and true state vectors are identical *only* for piecewise-constant parameters.

When parameters are piecewise-constant over each observation period, the simulated process is exact for the observation times at which X_t is sampled.

Input Arguments

`MDL` — Geometric Brownian motion (GBM) model

`gbm` object

Geometric Brownian motion (GBM) model, specified as a `gbm` object that is created using the `gbm` constructor.

Data Types: `struct`

`NPERIODS` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of this argument determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`NTRIALS` — Number of simulated trials (sample paths)

`1` (default) | positive scalar integer

Number of simulated trials (sample paths), specified as positive scalar integer of `NPERIODS` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.

Data Types: `double`

`DeltaTime` — Time increments between observations

`1` (default) | positive scalar

Time increments between observations, specified as scalar or `NPERIODS`-by-`1` column vector of positive values. `DeltaTime` represents the familiar *dt* found in stochastic differential equations, and determines the times at which `simBySolution` reports the simulated paths of the output state variables. If you do not specify a value for this argument, the default is `1`.

Data Types: `double`

`NSTEPS` — Number of intermediate time steps within each time increment

`1` (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (defined as `DeltaTime`), specified positive scalar integer. `simBySolution` partitions each time increment *dt* into `NSTEPS` subintervals of length *dt*/`NSTEPS`, and refines the simulation by evaluating the simulated state vector at `NSTEPS - 1` intermediate points. Although `simBySolution` does not report the output state vector at these intermediate points, the refinement improves accuracy by allowing the simulation to more closely approximate the underlying continuous-time process. If you do not specify a value for `NSTEPS`, the default is `1`, indicating no intermediate evaluation.

Data Types: `double`

`Antithetic` — Flag that indicates whether antithetic sampling is used

`0` (default) | scalar logical with values `0` or `1`

Flag that indicates whether antithetic sampling is used to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified using a scalar logical with values `0` or `1`. When `Antithetic` is `TRUE` (logical `1`), `simBySolution` performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs:

- Odd trials `(1,3,5,...)` correspond to the primary Gaussian paths

- Even trials `(2,4,6,...)` are the matching antithetic paths of each pair derived by negating the Gaussian draws of the corresponding primary (odd) trial.

If you specify `Antithetic` to be any value other than `TRUE`, `simBySolution` assumes that it is `FALSE` (logical `0`) by default, and does not perform antithetic sampling. When you specify an input noise process (see `Z`), `simBySolution` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of the dependent random noise process
if you do not specify `Z`, `simBySolution` generates correlated Gaussian variates based on the `Correlation` member of the `sde` object (default) | array

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation, specified as an `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` array of dependent random variates. If you specify `Z` as a function, it must return an `NBROWNS`-by-`1` column vector, and you must call it with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double`

`StorePaths` — Flag that indicates how output array `Paths` is stored
`1` (default) | scalar logical with values `0` or `1`

Flag that indicates how output array `Paths` is stored, specified as a scalar logical with values `0` or `1`. If `StorePaths` is `TRUE` (the default value) or is unspecified, `simBySolution` returns `Paths` as a three-dimensional time series array.

If `StorePaths` is `FALSE` (logical `0`), `simBySolution` returns the `Paths` output array as an empty matrix.

Data Types: `logical`

`Processes` — Function or cell array of functions indicating a sequence of end-of-period processes or state vector adjustments
if you do not specify a processing function, `simBySolution` makes no adjustments and performs no processing (default) | function or cell array of functions

Function or cell array of functions indicating a sequence of end-of-period processes or state vector adjustments of the form
$X_t = P(t, X_t)$
specified as function or cell array of functions.

`simBySolution` applies processing functions at the end of each observation period. These functions must accept the current observation time t and the current state vector X_t, and return a state vector that may be an adjustment to the input state. If you specify more than one processing function, `simBySolution` invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and more.

Data Types: `double`

Output Arguments

`Paths` — Simulated paths of correlated state variables

array

Simulated paths of correlated state variables, returned as a `(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` three-dimensional time series array. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t.

When the input flag `StorePaths` = `FALSE`, `simBySolution` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths

vector

Observation times associated with simulated paths, returned as a `(NPERIODS + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Array of dependent random variates used to generate the Brownian motion vector

array

Array of dependent random variates used to generate the Brownian motion vector, returned as a `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` three-dimensional time series array.

Examples

Simulating Equity Markets Using GBM Simulation Methods

Use GBM simulation methods. Separable GBM models have two specific simulation methods:

- An overloaded Euler `simulation` method, designed for optimal performance.
- A `simBySolution` method that provides an approximate solution of the underlying stochastic differential equation, designed for accuracy.

Load the `Data_GlobalIdx2` data set and specify the SDE model as in "Representing Market Models Using SDE Objects" on page 17-34, and the GBM model as in "Representing Market Models Using SDELD, CEV, and GBM Objects" on page 17-37.

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];

returns =  tick2ret(prices);

nVariables  = size(returns,2);
expReturn   = mean(returns);
sigma       = std(returns);
correlation = corrcoef(returns);
t           = 0;
X           = 100;
X           = X(ones(nVariables,1));

F = @(t,X) diag(expReturn)* X;
G = @(t,X) diag(X) * diag(sigma);

SDE = sde(F, G, 'Correlation', ...
    correlation, 'StartState', X);

GBM = gbm(diag(expReturn),diag(sigma), 'Correlation', ...
    correlation, 'StartState', X);
```

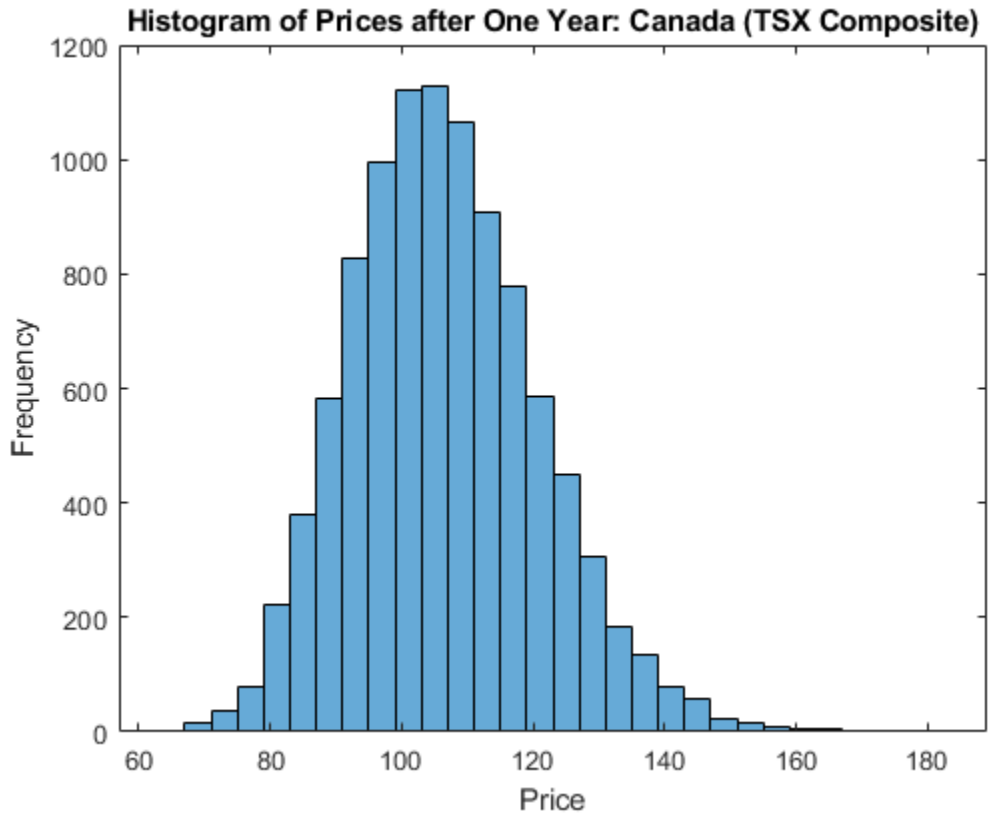
To illustrate the performance benefit of the overloaded Euler approximation method, increase the number of trials to `10000`.

```
nPeriods = 249;      % # of simulated observations
dt       =   1;      % time increment = 1 day
rng(142857,'twister')
[X,T] = simulate(GBM, nPeriods, 'DeltaTime', dt, ...
    'nTrials', 10000);

whos X
```

```
  Name        Size                     Bytes  Class     Attributes

  X         250x6x10000            120000000  double
```

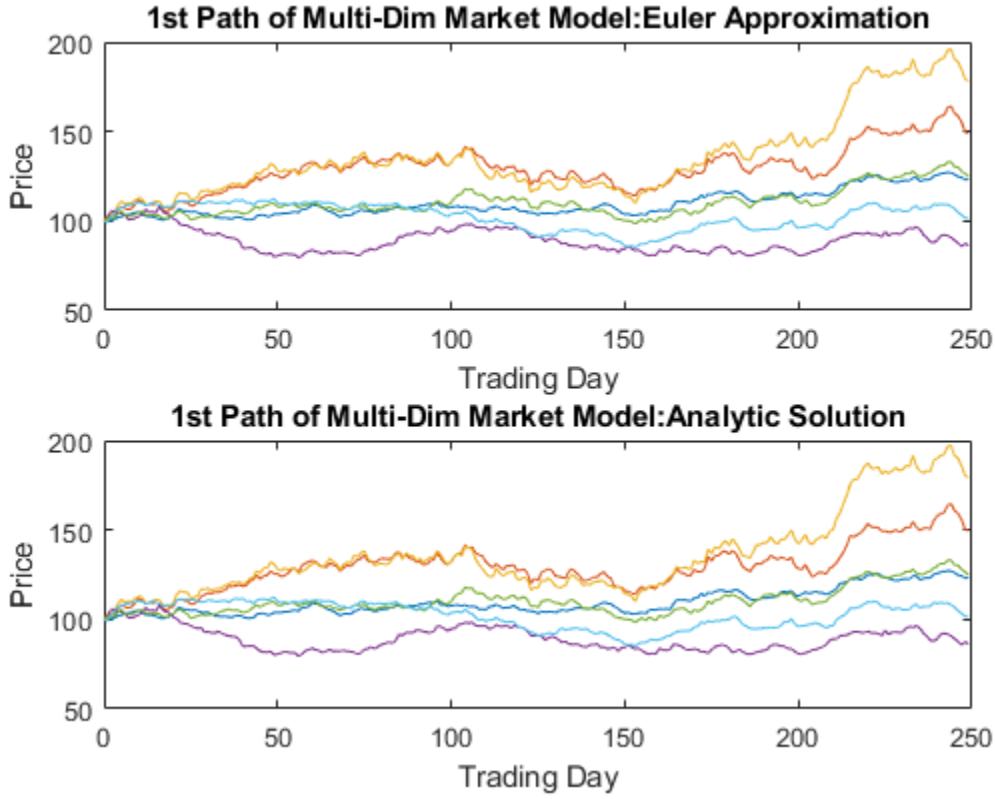
Using this sample size, examine the terminal distribution of Canada's TSX Composite to verify qualitatively the lognormal character of the data.

```
histogram(squeeze(X(end,1,:)), 30), xlabel('Price'), ylabel('Frequency')
title('Histogram of Prices after One Year: Canada (TSX Composite)')
```



Simulate 10 trials of the solution and plot the first trial:

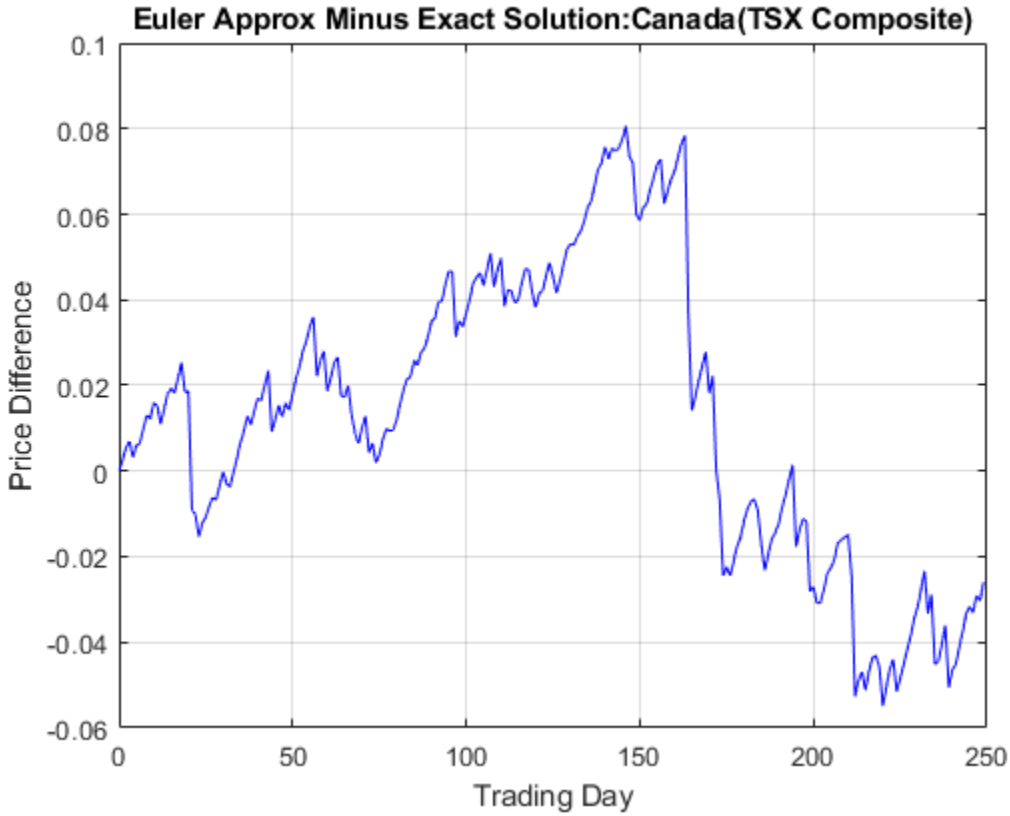
```
rng('default')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);
rng('default')
[X,T] = simBySolution(GBM, nPeriods,...
    'DeltaTime', dt, 'nTrials', 10);
subplot(2,1,1)
plot(T, S(:,:,1)), xlabel('Trading Day'),ylabel('Price')
title('1st Path of Multi-Dim Market Model:Euler Approximation')
subplot(2,1,2)
plot(T, X(:,:,1)), xlabel('Trading Day'),ylabel('Price')
title('1st Path of Multi-Dim Market Model:Analytic Solution')
```



In this example, all parameters are constants, and `simBySolution` does indeed sample the exact solution. The details of a single index for any given trial show that the price paths of the Euler approximation and the exact solution are close, but not identical.

The following plot illustrates the difference between the two methods:

```
subplot(1,1,1)
plot(T, S(:,1,1) - X(:,1,1), 'blue'), grid('on')
xlabel('Trading Day'), ylabel('Price Difference')
title('Euler Approx Minus Exact Solution:Canada(TSX Composite)')
```



The `simByEuler` Euler approximation literally evaluates the stochastic differential equation directly from the equation of motion, for some suitable value of the `dt` time increment. This simple approximation suffers from discretization error. This error can be attributed to the discrepancy between the choice of the `dt` time increment and what in theory is a continuous-time parameter.

The discrete-time approximation improves as `DeltaTime` approaches zero. The Euler method is often the least accurate and most general method available. All models shipped in the simulation suite have this method.

In contrast, the `simBySolution` method provides a more accurate description of the underlying model. This method simulates the price paths by an approximation of the closed-form solution of separable models. Specifically, it applies a Euler approach to a

transformed process, which in general is not the exact solution to this `GBM` model. This is because the probability distributions of the simulated and true state vectors are identical only for piecewise constant parameters.

When all model parameters are piecewise constant over each observation period, the simulated process is exact for the observation times at which the state vector is sampled. Since all parameters are constants in this example, `simBySolution` does indeed sample the exact solution.

For an example of how to use `simBySolution` to optimize the accuracy of solutions, see "Optimizing Accuracy: About Solution Precision and Error" on page 17-78.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

- The input argument `Z` allows you to directly specify the noise generation process. This process takes precedence over the `Correlation` parameter of the `sde` object and the value of the `Antithetic` input flag. If you do not specify a value for `Z`, `simBySolution` generates correlated Gaussian variates, with or without antithetic sampling as requested.
- Gaussian diffusion models, such as HWV, allow negative states. By default, `simBySolution` does nothing to prevent negative states, nor does it guarantee that the model be strictly mean-reverting. Thus, the model may exhibit erratic or explosive growth.
- The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simBySolution` tests the state vector X_t for an all-`NaN` condition. Thus, to signal an early termination of a given trial, all elements of the

state vector X_t must be `NaN`. This test enables a user-defined `Processes` function to signal early termination of a trial, and offers significant performance benefits in some situations (for example, pricing down-and-out barrier options).

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`gbm` | `simByEuler` | `simBySolution` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8

"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

simBySolution

Class: hwv

Simulate approximate solution of diagonal-drift HWV processes

Syntax

```
[Paths,Times,Z] = simBySolution(MDL,NPERIODS)
[Paths,Times,Z] = simBySolution(MDL,NPERIODS,Name,Value)
```

Description

`[Paths,Times,Z] = simBySolution(MDL,NPERIODS)` simulates approximate solution of diagonal-drift for Hull-White/Vasicek Gaussian Diffusion (HWV) processes.

`[Paths,Times,Z] = simBySolution(MDL,NPERIODS,Name,Value)` simulates approximate solution of diagonal-drift for Hull-White/Vasicek Gaussian Diffusion (HWV) processes with additional options specified by one or more `Name,Value` pair arguments.

The `simBySolution` method simulates `NTRIALS` sample paths of `NVARS` correlated state variables, driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Hull-White/Vasicek (HWV) by an approximation of the closed-form solution.

Consider a separable, vector-valued HWV model of the form:
$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$

where:

- X is an *NVARS*-by-`1` state vector of process variables.
- S is an *NVARS*-by-*NVARS* matrix of mean reversion speeds (the rate of mean reversion).
- L is an *NVARS*-by-`1` vector of mean reversion levels (long-run mean or level).
- V is an *NVARS*-by-*NBROWNS* instantaneous volatility rate matrix.

- *W* is an *NBROWNS*-by-`1` Brownian motion vector.

The `simBySolution` method simulates the state vector X_t using an approximation of the closed-form solution of diagonal-drift models.

When evaluating the expressions, `simBySolution` assumes that all model parameters are piecewise-constant over each simulation period.

In general, this is *not* the exact solution to the models, because the probability distributions of the simulated and true state vectors are identical *only* for piecewise-constant parameters.

When parameters are piecewise-constant over each observation period, the simulated process is exact for the observation times at which X_t is sampled.

Input Arguments

`MDL` — Hull-White/Vasicek (HWV) model

`hwv` object

Hull-White/Vasicek (HWV) model, specified as a `hwv` object that is created using the `hwv` constructor.

Data Types: `struct`

`NPERIODS` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of this argument determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`NTRIALS` — Number of simulated trials (sample paths)
`1` (default) | positive scalar integer

Number of simulated trials (sample paths), specified as positive scalar integer of `NPERIODS` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.

Data Types: `double`

`DeltaTime` — Time increments between observations
`1` (default) | positive scalar

Time increments between observations, specified as scalar or `NPERIODS`-by-`1` column vector of positive values. `DeltaTime` represents the familiar *dt* found in stochastic differential equations, and determines the times at which `simBySolution` reports the simulated paths of the output state variables. If you do not specify a value for this argument, the default is `1`.

Data Types: `double`

`NSTEPS` — Number of intermediate time steps within each time increment
`1` (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (defined as `DeltaTime`), specified positive scalar integer. `simBySolution` partitions each time increment *dt* into `NSTEPS` subintervals of length *dt*/`NSTEPS`, and refines the simulation by evaluating the simulated state vector at `NSTEPS - 1` intermediate points. Although `simBySolution` does not report the output state vector at these intermediate points, the refinement improves accuracy by allowing the simulation to more closely approximate the underlying continuous-time process.

If you do not specify a value for `NSTEPS`, the default is `1`, indicating no intermediate evaluation.

Data Types: `double`

`Antithetic` — Flag that indicates whether antithetic sampling is used
`0` (default) | scalar logical with values `0` or `1`

Flag that indicates whether antithetic sampling is used to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified using a scalar logical with values `0` or `1`. When `Antithetic` is `TRUE` (logical `1`),

`simBySolution` performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs:

- Odd trials `(1,3,5,...)` correspond to the primary Gaussian paths
- Even trials `(2,4,6,...)` are the matching antithetic paths of each pair derived by negating the Gaussian draws of the corresponding primary (odd) trial.

If you specify `Antithetic` to be any value other than `TRUE`, `simBySolution` assumes that it is `FALSE` (logical `0`) by default, and does not perform antithetic sampling. When you specify an input noise process (see `Z`), `simBySolution` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of the dependent random noise process

if you do not specify `Z`, `simBySolution` generates correlated Gaussian variates based on the `Correlation` member of the `sde` object (default) | array

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation, specified as an `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` array of dependent random variates. If you specify `Z` as a function, it must return an `NBROWNS`-by-`1` column vector, and you must call it with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double`

`StorePaths` — Flag that indicates how output array `Paths` is stored

`1` (default) | scalar logical with values `0` or `1`

Flag that indicates how output array `Paths` is stored, specified as a scalar logical with values `0` or `1`. If `StorePaths` is `TRUE` (the default value) or is unspecified, `simBySolution` returns `Paths` as a three-dimensional time series array.

If `StorePaths` is `FALSE` (logical `0`), `simBySolution` returns the `Paths` output array as an empty matrix.

Data Types: `logical`

`Processes` — Function or cell array of functions indicating a sequence of end-of-period processes or state vector adjustments
if you do not specify a processing function, `simBySolution` makes no adjustments and performs no processing (default) | function or cell array of functions

Function or cell array of functions indicating a sequence of end-of-period processes or state vector adjustments of the form
$X_t = P(t, X_t)$
specified as function or cell array of functions.

`simBySolution` applies processing functions at the end of each observation period. These functions must accept the current observation time t and the current state vector X_t, and return a state vector that may be an adjustment to the input state.

If you specify more than one processing function, `simBySolution` invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and more.

Data Types: `double`

Output Arguments

`Paths` — Simulated paths of correlated state variables
array

Simulated paths of correlated state variables, returned as a `(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` three-dimensional time series array. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t.

When the input flag `StorePaths` = `FALSE`, `simBySolution` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths
vector

Observation times associated with simulated paths, returned as a `(NPERIODS + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

z — Array of dependent random variates used to generate the Brownian motion vector

array

Array of dependent random variates used to generate the Brownian motion vector, returned as a `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` three-dimensional time series array.

Examples

Use `simBySolution` with an `hwv` Object

Create an `hwv` object to represent the model:
$dX_t = 0.2(0.1 - X_t)dt + 0.05dW_t$.

```
hwv = hwv(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
hwv =

   Class HWV: Hull-White/Vasicek
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

The `simBySolution` method simulates the state vector X_t using an approximation of the closed-form solution of diagonal drift `HWV` models. Each element of the state vector X_t is expressed as the sum of `NBROWNS` correlated Gaussian random draws added to a deterministic time-variable drift.

```
nPeriods = 100
[Paths,Times,Z] = simBySolution(hwv, nPeriods,'nTrials', 10);
```

- "Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 17-30
- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

- The input argument `Z` allows you to directly specify the noise generation process. This process takes precedence over the `Correlation` parameter of the `sde` object and the value of the `Antithetic` input flag. If you do not specify a value for `Z`, `simBySolution` generates correlated Gaussian variates, with or without antithetic sampling as requested.
- Gaussian diffusion models, such as HWV, allow negative states. By default, `simBySolution` does nothing to prevent negative states, nor does it guarantee that the model be strictly mean-reverting. Thus, the model may exhibit erratic or explosive growth.
- The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simBySolution` tests the state vector X_t for an all-`NaN` condition. Thus, to signal an early termination of a given trial, all elements of the state vector X_t must be `NaN`. This test enables a user-defined `Processes` function to signal early termination of a trial, and offers significant performance benefits in some situations (for example, pricing down-and-out barrier options).

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`hwv` | `simByEuler` | `simBySolution` | `simulate`

Topics

“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 17-30
“Simulating Equity Prices” on page 17-34
“Simulating Interest Rates” on page 17-59
“Stratified Sampling” on page 17-70
“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

geom2arith

Geometric to arithmetic moments of asset returns

Syntax

```
[ma,Ca] = geom2arith(mg,Cg)
[ma,Ca] = geom2arith(mg,Cg,t)
```

Arguments

`mg`	Continuously compounded or "geometric" mean of asset returns (positive n-vector).
`Cg`	Continuously compounded or "geometric" covariance of asset returns, a `n`-by-`n` symmetric, positive-semidefinite matrix.
`t`	(Optional) Target period of arithmetic moments in terms of periodicity of geometric moments with default value `1` (positive scalar).

Description

`geom2arith` transforms moments associated with a continuously compounded geometric Brownian motion into equivalent moments associated with a simple Brownian motion with a possible change in periodicity.

`[ma,Ca] = geom2arith(mg,Cg,t)` returns `ma`, arithmetic mean of asset returns over the target period (n-vector), and`Ca`, which is an arithmetric covariance of asset returns over the target period (n-by-n matrix).

Geometric returns over period t_G are modeled as multivariate lognormal random variables with moments

$E[\mathrm{Y}] = 1 + \mathrm{m}_G$

and

$$\mathrm{cov}(Y) = C_G$$

Arithmetic returns over period t_A are modeled as multivariate normal random variables with moments

$$E[X] = m_A$$

$$\mathrm{cov}(X) = C_A$$

Given $t = t_A / t_G$, the transformation from geometric to arithmetic moments is

$$C_{A_{ij}} = t \log\left(1 + \frac{C_{G_{ij}}}{(1 + m_{G_i})(1 + m_{G_j})} \right)$$

$$m_{A_i} = t \log(1 + m_{G_i}) - \frac{1}{2} C_{A_{ii}}$$

For $i,j = 1,\ldots, n$.

Note If $t = 1$, then $\mathbf{X} = \log(\mathbf{Y})$.

This function requires that the input mean must satisfy `1 + mg > 0` and that the input covariance `Cg` must be a symmetric, positive, semidefinite matrix.

The functions `geom2arith` and `arith2geom` are complementary so that, given `m`, `C`, and `t`, the sequence

```
[ma,Ca] = geom2arith(m,C,t);
[mg,Cg] = arith2geom(ma,Ca,1/t);
```

yields `mg` = `m` and `Cg` = `C`.

Examples

Example 1. Given geometric mean `m` and covariance `C` of monthly total returns, obtain annual arithmetic mean `ma` and covariance `Ca`. In this case, the output period (1 year) is 12 times the input period (1 month) so that `t = 12` with

```
[ma, Ca] = geom2arith(m, C, 12);
```

Example 2. Given annual geometric mean `m` and covariance `C` of asset returns, obtain monthly arithmetic mean `ma` and covariance `Ca`. In this case, the output period (1 month) is 1/12 times the input period (1 year) so that `t = 1/12` with

```
[ma, Ca] = geom2arith(m, C, 1/12);
```

Example 3. Given geometric means `m` and standard deviations `s` of daily total returns (derived from 260 business days per year), obtain annualized arithmetic mean `ma` and standard deviations `sa` with

```
[ma, Ca] = geom2arith(m, diag(s .^2), 260);
sa = sqrt(diag(Ca));
```

Example 4. Given geometric mean `m` and covariance `C` of monthly total returns, obtain quarterly arithmetic return moments. In this case, the output is 3 of the input periods so that `t = 3` with

```
[ma, Ca] = geom2arith(m, C, 3);
```

Example 5. Given geometric mean `m` and covariance `C` of 1254 observations of daily total returns over a 5-year period, obtain annualized arithmetic return moments. Since the periodicity of the geometric data is based on 1254 observations for a 5-year period, a 1-year period for arithmetic returns implies a target period of `t = 1254/5` so that

```
[ma, Ca] = geom2arith(m, C, 1254/5);
```

See Also

`arith2geom`

Topics

"Data Transformation and Frequency Conversion" on page 12-12

Introduced before R2006a

getAssetMoments

Obtain mean and covariance of asset returns from Portfolio object

Use the `getAssetMoments` function with a `Portfolio` object to obtain mean and covariance of asset returns.

For details on the workflow, see "Portfolio Object Workflow" on page 4-21.

Syntax

```
[AssetMean,AssetCovar] = getAssetMoments(obj)
```

Description

`[AssetMean,AssetCovar] = getAssetMoments(obj)` obtains mean and covariance of asset returns for a `Portfolio` object.

Examples

Obtain Asset Moment Properties for a Portfolio Object

Given the mean and covariance of asset returns in the variables `m` and `C`, the asset moment properties can be set and then obtained using the `getAssetMoments` function:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

p = Portfolio;
p = setAssetMoments(p, m, C);
[assetmean, assetcovar] = getAssetMoments(p)
```

```
assetmean =

    0.0042
    0.0083
    0.0100
    0.0150


assetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-48
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object. For more information on creating a portfolio object, see

- `Portfolio`

Output Arguments

`AssetMean` — Mean of asset returns

vector

Mean of asset returns, returned as a vector.

`AssetCovar` — Covariance of asset returns

matrix

Covariance of asset returns, returned as a matrix.

Tips

You can also use dot notation to obtain the mean and covariance of asset returns from a Portfolio object:

```
[AssetMean, AssetCovar] = obj.getAssetMoments;
```

See Also

`setAssetMoments`

Topics

"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-48
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

getBounds

Obtain bounds for portfolio weights from portfolio object

Use the `getBounds` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain bounds for portfolio weights from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[LowerBound,UpperBound] = getBounds(obj)
```

Description

`[LowerBound,UpperBound] = getBounds(obj)` obtains bounds for portfolio weights from portfolio objects.

Examples

Obtain Values for Lower and Upper Bounds for a Portfolio Object

Given portfolio `p` with the default constraints set, obtain the values for `LowerBound` and `UpperBound`.

```
p = Portfolio;
p = setDefaultConstraints(p, 5);
[LowerBound, UpperBound] = getBounds(p)
```

```
LowerBound =

     0
     0
```

```
     0
     0
     0


UpperBound =

     []
```

Obtain Values for Lower and Upper Bounds for a PortfolioCVaR Object

Given a PortfolioCVaR object `p` with the default constraints set, obtain the values for `LowerBound` and `UpperBound`.

```
p = PortfolioCVaR;
p = setDefaultConstraints(p, 5);
[LowerBound, UpperBound] = getBounds(p)
```

```
LowerBound =

     0
     0
     0
     0
     0


UpperBound =

     []
```

Obtain Values for Lower and Upper Bounds for a PortfolioMAD Object

Given a PortfolioMAD object `p` with the default constraints set, obtain the values for `LowerBound` and `UpperBound`.

```
p = PortfolioMAD;
p = setDefaultConstraints(p, 5);
[LowerBound, UpperBound] = getBounds(p)
```

```
LowerBound =

     0
     0
     0
     0
     0


UpperBound =

     []
```

- "Working with Bound Constraints Using Portfolio Object" on page 4-72
- "Working with Bound Constraints Using PortfolioCVaR Object" on page 5-68
- "Working with Bound Constraints Using PortfolioMAD Object" on page 6-66
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`LowerBound` — Lower-bound weight for each asset

vector

Lower-bound weight for each asset, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`). For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`UpperBound` — Upper-bound weight for each asset

vector

Upper-bound weight for each asset, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`). For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to obtain bounds for portfolio weights from portfolio objects.

```
[LowerBound, UpperBound] = obj.getBounds;
```

See Also

`setBounds`

Topics

"Working with Bound Constraints Using Portfolio Object" on page 4-72
"Working with Bound Constraints Using PortfolioCVaR Object" on page 5-68
"Working with Bound Constraints Using PortfolioMAD Object" on page 6-66
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getBudget

Obtain budget constraint bounds from portfolio object

Use the `getBudget` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain budget constraint bounds from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[LowerBudget,UpperBudget] = getBudget(obj)
```

Description

`[LowerBudget,UpperBudget] = getBudget(obj)` obtains budget constraint bounds from portfolio objects.

Examples

Obtain Values for Lower and Upper Budgets for a Portfolio Object

Given portfolio `p` with the default constraints set, obtain the values for `LowerBudget` and `UpperBudget`.

```
p = Portfolio;
p = setDefaultConstraints(p, 5);
[LowerBudget, UpperBudget] = getBudget(p)
```

```
LowerBudget = 1
```

```
UpperBudget = 1
```

Obtain Values for Lower and Upper Budgets for a PortfolioCVaR Object

Given a PortfolioCVaR object `p` with the default constraints set, obtain the values for `LowerBudget` and `UpperBudget`.

```
p = PortfolioCVaR;
p = setDefaultConstraints(p, 5);
[LowerBudget, UpperBudget] = getBudget(p)
```

```
LowerBudget = 1
```

```
UpperBudget = 1
```

Obtain Values for Lower and Upper Budgets for a PortfolioMAD Object

Given a PortfolioMAD object `p` with the default constraints set, obtain the values for `LowerBudget` and `UpperBudget`.

```
p = PortfolioMAD;
p = setDefaultConstraints(p, 5);
[LowerBudget, UpperBudget] = getBudget(p)
```

```
LowerBudget = 1
```

```
UpperBudget = 1
```

- “Working with Budget Constraints Using Portfolio Object” on page 4-75
- “Working with Budget Constraints Using PortfolioCVaR Object” on page 5-71
- “Working with Budget Constraints Using PortfolioMAD Object” on page 6-69
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`LowerBudget` — Lower-bound weight for each asset
scalar

Lower bound for budget constraint, returned as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`UpperBudget` — Upper bound for budget constraint
scalar

Upper bound for budget constraint, returned as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to obtain the budget constraint bounds from portfolio objects.

```
[LowerBudget, UpperBudget] = obj.getBudget;
```

See Also

`setBudget`

Topics

"Working with Budget Constraints Using Portfolio Object" on page 4-75
"Working with Budget Constraints Using PortfolioCVaR Object" on page 5-71
"Working with Budget Constraints Using PortfolioMAD Object" on page 6-69
"Portfolio Optimization Examples" on page 4-147

"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getCosts

Obtain buy and sell transaction costs from portfolio object

Use the `getCosts` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain buy and sell transaction costs from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[BuyCost,SellCost] = getCosts(obj)
```

Description

`[BuyCost,SellCost] = getCosts(obj)` obtains buy and sell transaction costs from portfolio objects.

Examples

Obtain Buy and Sell Costs for a Portfolio Object

Given portfolio `p` with the costs set, obtain the values for `BuyCost` and `SellCost`.

```
p = Portfolio;
p = setCosts(p, 0.001, 0.001, 5);
[BuyCost, SellCost] = getCosts(p)
```

```
BuyCost = 1.0000e-03
```

```
SellCost = 1.0000e-03
```

Obtain Buy and Sell Costs for a PortfolioCVaR Object

Given a PortfolioCVaR object `p` with the costs set, obtain the values for `BuyCost` and `SellCost`.

```
p = PortfolioCVaR;
p = setCosts(p, 0.001, 0.001, 5);
[BuyCost, SellCost] = getCosts(p)
```

```
BuyCost = 1.0000e-03
```

```
SellCost = 1.0000e-03
```

Obtain Buy and Sell Costs for a PortfolioMAD Object

Given a PortfolioMAD object `p` with the costs set, obtain the values for `BuyCost` and `SellCost`.

```
p = PortfolioMAD;
p = setCosts(p, 0.001, 0.001, 5);
[BuyCost, SellCost] = getCosts(p)
```

```
BuyCost = 1.0000e-03
```

```
SellCost = 1.0000e-03
```


- "Working with Transaction Costs" on page 4-62
- "Working with Transaction Costs" on page 5-58
- "Working with Transaction Costs" on page 6-56
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`BuyCost` — Proportional transaction cost to purchase each asset

vector

Proportional transaction cost to purchase each asset, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`SellCost` — Proportional transaction cost to sell each asset

vector

Proportional transaction cost to sell each asset, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to obtain the buy and sell transaction costs from portfolio objects.

```
[BuyCost, SellCost] = obj.getCosts;
```

See Also

`setCosts`

Topics

“Working with Transaction Costs” on page 4-62
“Working with Transaction Costs” on page 5-58
“Working with Transaction Costs” on page 6-56
“Portfolio Optimization Examples” on page 4-147
“Portfolio Set for Optimization Using Portfolio Object” on page 4-10
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-10

“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-10

Introduced in R2011a

getEquality

Obtain equality constraint arrays from portfolio object

Use the `getEquality` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain equality constraint arrays from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[AEquality,bEquality] = getEquality(obj)
```

Description

`[AEquality,bEquality] = getEquality(obj)` obtains equality constraint arrays from portfolio objects.

Examples

Obtain Equality Constraints for a Portfolio Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are exactly 50% of your portfolio. Given a Portfolio object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio;
p = setEquality(p, A, b);
[AEquality, bEquality] = getEquality(p)

AEquality =
```

```
     1     1     1     0     0

bEquality = 0.5000
```

Obtain Equality Constraints for a PortfolioCVaR Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioCVaR object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setEquality(p, A, b);
[AEquality, bEquality] = getEquality(p)

AEquality =

     1     1     1     0     0


bEquality = 0.5000
```

Obtain Equality Contraintsfor a PortfolioMAD Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioMAD object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setEquality(p, A, b);
[AEquality, bEquality] = getEquality(p)

AEquality =

     1     1     1     0     0
```

```
bEquality = 0.5000
```

- “Working with Linear Equality Constraints Using Portfolio Object” on page 4-86
- “Working with Linear Equality Constraints Using PortfolioCVaR Object” on page 5-82
- “Working with Linear Equality Constraints Using PortfolioMAD Object” on page 6-79
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`AEquality` — Matrix to form linear equality constraints

matrix

Matrix to form linear equality constraints, returned as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`bEquality` — Vector to form linear equality constraints

vector

Vector to form linear equality constraints, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to obtain the equality constraint arrays from portfolio objects.

```
[AEquality, bEquality] = obj.getEquality;
```

See Also

`setEquality`

Topics

"Working with Linear Equality Constraints Using Portfolio Object" on page 4-86
"Working with Linear Equality Constraints Using PortfolioCVaR Object" on page 5-82
"Working with Linear Equality Constraints Using PortfolioMAD Object" on page 6-79
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getGroupRatio

Obtain group ratio constraint arrays from portfolio object

Use the `getGroupRatio` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain group ratio constraint arrays from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj)
```

Description

`[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj)` obtains equality constraint arrays from portfolio objects.

Examples

Obtain Group Ratio Constraints for a Portfolio Object

Suppose you want to make sure that the ratio of financial to nonfinancial companies in your portfolios never goes above 50%. Assume you have 6 assets with 3 financial companies (assets 1-3) and 3 nonfinancial companies (assets 4-6). After setting group ratio constraints, obtain the values for `GroupA`, `GroupB`, `LowerRatio`, and `UpperRatio`.

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);
[GroupA, GroupB, LowerRatio, UpperRatio] = getGroupRatio(p)
```

```
GroupA =

     1     1     1     0     0     0


GroupB =

     0     0     0     1     1     1


LowerRatio =

     []

UpperRatio = 0.5000
```

Obtain Group Ratio Constraints for a PortfolioCVaR Object

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolios never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). After setting group ratio constraints, obtain the values for `GroupA`, `GroupB`, `LowerRatio`, and `UpperRatio`.

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);
[GroupA, GroupB, LowerRatio, UpperRatio] = getGroupRatio(p)
```

```
GroupA =

     1     1     1     0     0     0


GroupB =

     0     0     0     1     1     1


LowerRatio =

     []
```

```
UpperRatio = 0.5000
```

Obtain Group Ratio Constraints for a PortfolioMAD Object

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolios never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinancial companies (assets 4-6). After setting group ratio constraints, obtain the values for `GroupA`, `GroupB`, `LowerRatio`, and `UpperRatio`.

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);
[GroupA, GroupB, LowerRatio, UpperRatio] = getGroupRatio(p)
```

```
GroupA =

     1     1     1     0     0     0


GroupB =

     0     0     0     1     1     1


LowerRatio =

     []

UpperRatio = 0.5000
```

- “Working with Group Ratio Constraints Using Portfolio Object” on page 4-82
- “Working with Group Constraints Using PortfolioCVaR Object” on page 5-74
- “Working with Group Constraints Using PortfolioMAD Object” on page 6-71
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`GroupA` — Matrix that forms base groups for comparison

matrix

Matrix that forms base groups for comparison, returned as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`GroupB` — Matrix that forms comparison groups

matrix

Matrix that forms comparison groups, returned as a matrix `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`LowerRatio` — Lower bound for ratio of `GroupB` groups to `GroupA` groups

vector

Lower bound for ratio of `GroupB` groups to `GroupA` groups, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`UpperRatio` — Upper bound for ratio of `GroupB` groups to `GroupA` groups

vector

Upper bound for ratio of `GroupB` groups to `GroupA` groups, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to obtain group ratio constraint arrays from portfolio objects.

```
[GroupA, GroupB, LowerRatio, UpperRatio] = obj.getGroupRatio;
```

See Also

`setGroupRatio`

Topics

"Working with Group Ratio Constraints Using Portfolio Object" on page 4-82
"Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
"Working with Group Constraints Using PortfolioMAD Object" on page 6-71
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getGroups

Obtain group constraint arrays from portfolio object

Use the `getGroups` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain group constraint arrays from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[GroupMatrix,LowerGroup,UpperGroup] = getGroups(obj)
```

Description

`[GroupMatrix,LowerGroup,UpperGroup] = getGroups(obj)` obtains group constraint arrays from portfolio objects.

Examples

Obtain Group Constraints for a Portfolio Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute no more than 30% of your portfolio. Given a Portfolio object `p` with the group constraints set, obtain the values for `GroupMatrix`, `LowerGroup`, and `UpperGroup`.

```
G = [ true true true false false ];
p = Portfolio;
p = setGroups(p, G, [], 0.3);
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(p)
```

```
GroupMatrix =
```

```
     1     1     1     0     0

LowerGroup =

     []

UpperGroup = 0.3000
```

Obtain Group Constraints for a PortfolioCVaR Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a PortfolioCVaR object `p` with the group constraints set, obtain the values for `GroupMatrix`, `LowerGroup`, and `UpperGroup`.

```
G = [ true true true false false ];
p = PortfolioCVaR;
p = setGroups(p, G, [], 0.3);
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(p)

GroupMatrix =

     1     1     1     0     0


LowerGroup =

     []

UpperGroup = 0.3000
```

Obtain Group Constraints for a PortfolioMAD Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a PortfolioMAD object `p` with the group constraints set, obtain the values for `GroupMatrix`, `LowerGroup`, and `UpperGroup`.

```
G = [ true true true false false ];
p = PortfolioMAD;
p = setGroups(p, G, [], 0.3);
[GroupMatrix, LowerGroup, UpperGroup] = getGroups(p)

GroupMatrix =

     1     1     1     0     0


LowerGroup =

     []

UpperGroup = 0.3000
```

- "Working with Group Constraints Using Portfolio Object" on page 4-78
- "Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
- "Working with Group Constraints Using PortfolioMAD Object" on page 6-71
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`GroupMatrix` — Group constraint matrix

matrix

Group constraint matrix, returned as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`LowerGroup` — Lower bound for group constraints
vector

Lower bound for group constraints, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`UpperGroup` — Upper bound for group constraints
vector

Upper bound for group constraints, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to obtain the group constraint arrays from portfolio objects.

```
[GroupMatrix, LowerGroup, UpperGroup] = obj.getGroups;
```

See Also

`setGroups`

Topics

"Working with Group Constraints Using Portfolio Object" on page 4-78
"Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
"Working with Group Constraints Using PortfolioMAD Object" on page 6-71
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getScenarios

Obtain scenarios from portfolio object

Use the `getScenarios` function with a `PortfolioCVaR` or `PortfolioMAD` objects to obtain scenarios.

For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
Y = getScenarios(obj)
```

Description

`Y = getScenarios(obj)` obtains scenarios for `PortfolioCVaR` or `PortfolioMAD` objects.

Examples

Obtain Scenarios for a CVaR Portfolio Object

For a given PortfolioCVaR object `p`, display the defined scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

rng(11);
```

```
rng(11);

AssetScenarios = mvnrnd(m, C, 10);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

Y = getScenarios(p)

Y =

   -0.0056    0.0440    0.1186    0.0488
   -0.0368   -0.0753    0.0087    0.1124
    0.0025    0.0856    0.0484    0.1404
    0.0318    0.0826    0.0377    0.0404
    0.0013   -0.0561   -0.1466   -0.0621
    0.0035    0.0310   -0.0183    0.1225
   -0.0519   -0.1634   -0.0526    0.1528
    0.0029   -0.1163   -0.0627   -0.0760
    0.0192   -0.0182   -0.1243   -0.1346
    0.0440    0.0189    0.0098    0.0821
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

Obtain Scenarios for a MAD Portfolio Object

For a given PortfolioMAD object `p`, display the defined scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
```

```
rng(11);

AssetScenarios = mvnrnd(m, C, 10);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

Y = getScenarios(p)

Y =

   -0.0056    0.0440    0.1186    0.0488
   -0.0368   -0.0753    0.0087    0.1124
    0.0025    0.0856    0.0484    0.1404
    0.0318    0.0826    0.0377    0.0404
    0.0013   -0.0561   -0.1466   -0.0621
    0.0035    0.0310   -0.0183    0.1225
   -0.0519   -0.1634   -0.0526    0.1528
    0.0029   -0.1163   -0.0627   -0.0760
    0.0192   -0.0182   -0.1243   -0.1346
    0.0440    0.0189    0.0098    0.0821
```

The function `rng(`*seed*`)` resets the random number generator to produce the documented results. It is not necessary to reset the random number generator to simulate scenarios.

- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` or `PortfolioMAD` object.

For more information on creating a `PortfolioCVaR` or `PortfolioMAD` object, see

- `PortfolioCVaR`

- `PortfolioMAD`

Output Arguments

`Y` — Scenarios matrix

matrix

Scenarios matrix, returned as a `NumScenarios`-by-`NumAssets` matrix for a `PortfolioCVaR` or `PortfolioMAD` object.

Tips

You can also use dot notation to obtain scenarios from a `PortfolioCVaR` or `PortfolioMAD` object.

```
Y = obj.getScenarios;
```

See Also

`rng` | `setScenarios`

Topics

“Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

getfield

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 character vectors or a cell of a date character vector 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`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

getInequality

Obtain inequality constraint arrays from portfolio object

Use the `getInequality` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain inequality constraint arrays from portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[AInequality,bInequality] = getInequality(obj)
```

Description

`[AInequality,bInequality] = getInequality(obj)` obtains equality constraint arrays from portfolio objects.

Examples

Obtain Inequality Constraints for a Portfolio Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given a Portfolio object `p`, set the linear inequality constraints and then obtain values for `AInequality` and `bInequality`.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = Portfolio;
p = setInequality(p, A, b);
[AInequality, bInequality] = getInequality(p)
```

```
AInequality =
```

```
     1     1     1     0     0

bInequality = 0.5000
```

Obtain Inequality Constraints for a PortfolioCVaR Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 50% of your portfolio. Given a PortfolioCVaR object `p`, set the linear inequality constraints and then obtain values for `AInequality` and `bInequality`.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setInequality(p, A, b);
[AInequality, bInequality] = getInequality(p)

AInequality =

     1     1     1     0     0


bInequality = 0.5000
```

Obtain Inequality Constraints for a PortfolioMAD Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 50% of your portfolio. Given a PortfolioMAD object `p`, set the linear inequality constraints and then obtain values for `AInequality` and `bInequality`.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setInequality(p, A, b);
[AInequality, bInequality] = getInequality(p)

AInequality =
```

```
     1     1     1     0     0

bInequality = 0.5000
```

- "Working with Linear Inequality Constraints Using Portfolio Object" on page 4-89
- "Working with Linear Inequality Constraints Using PortfolioCVaR Object" on page 5-85
- "Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-82
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`AInequality` — Matrix to form linear inequality constraints

matrix

Matrix to form linear inequality constraints, returned as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`bInequality` — Vector to form linear inequality constraints

vector

Vector to form linear inequality constraints, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Tips

You can also use dot notation to obtain the inequality constraint arrays from portfolio objects.

```
[AInequality, bInequality] = obj.getInequality;
```

See Also

`setInequality`

Topics

"Working with Linear Inequality Constraints Using Portfolio Object" on page 4-89
"Working with Linear Inequality Constraints Using PortfolioCVaR Object" on page 5-85
"Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-82
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getOneWayTurnover

Obtain one-way turnover constraints from portfolio object

Use the `getOneWayTirnover` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to obtain one-way turnover constraints from portfolio objects.

For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-21, “PortfolioCVaR Object Workflow” on page 5-20, and “PortfolioMAD Object Workflow” on page 6-19.

Syntax

```
[BuyTurnover,SellTurnover] = getOneWayTurnover(obj)
```

Description

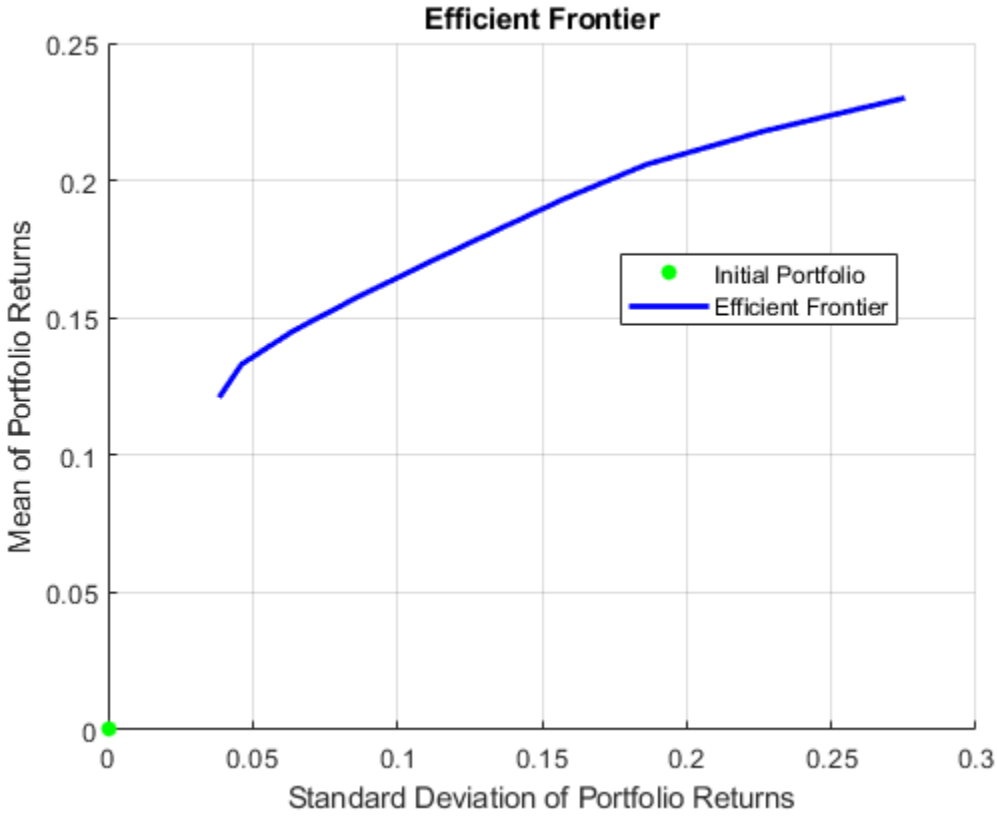
`[BuyTurnover,SellTurnover] = getOneWayTurnover(obj)` obtain one-way turnover constraints from portfolio objects.

Examples

Obtain One-Way Turnover Costs for a Portfolio Object

Set one-way turnover costs.

```
p = Portfolio('AssetMean',[0.1, 0.2, 0.15], 'AssetCovar',...
[ 0.005, -0.010,  0.004; -0.010,  0.040, -0.002;  0.004, -0.002,  0.023]);
p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0);    %130-30 portfolio
plotFrontier(p);
```



Obtain one-way turnover costs.

```
[BuyTurnover,SellTurnover] = getOneWayTurnover(p)
```

```
BuyTurnover = 1.3000
```

```
SellTurnover = 0.3000
```

Obtain One-Way Turnover Costs for a PortfolioCVaR Object

Set one-way turnover costs and obtain the buy and sell turnover values.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0); %130-30 portfolio

[BuyTurnover,SellTurnover] = getOneWayTurnover(p)

BuyTurnover = 1.3000

SellTurnover = 0.3000
```

Obtain One-Way Turnover Costs for a PortfolioMAD Object

Set one-way turnover costs and obtain the buy and sell turnover values.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

```
p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0); %130-30 portfolio

[BuyTurnover,SellTurnover] = getOneWayTurnover(p)

BuyTurnover = 1.3000

SellTurnover = 0.3000
```

- "Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-96
- "Working with One-Way Turnover Constraints Using PortfolioCVaR Object" on page 5-92
- "Working with One-Way Turnover Constraints Using PortfolioMAD Object" on page 6-88
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Output Arguments

`BuyTurnover` — Turnover constraint on purchases

scalar

Turnover constraint on purchases, returned as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`SellTurnover` — Turnover constraint on sales
scalar

Turnover constraint on sales, returned as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Definitions

One-way Turnover Constraint

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales.

The constraints take the form

$$1^T \max\{0, x - x_0\} \le \tau_B$$

$$1^T \max\{0, x_0 - x\} \le \tau_S$$

with

- x — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper-bound for turnover constraint on purchases (scalar)
- τ_S — Upper-bound for turnover constraint on sales (scalar)

Specify one-way turnover constraints using the following properties in a supported portfolio object: `BuyTurnover` for τ_B, `SellTurnover` for τ_S, and `InitPort` for x_0.

Note The average turnover constraint (which is set using `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Tips

You can also use dot notation to get the one-way turnover constraint for portfolio objects.

```
[BuyTurnover,SellTurnover] = obj.getOneWayTurnover
```

See Also

`setOneWayTurnover` | `setTurnover`

Topics

"Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-96
"Working with One-Way Turnover Constraints Using PortfolioCVaR Object" on page 5-92
"Working with One-Way Turnover Constraints Using PortfolioMAD Object" on page 6-88
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

getnameidx

Find name in list

Syntax

```
nameidx = getnameidx(list,name)
```

Arguments

`list`	Cell array of name character vectors.
`name`	Character vector or cell array of name character vectors.

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` character vectors 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 are 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 (character vectors) only. It does not find multiple occurrences of a name or a list of names within `list`.

Examples

Given

```
poultry = {'duck', 'chicken'}
animals = {'duck', 'cow', 'sheep', 'horse', 'chicken'}
nameidx = getnameidx(animals, poultry)

ans =
    1   5
```

Given

```
poultry = {'duck', 'goose', 'chicken'}
animals = {'duck', 'cow', 'sheep', 'horse', 'chicken'}
nameidx = getnameidx(animals, poultry)

ans =
    1   0   5
```

See Also

`strcmp` | `strfind`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

heston class

Heston model

Description

`heston` objects derive from the `sdeddo` (SDE from drift and diffusion objects) class. Use `heston` objects to simulate sample paths of two state variables. Each state variable is driven by a single Brownian motion source of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic volatility processes.

Heston models are bivariate composite models. Each Heston model consists of two coupled univariate models:

- A geometric Brownian motion (`gbm`) model with a stochastic volatility function.

  $$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

  This model usually corresponds to a price process whose volatility (variance rate) is governed by the second univariate model.
- A Cox-Ingersoll-Ross (`cir`) square root diffusion model.

  $$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

  This model describes the evolution of the variance rate of the coupled GBM price process.

Construction

`heston = heston(Return,Speed,Level,Volatility)` constructs a default `heston` object.

`heston = heston(Return,Speed,Level,Volatility,Name,Value)` constructs a `heston` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `heston` object, see `heston`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Return` — `Return` represents the parameter μ

array or deterministic function of time or deterministic function of time and state

`Return` represents the parameter μ, specified as an array or deterministic function of time.

If you specify `Return` as an array, it must be an `NVARS`-by-`NVARS` matrix representing the expected (mean) instantaneous rate of return.

As a deterministic function of time, when `Return` is called with a real-valued scalar time `t` as its only input, `Return` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Return` as a function of time and state, it must return an `NVARS`-by-`NVARS` matrix when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Level` — `Level` represents the parameter L
array or deterministic function of time or deterministic function of time and state

`Level` represents the parameter L, specified as an array or deterministic function of time.

If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels.

As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1` column vector. If you specify `Level` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of reversion levels when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Speed` — `Speed` represents the parameter S
array or deterministic function of time or deterministic function of time and state

`Speed` represents the parameter S, specified as an array or deterministic function of time.

If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate at which the state vector reverts to its long-run average `Level`).

As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed` as a function of time and state, it calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Volatility` — `Volatility` represents the instantaneous volatility of the CIR stochastic variance model
scalar or deterministic function of time or deterministic function of time and state

`Volatility` (often called the *volatility of volatility* or *volatility of variance*) represents the instantaneous volatility of the CIR stochastic variance model, specified as a scalar or deterministic function of time.

If you specify`Volatility` as a scalar, it represents the instantaneous volatility of the CIR stochastic variance model.

As a deterministic function of time, when `Volatility` is called with a real-valued scalar time `t` as its only input, `Volatility` must produce a scalar. If you specify it as a function time and state, `Volatility` generates a scalar when invoked with two inputs:

- A real-valued scalar observation time t.
- A `2`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `heston`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t).

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

SetAccess	private

`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t,X_t) = D(t,X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the `gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
if you do not specify a value for Simulation, the default method is simulation by Euler approximation (simByEuler) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

SetAccess	public
GetAccess	public

Data Types: function_handle

Methods

Inherited Methods

The following methods are inherited from the sde class.

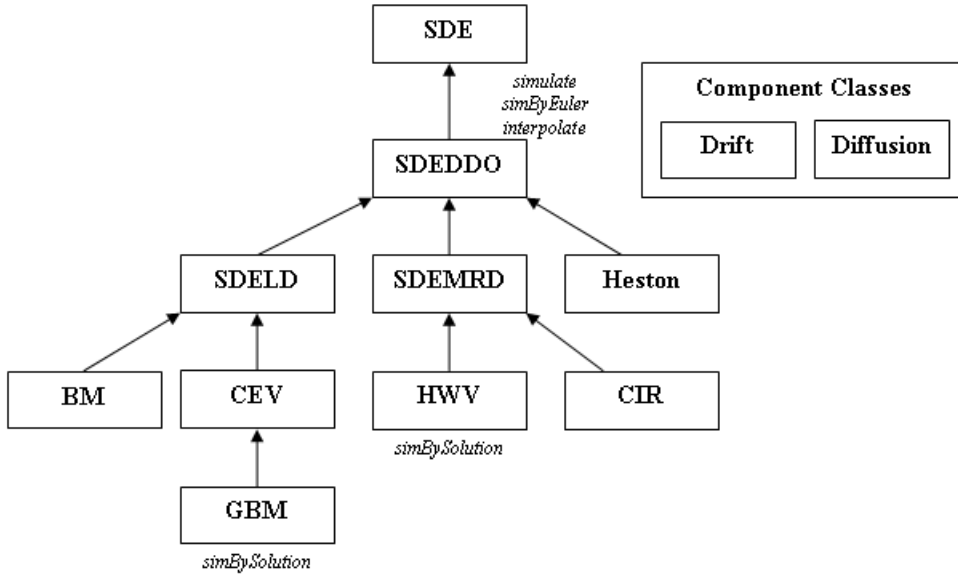
interpolate

simulate

simByEuler

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a heston Object

The Heston (`heston`) class derives directly from SDE from Drift and Diffusion (`sdeddo`). Each Heston model is a bivariate composite model, consisting of two coupled univariate models:

$$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

$$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

Create a `heston` object to represent the model:

$$dX_{1t} = 0.1X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

$$dX_{2t} = 0.2[0.1 - X_{2t}]dt + 0.05\sqrt{X_{2t}}dW_{2t}$$

```
obj = heston (0.1, 0.2, 0.1, 0.05)  % (Return, Speed, Level, Volatility)
```

```
obj =
   Class HESTON: Heston Bivariate Stochastic Volatility
   -----------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   -----------------------------------------------------
      StartTime: 0
     StartState: 1 (2x1 double array)
    Correlation: 2x2 diagonal double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.1
          Speed: 0.2
          Level: 0.1
     Volatility: 0.05
```

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Stratified Sampling” on page 17-70
- “Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19
- “Linear Drift Models” on page 17-23
- “Parametric Models” on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `heston` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering.* New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeddo` | `simByEuler` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

heston

Construct Heston model

Syntax

```
heston = heston(Return, Speed, Level, Volatility)

heston = heston(Return, Speed, Level, Volatility, 'Name1', Value1,
'Name2', Value2, ...)
```

Class

`heston`

Description

This constructor creates and displays `heston` objects, which derive from the`sdeddo` (SDE from drift and diffusion objects) class. Use `heston` objects to simulate sample paths of two state variables. Each state variable is driven by a single Brownian motion source of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic volatility processes.

Heston models are bivariate composite models. Each Heston model consists of two coupled univariate models:

- A geometric Brownian motion (`gbm`) model with a stochastic volatility function.

  $$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$$

  This model usually corresponds to a price process whose volatility (variance rate) is governed by the second univariate model.
- A Cox-Ingersoll-Ross (`cir`) square root diffusion model.

  $$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

This model describes the evolution of the variance rate of the coupled GBM price process.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Return`	If you specify `Return` as a scalar, it represents the expected (mean) instantaneous rate of return of the univariate GBM price model. As a deterministic function of time, when `Return` is called with a real-valued scalar time `t` as its only input, `Return` must produce a scalar. If you specify it as a function of time and state, `Return` calculates the instantaneous rate of return of the GBM price model. This function generates a scalar when invoked with two inputs: • A real-valued scalar observation time *t*. • A `2`-by-`1` bivariate state vector X_t.

`Speed`	If you specify `Speed` as a scalar, it represents the mean-reversion speed of the univariate CIR stochastic variance model (the speed at which the CIR variance reverts to its long-run average level). As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce a scalar. If you specify it as a function time and state, `Speed` calculates the speed of mean reversion of the CIR variance model. This function generates a scalar when invoked with two inputs: • A real-valued scalar observation time *t*. • A `2`-by-`1` state vector X_t.
`Level`	If you specify `Level` as a scalar, it represents the reversion level of the univariate CIR stochastic variance model. As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce a scalar. If you specify it as a function time and state, `Level` calculates the reversion level of the CIR variance model. This function generates a scalar when invoked with two inputs: • A real-valued scalar observation time *t*. • A `2`-by-`1` state vector X_t.
`Volatility`	If you specify `Volatility` as a scalar, it represents the instantaneous volatility of the CIR stochastic variance model, often called the *volatility of volatility* or *volatility of variance*. As a deterministic function of time, when `Volatility` is called with a real-valued scalar time `t` as its only input, `Volatility` must produce a scalar. If you specify it as a function time and state, `Volatility` generates a scalar when invoked with two inputs: • A real-valued scalar observation time *t*. • A `2`-by-`1` state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of any of these input arguments, each argument is specified as a positive value.

Optional Input Arguments

Specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character vector, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

The following table lists valid parameter names.

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `2`-by-`1` column vector, or `2`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `heston` applies the same initial value to both state variables on all trials. If `StartState` is a bivariate column vector, `heston` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `heston` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.

`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as a scalar, a `2`-by-`2` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns a `2`-by-`2` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is a `2`-by-`2` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`heston`	Object of class `heston` with the following displayed parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at `StartTime` • `Correlation`: Access function for the `Correlation` input, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method • `Return`: Access function for the input argument `Return`, callable as a function of time and state • `Speed`: Access function for the input argument `Speed`, callable as a function of time and state • `Level`: Access function for the input argument `Level`, callable as a function of time and state • `Volatility`: Access function for the input argument `Volatility`, callable as a function of time and state

Examples

See “Creating Heston Stochastic Volatility Models” on page 17-31.

References

Ait-Sahalia, Y. “Testing Continuous-Time Models of the Spot Interest Rate.” *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`cir` | `gbm` | `sdeddo`

Topics

“Simulating Equity Prices” on page 17-34
“Simulating Interest Rates” on page 17-59
“Stratified Sampling” on page 17-70
“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008b

hhigh

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.
`'ParameterName'`	The valid parameter name is: • `HighName`: high prices series name
`ParameterValue`	The parameter value is a character vector that represents the valid parameter name.

Description

`hhv = hhigh(data)` generates a vector of highest high values for the past 14 periods from the matrix `data`.

`hhv = hhigh(data,nperiods,dim)` generates a vector of highest high values for the past `nperiods` periods. `dim` indicates the direction in which the highest high is to be searched. If you input `[]` for `nperiods`, the default is `14`.

`hhvts = hhigh(tsobj,nperiods)` generates a vector of highest high values from `tsobj`, a financial time series object. `tsobj` must include at least the series `High`. The output `hhvts` is a financial time series object with the same dates as `tsobj` and data series named `HighestHigh`. If `nperiods` is specified, `hhigh` generates a financial time series object of highest high values for the past `nperiods` periods.

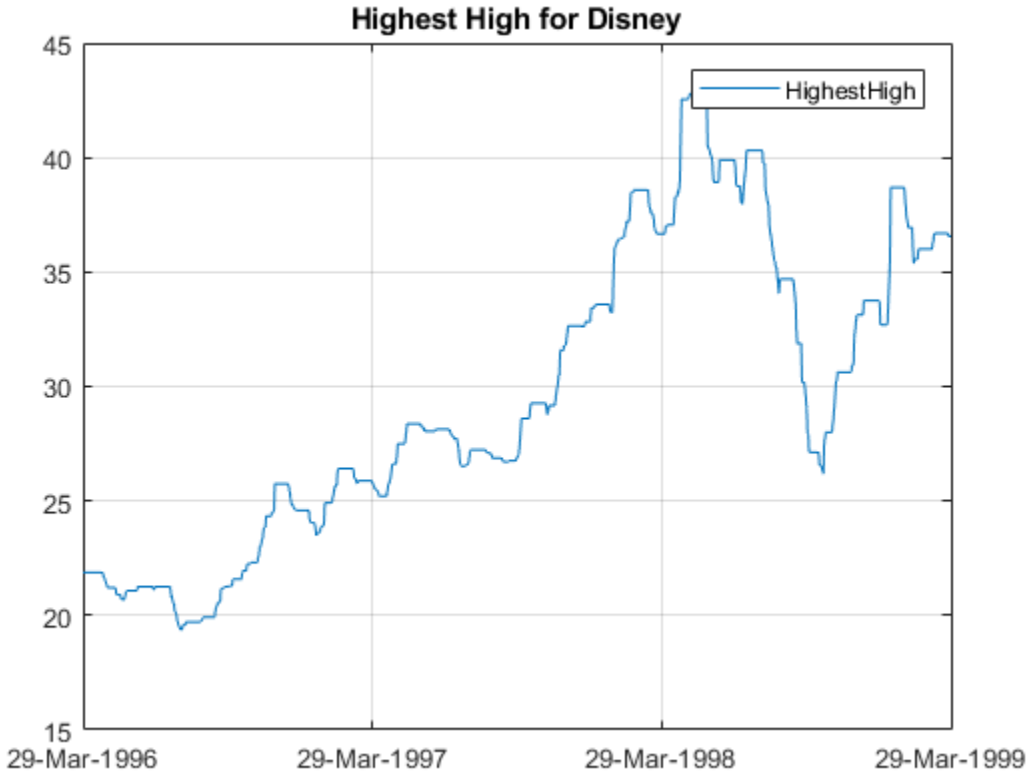
`hhvts = hhigh(tsobj,nperiods,'ParameterName',ParameterValue, ...)` specifies the name for the required data series when it is different from the default name. The valid parameter name is`HighName`. The parameter value is a character vector that represents the valid parameter name.

Examples

Compute the Highest High Price

This example shows how to compute the highest high price for Disney stock and plot the results.

```
load disney.mat
dis_HHigh = hhigh(dis);
plot(dis_HHigh)
title('Highest High for Disney')
```



- “Technical Analysis Examples” on page 16-4

See Also

`llow`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

highlow (fts)

Time series High-Low plot

Syntax

```
highlow(tsobj)

highlow(tsobj,color)

highlow(tsobj,color,dateform)

highlow(tsobj,color,dateform,'ParameterName',ParameterValue, ...)

hll = highlow(tsobj,color,dateform,'ParameterName',ParameterValue, ...)
```

Arguments

`tsobj`	Financial time series object.
`color`	(Optional) A three-element row vector representing RGB or a color identifier. (See `plot`.)
`dateform`	(Optional) Date format used as the *x*-axis tick labels. (See `datetick`.) 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'`.
`'ParameterName'`	`'ParameterName'` can be: • `HighName`: high prices series name • `LowName`: low prices series name • `OpenName`: open prices series name • `CloseName`: closing prices series name
`ParameterValue`	The parameter value is a character vector that represents the valid parameter name.

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 format used as the *x*-axis tick labels. See `datestr` for a list of date formats.

`highlow(tsobj,color,dateform,'ParameterName',ParameterValue, ...)` indicates the actual names of the required data series if the data series do not have the default names.

You can specify open prices as optional by providing the parameter name `'OpenName'` and the parameter value `''`.

```
highlow(tsobj, color, dateform, 'OpenName', '')
```

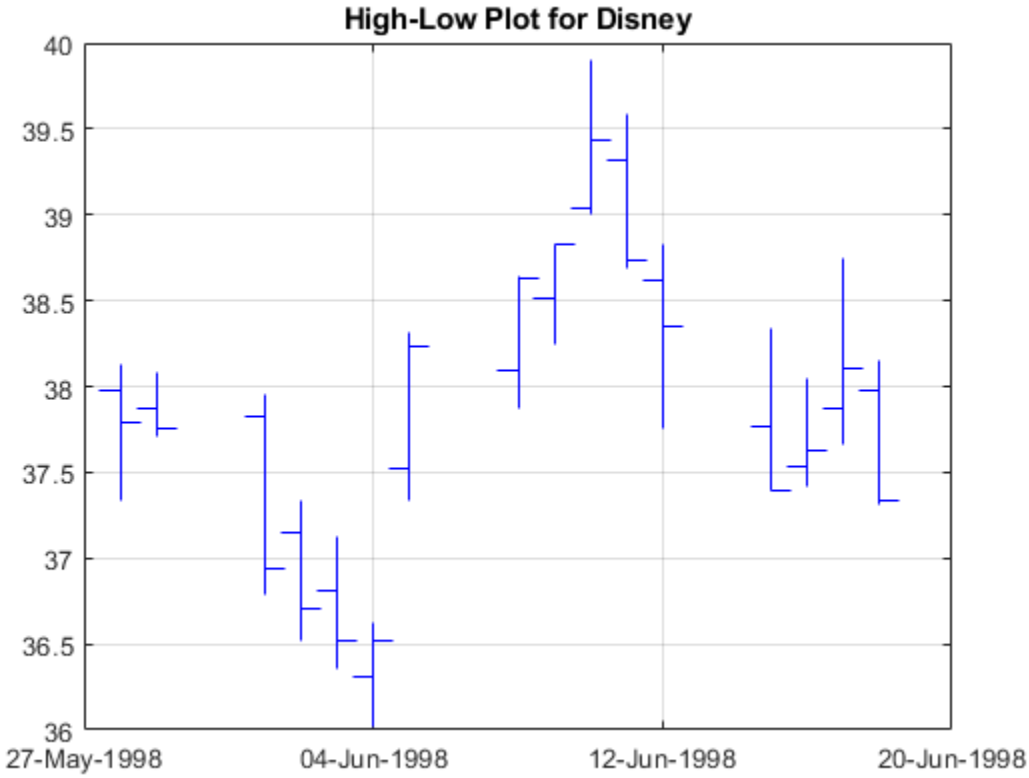
`hhll = highlow(tsobj,color,dateform,'ParameterName',ParameterValue, ...)` returns the handle to the line object that makes up the High-Low plot.

Examples

Generate a High-Low Plot

This example shows how to generate a High-Low plot for Disney stock for the dates May 28 to June 18, 1998.

```
load disney.mat
highlow(dis('28-May-1998::18-Jun-1998'))
title('High-Low Plot for Disney')
```



- “Technical Analysis Examples” on page 16-4

See Also

`candle`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

highlow

High, low, open, close chart

Syntax

```
highlow(High,Low,Close,Open,Color)

highlow(High,Low,Close,Open,Color,Dates,Dateform)

Handles = highlow(High,Low,Close,Open,Color,Dates,Dateform)
```

Arguments

`High`	High prices for a security. A column vector.
`Low`	Low prices for a security. A column vector.
`Close`	Closing prices for a security. A column vector.
`Open`	(Optional) Opening prices for a security. A column vector. To specify `Color` when `Open` is unknown, enter `Open` as an empty matrix `[]`.
`Color`	(Optional) Vertical line color, specified as a character vector. MATLAB software supplies a default color if none is specified. The default color differs depending on the background color of the figure window. See `ColorSpec` for `color` names.
`Dates`	(Optional) User-defined dates, specified as a serial date number or datetime array. A column vector.
`Dateform`	(Optional) Format of the date character vector as tick labels. For more information on date formats, see `dateaxis`.

Description

`highlow(High,Low,Close,Open,Color)` plots the high, low, opening, and closing prices of an asset. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right.

`highlow(High,Low,Close,Open,Color,Dates,Dateform)` plots the high, low, opening, and closing prices of an asset. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right. The plot also contains user-defined dates and date character vector format for tick labels.

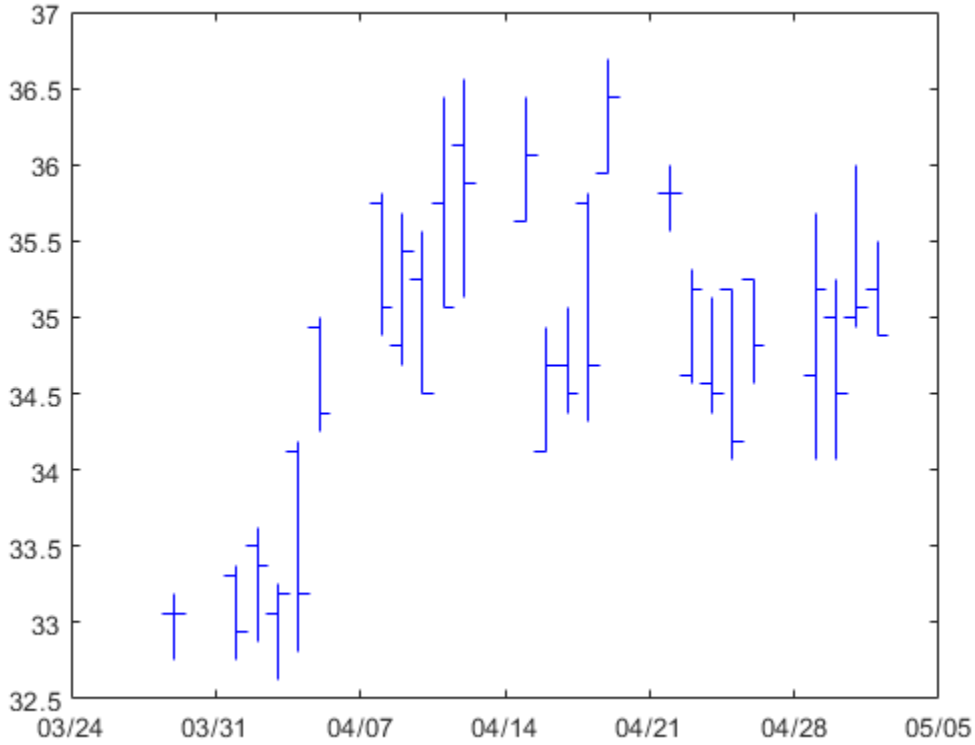
`Handles = highlow(High,Low,Close,Open,Color,Dates,Dateform)` plots the figure and returns the handles of the lines.

Examples

Create a HighLow Chart

The high, low, open, and closing prices for the equity DIS are stored in equal-length vectors `dis_HIGH`, `dis_LOW`, `dis_OPEN`, and `dis_CLOSE` respectively and `highlow` plots the price data using blue lines.

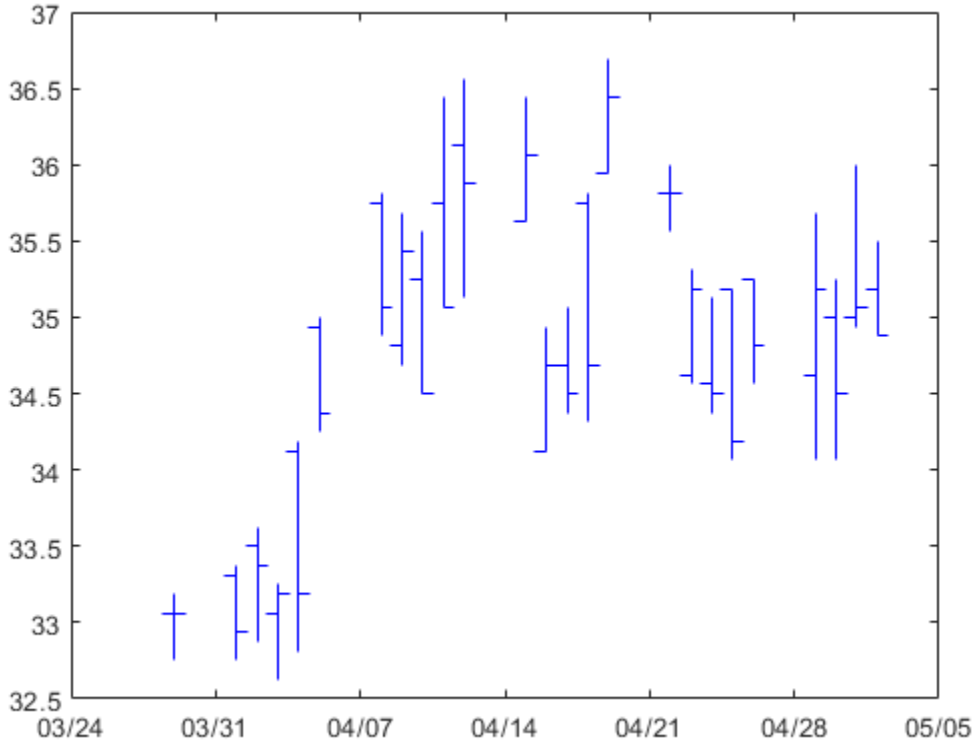
```
load disney.mat
range = 1:25;
highlow(dis_HIGH(range), dis_LOW(range), dis_CLOSE(range),...
    dis_OPEN(range),'blue',dis.dates(range));
```



Create a HighLow Chart Using datetime Input

The high, low, open, and closing prices for the equity DIS are stored in equal-length vectors `dis_HIGH`, `dis_LOW`, `dis_OPEN`, and `dis_CLOSE` respectively and `highlow` plots the price data using blue lines.

```
load disney.mat
range = 1:25;
highlow(dis_HIGH(range), dis_LOW(range), dis_CLOSE(range),...
dis_OPEN(range),'blue',datetime(dis.dates(range),'ConvertFrom','datenum','Locale','en_U
```



- "High-Low-Close Chart" on page 2-16
- "Technical Analysis Examples" on page 16-4

See Also

`bolling` | `candle` | `dateaxis` | `datetime` | `highlow` | `movavg` | `pointfig`

Topics

"High-Low-Close Chart" on page 2-16
"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

hist

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`.

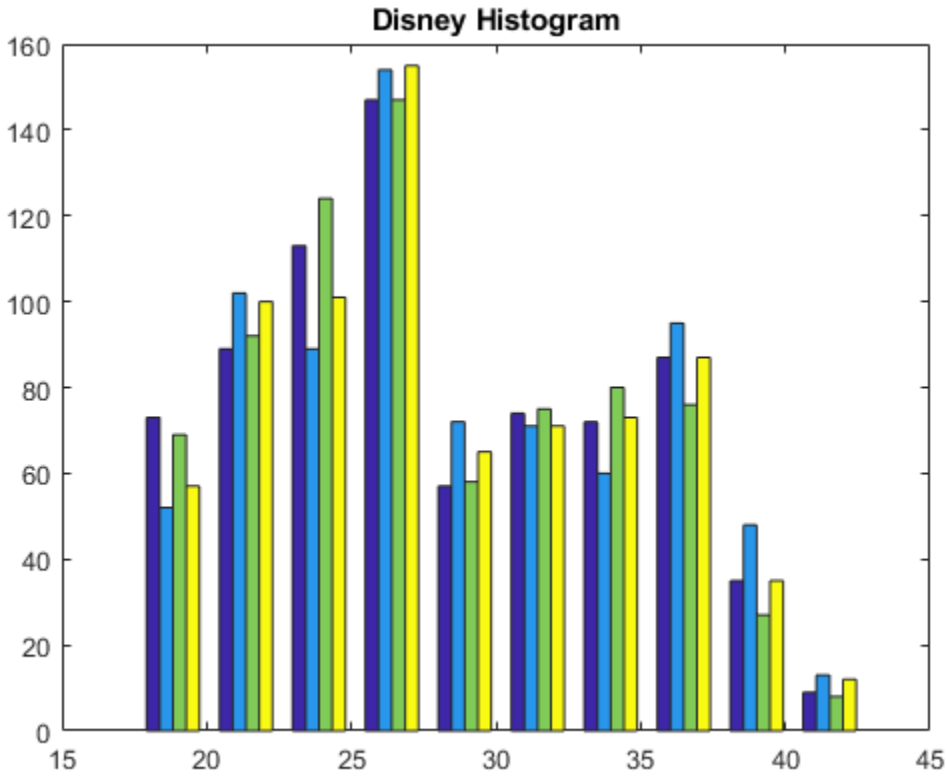
`[ftshist,binpos] = hist(tsobj,numbins)` additionally returns the bin positions `binpos`. The positions are the centers of each bin. `binpos` is a column vector.

Examples

Create a Histogram

This example shows how to generate a histogram of Disney open, high, low, and close prices.

```
load disney.mat
dis = rmfield(dis,'VOLUME'); % Remove VOLUME field
hist(dis)
title('Disney Histogram')
```



- “Financial Time Series Operations” on page 12-8
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`histogram` | `mean` | `std`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

hwv class

Hull-White/Vasicek Gaussian Diffusion model

Description

`hwv` objects derive from the `sdemrd` (SDE with drift rate expressed in mean-reverting form) class. Use `hwv` objects to simulate sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form. These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Hull-White/Vasicek stochastic processes with Gaussian diffusions.

This model allows you to simulate vector-valued Hull-White/Vasicek processes of the form:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- S is an `NVARS`-by-`NVARS` of mean reversion speeds (the rate of mean reversion).
- L is an `NVARS`-by-`1` vector of mean reversion levels (long-run mean or level).
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- $dW_t$is an `NBROWNS`-by-`1` Brownian motion vector.

Construction

`HWV = hwv(Speed,Level,Sigma)` constructs a default `hwv` object.

`HWV = hwv(Speed,Level,Sigma,Name,Value)` constructs a `hwv` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `hwv` object, see `hwv`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Speed` — `Return` represents the parameter *S*

array or deterministic function of time or deterministic function of time and state

`Speed` represents the parameter *S*, specified as an array or deterministic function of time.

If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate at which the state vector reverts to its long-run average `Level`).

As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed` as a function of time and state, it calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Level` — `Level` represents the parameter L

array or deterministic function of time or deterministic function of time and state

`Level` represents the parameter L, specified as an array or deterministic function of time.

If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels.

As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1` column vector. If you specify `Level` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of reversion levels when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter V

array or deterministic function of time or deterministic function of time and state

`Sigma` represents the parameter V, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Although the`hwv` constructor does not enforce restrictions on the signs of any of these input arguments, each argument is specified as a positive value.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `hwv`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t,X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by $(t, X_t$.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

`StartState` — Initial values of state variables

1 (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the `gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

`Simulation` — User-defined simulation function or SDE simulation method

if you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

SetAccess	public
GetAccess	public

Data Types: function_handle

Methods

simBySolution	Simulate approximate solution of diagonal-drift HWV processes

Inherited Methods

The following methods are inherited from the `sde` class.

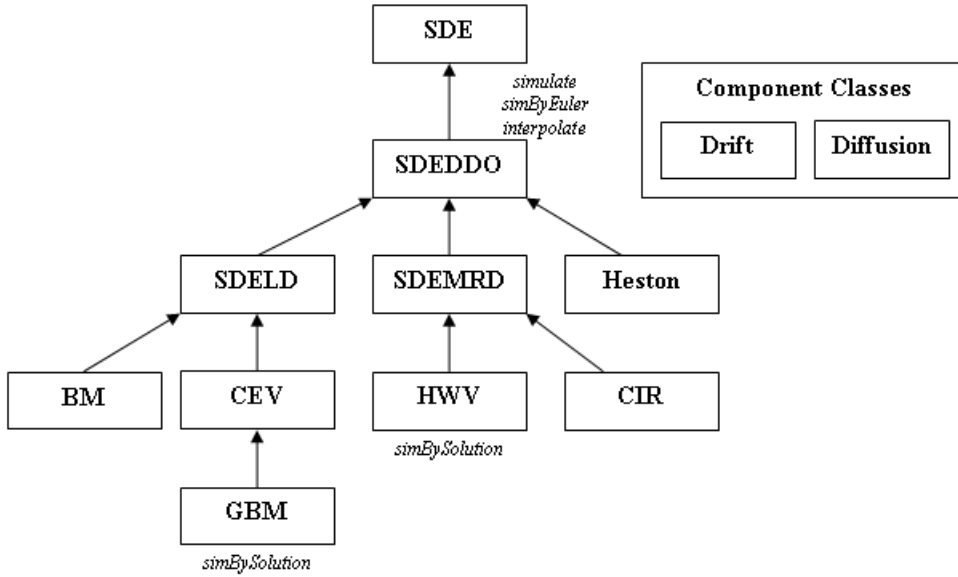
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a hwv Object

The Hull-White/Vasicek (HWV) short rate class derives directly from SDE with mean-reverting drift (that is, `SDEMRD`): $dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$.

Create a `hwv` object to represent the model: $dX_t = 0.2(0.1 - X_t)dt + 0.05dW_t$.

```
obj = hwv(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
obj =
   Class HWV: Hull-White/Vasicek
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

- “Simulating Equity Prices” on page 17-34
- “Simulating Interest Rates” on page 17-59
- “Stratified Sampling” on page 17-70
- “Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
- “Base SDE Models” on page 17-16
- “Drift and Diffusion Models” on page 17-19

- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `hwv` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeddo` | `simByEuler` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

hwv

Construct HWV model

Syntax

```
HWV = hwv(Speed, Level, Sigma)

HWV = hwv(Speed, Level, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`hwv`

Description

This constructor creates and displays `hwv` objects, which derive from the`sdemrd` (SDE with drift rate expressed in mean-reverting form) class. Use `hwv` objects to simulate sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form. These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time Hull-White/Vasicek stochastic processes with Gaussian diffusions.

This method allows you to simulate vector-valued Hull-White/Vasicek processes of the form:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- S is an `NVARS`-by-`NVARS` of mean reversion speeds (the rate of mean reversion).
- L is an `NVARS`-by-`1` vector of mean reversion levels (long-run mean or level).
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.

- $dW_t$is an `NBROWNS`-by-`1` Brownian motion vector.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Speed`	`Speed` represents the function *S*. If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate at which the state vector reverts to its long-run average `Level`). As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed` as a function of time and state, it calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

`Level`	`Level` represents the function L. If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels. As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1` column vector. If you specify `Level` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of reversion levels when called with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.
`Sigma`	`Sigma` represents the parameter V. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify it as a function of time and state, `Sigma` must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of any of these input arguments, each argument is specified as a positive value.

Optional Input Arguments

Specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character vector, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.

- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `hwv` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `hwv` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `hwv` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.

`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`HWV`	Object of class `hwv` with the following displayed parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at `StartTime` • `Correlation`: Access function for the `Correlation` input, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method • `Speed`: Access function for the input argument `Speed`, callable as a function of time and state • `Level`: Access function for the input argument `Level`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state

Examples

“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 17-30

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `hwv` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59

"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

holdings2weights

Portfolio holdings into weights

Syntax

```
Weights = holdings2weights(Holdings,Prices,Budget)
```

Arguments

`Holdings`	Number of portfolios (`NPORTS`) by number of assets (`NASSETS`) matrix with the holdings of `NPORTS` portfolios containing `NASSETS` assets.
`Prices`	`NASSETS` vector of asset prices.
`Budget`	(Optional) Scalar or `NPORTS` vector of nonzero budget constraints. Default = `1`.

Description

`Weights = holdings2weights(Holdings,Prices,Budget)` converts portfolio holdings into portfolio weights. The weights must satisfy a budget constraint such that the weights sum to `Budget` for each portfolio.

Weights is a `NPORTS` by `NASSETS` matrix containing the normalized weights of `NPORTS` portfolios containing `NASSETS` assets.

Notes

- `Holdings` may be negative to indicate a short position, but the overall portfolio weights must satisfy a nonzero budget constraint.
- The weights in each portfolio sum to the `Budget` value (which is `1` if `Budget` is unspecified.)

See Also

`weights2holdings`

Topics

“Data Transformation and Frequency Conversion” on page 12-12

Introduced before R2006a

holidays

Holidays and nontrading days

Syntax

```
H = holidays
H = holidays(StartDate,EndDate)
H = holidays(___,AltHolidays)
```

Description

`H = holidays` returns a vector of serial date numbers corresponding to all holidays and nontrading days.

`H = holidays(StartDate,EndDate)` returns a vector of serial date numbers corresponding to the holidays and nontrading days between`StartDate` and `EndDate`, inclusive.

`H = holidays(___,AltHolidays)` returns a vector of serial date numbers corresponding to the alternate list of holidays and nontrading days.

Examples

Determine Holidays for a Given `StartDate` and `EndDate`

Create a vector of serial date numbers corresponding to all holidays and nontrading dates between a specified `StartDate` and `EndDate`:

```
H = holidays('jan 1 2001', 'jun 23 2001')
```

```
H =

      730852
      730866
```

```
       730901
       730954
       730999


datestr(H)

ans = 5x11 char array
    '01-Jan-2001'
    '15-Jan-2001'
    '19-Feb-2001'
    '13-Apr-2001'
    '28-May-2001'
```

Alternatively, using a datetime array for `StartDate` and `EndDate` returns a datetime array for `H`.

```
H = holidays(datetime('1-Jan-2001','Locale','en_US'),...
datetime('23-Jun-2001','Locale','en_US'))

H = 5x1 datetime array
   01-Jan-2001
   15-Jan-2001
   19-Feb-2001
   13-Apr-2001
   28-May-2001
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified using a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified using a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`AltHolidays` — Alternate list of holidays and nontrading days
serial date number | date character vector | datetime object

Alternate list of holidays and nontrading days, specified using a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

Output Arguments

`H` — Dates corresponding to all holidays and nontrading days
vector

Dates corresponding to all holidays and nontrading days, returned as a vector of dates. If `StartDate`, `EndDate`, and `AltHolidays` are all either serial date numbers or date character vectors, `H` is returned as serial date numbers. If either `StartDate`, `EndDate`, or `AltHolidays` are datetime arrays, `H` is returned as a datetime array.

Definitions

`holidays`

The `holidays` function is based on a modern 5-day workweek.

This function contains all holidays and special nontrading days for the New York Stock Exchange from January 1, 1885 to December 31, 2070.

Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures from 1885 to 2070 should include Saturday trading days. To capture these dates, use the function `nyseclosures`. The results from `holidays` and `nyseclosures` are identical if the `WorkWeekFormat` in `nyseclosures` is `'Modern'`.

See Also

`busdate` | `createholidays` | `datetime` | `isbusday` | `lbusdate` | `nyseclosures`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2
"UICalendar User Interface" on page 15-4

Introduced before R2006a

horzcat

Concatenate financial time series objects horizontally

Syntax

```
horzcat
```

Description

`horzcat` implements horizontal concatenation of financial time series objects. `horzcat` essentially merges the data columns of the financial time series objects. The time series objects must contain the exact same dates and times.

When multiple instances of a data series name occur, concatenation adds a suffix to the current names of the data series. The suffix has the format `_objectname<n>`, where `n` is a number indicating the position of the time series, from left to right, in the concatenation command. The `n` part of the suffix appears only when there is more than one instance of a particular data series name.

The description fields are concatenated as well. They are separated by two forward slashes (//).

Examples

Construct three financial time series, each containing a data series named `DataSeries`:

```
firstfts  = fints((today:today+4)', (1:5)','DataSeries','d');
secondfts = fints((today:today+4)', (11:15)','DataSeries','d');
thirdfts  = fints((today:today+4)', (21:25)','DataSeries','d');
```

Concatenate the time series horizontally into a new financial time series `newfts`:

```
newfts  = [firstfts secondfts thirdfts secondfts];
```

The resulting object `newfts` has data series names `DataSeries_firstfts`, `DataSeries_secondfts2`, `DataSeries_thirdfts`, and `DataSeries_secondfts4`.

Verify this with the command

```
fieldnames(newfts)

ans =

     'desc'
     'freq'
     'dates'
     'DataSeries_firstfts'
     'DataSeries_secondfts2'
     'DataSeries_thirdfts'
     'DataSeries_secondfts4'
     'times'
```

Use `chfield` to change the data series names.

Note If all input objects have the same frequency, the new object has that frequency as well. However, if one of the objects concatenated has a different frequency from the others, the frequency indicator of the resulting object is set to `Unknown (0)`.

See Also

`busdate` | `createholidays` | `isbusday` | `lbusdate` | `nyseclosures`

Topics

"Merge Financial Time Series Objects" on page 13-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

hour

Hour of date or time

Syntax

```
Hour = hour(Date)
Hour = hour(___,F)
```

Description

`Hour = hour(Date)` returns the hour of the day given a serial date number or a date character vector.

`Hour = hour(___,F)` returns the hour of one or more date character vectors using format defined by the optional input `F`. `Date` can be an array of date character vectors, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`.

Examples

Determine the Hour of the Day for Various Dates

Find the hour of the day for `Date` using a serial date number.

```
Hour = hour(730473.5584278936)
```

```
Hour = 13
```

Find the hour of the day for `Date` using a date character vector format.

```
Hour = hour('19-dec-1999, 13:24:08.17')
```

```
Hour = 13
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Date` — Date to determine hour

serial date number | date character vector | cell array of date character vectors

Date to determine hour, specified as a serial date number or date character vector.

`Date` can be an array of date character vectors, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All of the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`

Data Types: `single` | `double` | `char` | `cell`

`F` — Date format symbol

character vector designating date format

Date format symbol, specified as a character vector to designate the date format symbol for input argument `Date`. For more information on supported date character vector formats, see `datestr`. Note, formats with `'Q'` are not accepted.

Data Types: `char`

Output Arguments

`Hour` — Hour of day

serial date number | datetime array

Hour of the day, returned as a serial date number or date character vector.

See Also

`datevec` | `minute` | `second`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

inforatio

Calculate information ratio for one or more assets

Syntax

```
inforatio(Asset,Benchmark)

Ratio = inforatio(Asset,Benchmark)

[Ratio,TE] = inforatio(Asset,Benchmark)
```

Arguments

`Asset`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` observations of asset returns for `NUMSERIES` asset return series.
`Benchmark`	`NUMSAMPLES` vector of returns for a benchmark asset. The periodicity must be the same as the periodicity of `Asset`. For example, if `Asset` is monthly data, then `Benchmark` must be monthly returns.

Description

Given `NUMSERIES` assets with `NUMSAMPLES` returns for each asset in a `NUMSAMPLES x NUMSERIES` matrix `Asset` and given a `NUMSAMPLES` vector of benchmark returns in `Benchmark`, `inforatio` computes the information ratio and tracking error for each asset relative to the `Benchmark`.

To summarize the outputs of `inforatio`:

- `Ratio` is a `1 x NUMSERIES` row vector of information ratios for each series in `Asset`. Any series in `Asset` with a tracking error of 0 has a `NaN` value for its information ratio.
- `TE` is a `1 x NUMSERIES` row vector of tracking errors, that is, the standard deviation of `Asset` relative to `Benchmark` returns, for each series.

Note `NaN` values in the data are ignored. If the `Asset` and `Benchmark` series are identical, the information ratio is `NaN` since the tracking error is 0. The information ratio and the Sharpe ratio of an `Asset` versus a riskless `Benchmark` (a `Benchmark` with standard deviation of returns equal to 0) are equivalent. This equivalence is not necessarily true if the `Benchmark` is risky.

Examples

See “Information Ratio” on page 7-8.

References

Richard C. Grinold and Ronald N. Kahn. *Active Portfolio Management.* 2nd. Edition. McGraw-Hill, 2000.

Jack Treynor and Fischer Black. "How to Use Security Analysis to Improve Portfolio Selection." *Journal of Business.* Vol. 46, No. 1, January 1973, pp. 66–86.

See Also

`portalpha` | `sharpe`

Topics

“Information Ratio” on page 7-8
“Performance Metrics Overview” on page 7-2

Introduced in R2006b

interpolate

Brownian interpolation of stochastic differential equations

Syntax

```
[XT, T] = interpolate(MDL, T, Paths)

[XT, T] = interpolate(MDL, Paths, 'Name1', Value1, 'Name2',
Value2, ...)
```

Classes

All classes in “SDE Class Hierarchy” on page 17-5.

Description

This method performs a Brownian interpolation into a user-specified time series array, based on a piecewise-constant Euler sampling approach.

Consider a vector-valued SDE of the form:
$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$

where:

- X is an *NVARS*-by-`1` state vector.
- F is an *NVARS*-by-`1` drift-rate vector-valued function.
- G is an *NVARS*-by-*NBROWNS* diffusion-rate matrix-valued function.
- W is an *NBROWNS*-by-`1` Brownian motion vector.

Given a user-specified time series array associated with this equation, this method performs a Brownian (stochastic) interpolation by sampling from a conditional Gaussian distribution. This sampling technique is sometimes called a *Brownian bridge*.

Note Unlike simulation methods, the `interpolate` method does not support user-specified noise processes.

Input Arguments

`MDL`	Stochastic differential equation model.
`T`	`NTIMES` element vector of interpolation times. The length of this vector determines the number of rows in the interpolated output time series `XT`.
`Paths`	`NPERIODS`-by-`NVARS`-by-`NTRIALS` time series array of sample paths of correlated state variables. For a given trial, each row of this array is the transpose of the state vector X_t at time t. `Paths` is the initial time series array into which `interpolate` performs the Brownian interpolation.

Optional Input Arguments

Specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character vector, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`Times`	Vector of monotonically increasing observation times associated with the time series input `Paths`. If you do not specify a value for this parameter, `Times` is a zero-based, unit-increment column vector of length `NPERIODS`.

`Refine`	Scalar logical flag that indicates whether `interpolate` uses the interpolation times you request (see `T`) to refine the interpolation as new information becomes available. If you do not specify a value for this argument or set it to `FALSE` (the default value), `interpolate` bases the interpolation only on the state information specified in `Paths`. If you set `Refine` to `TRUE`, `interpolate` inserts all new interpolated states into the existing `Paths` array as they become available. This refines the interpolation grid available to subsequent interpolation times during the current trial.
`Processes`	Function or cell array of functions that indicates a sequence of background processes or state vector adjustments of the form $X_t = P(t, X_t)$ The `interpolate` method runs processing functions at each interpolation time. They must accept the current interpolation time t, and the current state vector X_t, and return a state vector that may be an adjustment to the input state. If you specify more than one processing function, `interpolate` invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and so on. If you do not specify a processing function, `interpolate` makes no adjustments and performs no processing.

Output Arguments

`XT`	`NTIMES`-by-`NVARS`-by-`NTRIALS` time series array of interpolated state variables. For a given trial, each row of this array is the transpose of the interpolated state vector X_t at time t. `XT` is the interpolated time series formed by interpolating into the input `Paths` time series array.

`T`	`NTIMES`-by-1 column vector of interpolation times associated with the output time series `XT`. If the input interpolation time vector `T` contains no missing observations (`NaN`s), this output is the same time vector as `T`, but with the `NaN`s removed. This reduces the length of `T` and the number of rows of `XT`.

Examples

Stochastic Interpolation Without Refinement

Many applications require knowledge of the state vector at intermediate sample times that are initially unavailable. One way to approximate these intermediate states is to perform a deterministic interpolation. However, deterministic interpolation techniques fail to capture the correct probability distribution at these intermediate times. Brownian (or stochastic) interpolation captures the correct joint distribution by sampling from a conditional Gaussian distribution. This sampling technique is sometimes referred to as a *Brownian Bridge*.

The default stochastic interpolation technique is designed to interpolate into an existing time series and ignore new interpolated states as additional information becomes available. This technique is the usual notion of interpolation, which is called *Interpolation without refinement*.

Alternatively, the interpolation technique may insert new interpolated states into the existing time series upon which subsequent interpolation is based, by that means refining information available at subsequent interpolation times. This technique is called *interpolation with refinement*.

Interpolation without refinement is a more traditional technique, and is most useful when the input series is closely spaced in time. In this situation, interpolation without refinement is a good technique for inferring data in the presence of missing information, but is inappropriate for extrapolation. Interpolation with refinement is more suitable when the input series is widely spaced in time, and is useful for extrapolation.

The stochastic interpolation method is available to any model. It is best illustrated, however, by way of a constant-parameter Brownian motion process. Consider a correlated, bivariate Brownian motion (`BM`) model of the form:

$$dX_{1t} = 0.3dt + 0.2dW_{1t} - 0.1dW_{2t}$$
$$dX_{2t} = 0.4dt + 0.1dW_{1t} - 0.2dW_{2t}$$
$$E[dW_{1t}dW_{2t}] = \rho dt = 0.5dt$$

1. Create a `bm` object to represent the bivariate model:

```
mu    = [0.3; 0.4];
sigma = [0.2 -0.1; 0.1 -0.2];
rho   = [1 0.5; 0.5 1];
obj   = bm(mu,sigma,'Correlation',rho);
```

2. Assuming that the drift (`Mu`) and diffusion (`Sigma`) parameters are annualized, simulate a single Monte Carlo trial of daily observations for one calendar year (250 trading days):

```
rng default                 % make output reproducible
dt    = 1/250;  % 1 trading day = 1/250 years
[X,T] = simulate(obj,250,'DeltaTime',dt);
```

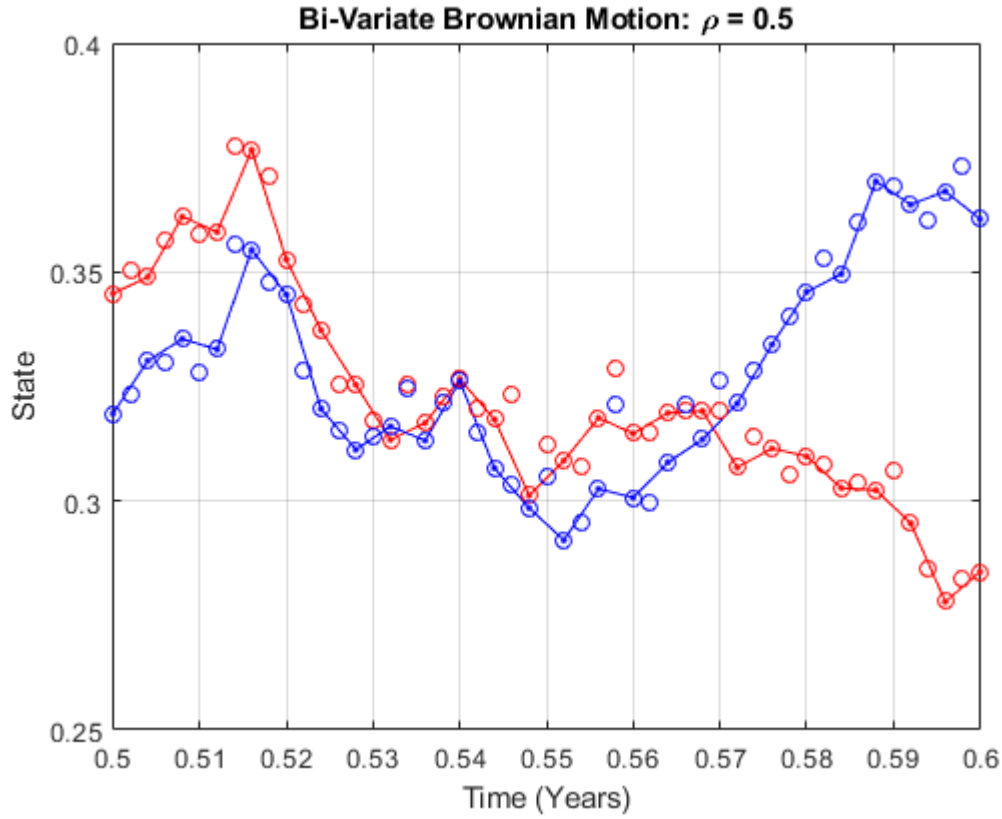
3. It is helpful to examine a small interval in detail.

   a. Interpolate into the simulated time series with a Brownian bridge:

```
t = ((T(1) + dt/2):(dt/2):(T(end) - dt/2));
x = interpolate(obj,t,X,'Times',T);
```

   b. Plot both the simulated and interpolated values:

```
plot(T,X(:,1),'.-r',T,X(:,2),'.-b')
grid on;
hold on;
plot(t,x(:,1),'or',t,x(:,2),'ob')
hold off;
xlabel('Time (Years)')
ylabel('State')
title('Bi-Variate Brownian Motion: \rho = 0.5')
axis([0.4999 0.6001 0.25 0.4])
```



In this plot:

- The solid red and blue dots indicate the simulated states of the bivariate model.
- The straight lines that connect the solid dots indicate intermediate states that would be obtained from a deterministic linear interpolation.
- Open circles indicate interpolated states.
- Open circles associated with every other interpolated state encircle solid dots associated with the corresponding simulated state. However, interpolated states at the midpoint of each time increment typically deviate from the straight line connecting each solid dot.

Simulation of Conditional Gaussian Distributions

You can gain additional insight into the behavior of stochastic interpolation by regarding a Brownian bridge as a Monte Carlo simulation of a conditional Gaussian distribution.

This example examines the behavior of a Brownian bridge over a single time increment.

1 Divide a single time increment of length `dt` into 10 subintervals:

```
mu    = [0.3; 0.4];
sigma = [0.2 -0.1; 0.1 -0.2];
rho   = [1 0.5; 0.5 1];
obj   = bm(mu,sigma,'Correlation',rho);

rng default; % make output reproducible
dt    = 1/250;  % 1 trading day = 1/250 years
[X,T] = simulate(obj,250,'DeltaTime',dt);

n        = 125;    % index of simulated state near middle
times    = (T(n):(dt/10):T(n + 1));
nTrials  = 25000;  % # of Trials at each time
```

2 In each subinterval, take 25000 independent draws from a Gaussian distribution, conditioned on the simulated states to the left, and right:

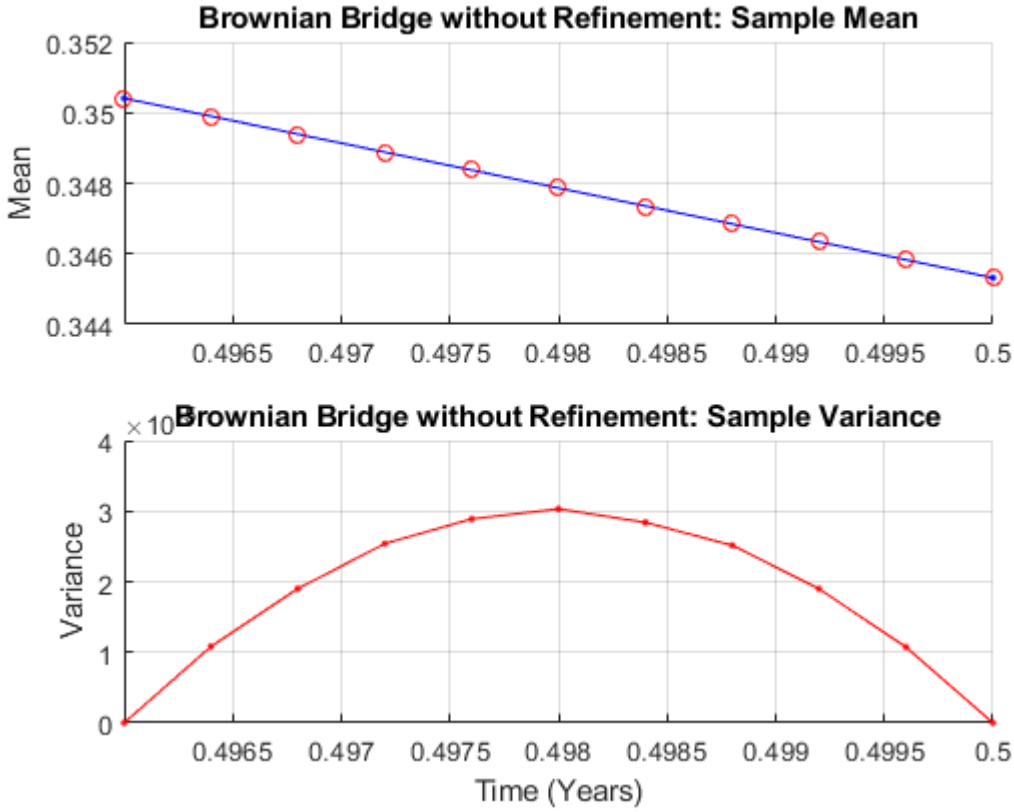
```
average  = zeros(length(times),1);
variance = zeros(length(times),1);
for i = 1:length(times)
    t = times(i);
    x = interpolate(obj,t(ones(nTrials,1)),...
        X,'Times',T);
    average(i)  = mean(x(:,1));
    variance(i) = var(x(:,1));
end
```

3 Plot the sample mean and variance of each state variable:

Note The following graph plots the sample statistics of the first state variable only, but similar results hold for any state variable.

```
subplot(2,1,1);
hold on;
grid on;
```

```
plot([T(n) T(n + 1)],[X(n,1) X(n + 1,1)],'.-b')
plot(times, average, 'or')
hold off;
title('Brownian Bridge without Refinement: Sample Mean')
ylabel('Mean')
limits = axis;
axis([T(n) T(n + 1) limits(3:4)]);

subplot(2,1,2)
hold on;
grid on;
plot(T(n),0,'.-b',T(n + 1),0,'.-b')
plot(times, variance, '.-r')
hold('off');
title('Brownian Bridge without Refinement: Sample Variance')
xlabel('Time (Years)')
ylabel('Variance')
limits = axis;
axis([T(n) T(n + 1) limits(3:4)]);
```



The Brownian interpolation within the chosen interval, *dt*, illustrates the following:

- The conditional mean of each state variable lies on a straight-line segment between the original simulated states at each endpoint.
- The conditional variance of each state variable is a quadratic function. This function attains its maximum midway between the interval endpoints, and is zero at each endpoint.
- The maximum variance, although dependent upon the actual model diffusion-rate function *G(t,X)*, is the variance of the sum of `NBROWNS` correlated Gaussian variates scaled by the factor *dt/4*.

The previous plot highlights interpolation without refinement, in that none of the interpolated states take into account new information as it becomes available. If you

had performed interpolation with refinement, new interpolated states would have been inserted into the time series and made available to subsequent interpolations on a trial-by-trial basis. In this case, all random draws for any given interpolation time would be identical. Also, the plot of the sample mean would exhibit greater variability, but would still cluster around the straight-line segment between the original simulated states at each endpoint. The plot of the sample variance, however, would be zero for all interpolation times, exhibiting no variability.

Algorithms

- The `interpolate` method assumes that all model parameters are piecewise-constant, and evaluates them from the most recent observation time in `Times` that precedes a specified interpolation time in `T`. This is consistent with the Euler approach of Monte Carlo simulation.
- When an interpolation time falls outside the interval specified by `Times`, a Euler simulation extrapolates the time series by using the nearest available observation.
- The user-defined time series `Paths` and corresponding observation `Times` must be fully observed (no missing observations denoted by `NaN`s).
- The `interpolate` method assumes that the user-specified time series array `Paths` is associated with the`sde` object. For example, the `Times`/`Paths` input pair is the result of an initial course-grained simulation. However, the interpolation ignores the initial conditions of the`sde` object (`StartTime` and `StartState`), allowing the user-specified `Times`/`Paths` input series to take precedence.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

`sde` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

irr

Internal rate of return

Syntax

```
Return = irr(CashFlow)
[Return, AllRates] = irr(CashFlow)
```

Description

`Return = irr(CashFlow)` calculates the internal rate of return for a series of periodic cash flows.

`[Return, AllRates] = irr(CashFlow)` calculates the internal rate of return and a vector of all internal rates for a series of periodic cash flows.

`irr` uses the following conventions:

- If one or more internal rates of returns (warning if multiple) are strictly positive rates, `Return` sets to the minimum.
- If there is no strictly positive rate of returns, but one or multiple (warning if multiple) returns are nonpositive rates, `Return` sets to the maximum.
- If no real-valued rates exist, `Return` sets to `NaN` (no warnings).

Input Arguments

`CashFlow`

A vector containing a stream of periodic cash flows. The first entry in `CashFlow` is the initial investment. If `CashFlow` is a matrix, `irr` handles each column of `CashFlow` as a separate cash-flow stream.

Output Arguments

`Return`

An internal rate of return associated to `CashFlow`. If `CashFlow` is a matrix, then `Return` is a vector whose entry `j` is an internal rate of return for column `j` in `CashFlow`.

`AllRates`

A vector containing all the internal rates of return associated with `CashFlow`. If `CashFlow` is a matrix, then `AllRates` is also a matrix, with the same number of columns as `CashFlow` and one less row. Also, column `j` in `AllRates` contains all the rates of return associated to column `j` in `CashFlow` (including complex-valued rates).

Examples

Find the internal rate of return for a simple investment with a unique positive rate of return. The initial investment is $100,000 and the following cash flows represent the yearly income from the investment.

- Year 1 — $10,000
- Year 2 — $20,000
- Year 3 — $30,000
- Year 4 — $40,000
- Year 5 — $50,000

Calculate the internal rate of return on the investment:

```
Return = irr([-100000 10000 20000 30000 40000 50000])
```

This returns:

```
Return =

    0.1201
```

If the cash flow payments were monthly, then the resulting rate of return is multiplied by 12 for the annual rate of return.

Find the internal rate of return for multiple rates of return. The project has the following cash flows and a market rate of 10%.

```
 CashFlow = [-1000 6000 -10900 5800]
```

Use `irr` with a single output argument:

```
Return = irr(CashFlow)
```

A warning appears and `irr` returns a 100% rate of return. The 100% rate on the project looks attractive:

```
Warning: Multiple rates of return

> In irr at 166

Return =

    1.0000
```

Use `irr` with two output arguments:

```
[Return, AllRates] = irr(CashFlow)
```

This returns:

```
>> [Return, AllRates] = irr(CashFlow)

Return =

    1.0000

AllRates =

   -0.0488
    1.0000
    2.0488
```

The rates of return in `AllRates` are -4.88%, 100%, and 204.88%. Though some rates are lower and some higher than the market rate, based on the work of Hazen, any rate gives a consistent recommendation on the project. However, you can use a present value analysis in these kinds of situations. To check the present value of the project, use `pvvar`:

```
PV = pvvar(CashFlow,0.10)
```

This returns:

```
PV =

 -196.0932
```

The second argument is the 10% market rate. The present value is -196.0932, negative, so the project is undesirable.

References

Brealey and Myers. *Principles of Corporate Finance.* McGraw-Hill Higher Education, Chapter 5, 2003.

Hazen G. “A New Perspective on Multiple Internal Rates of Return.” *The Engineering Economist.* Vol. 48-1, 2003, pp. 31–51.

See Also

`effrr` | `mirr` | `nomrr` | `pvvar` | `xirr`

Topics

“Interest Rates/Rates of Return” on page 2-21
“Analyzing and Computing Cash Flows” on page 2-21
“Performance Metrics Overview” on page 7-2

Introduced before R2006a

isbusday

True for dates that are business days

Syntax

```
Busday = isbusday(Date)
Busday = isbusday(___,Holiday,Weekend)
```

Description

`Busday = isbusday(Date)` returns logical true (`1`) if `Date` is a business day and logical false (`0`) otherwise.

`Busday = isbusday(___,Holiday,Weekend)`, using optional input arguments, returns logical true (`1`) if `Date` is a business day, and logical false (`0`) otherwise.

Examples

Determine If a Given Date Is a Business Day

Determine if `Date` is a business day.

```
Busday = isbusday('16 jun 2001')
```

```
Busday = logical
   0
```

Determine if a `Date` vector are business days.

```
Date = ['15 feb 2001'; '16 feb 2001'; '17 feb 2001'];
Busday = isbusday(Date)
```

```
Busday = 3x1 logical array
   1
```

```
   1
   0
```

Determine if a `Date` vector are business days using a datetime array.

```
Date = ['15-feb-2001'; '16-feb-2001'; '17-feb-2001'];
Busday = isbusday(datetime(Date,'Locale','en_US'))

Busday = 3x1 logical array
   1
   1
   0
```

Set June 21, 2003 (a Saturday) as a business day.

```
Weekend = [1 0 0 0 0 0 0];
isbusday('June 21, 2003', [], Weekend)

ans = logical
   1
```

If the second argument, `Holiday`, is empty (`[ ]`), the default `Holidays` vector (generated with `holidays` and then associated to the NYSE calendar) is used.

- "Handle and Convert Dates" on page 2-2

Input Arguments

`Date` — Date being checked

serial date number | date character vector | datetime array

Date being checked, specified as a serial date number, date character vector, or datetime array. `Date` can contain multiple dates, but they must all be in the same format. Dates are assumed to be whole date numbers or date stamps with no fractional or time values.

Data Types: `double` | `char` | `datetime`

`Holiday` — Holidays and nontrading-day dates

non-trading day vector is determined by the routine `holidays` (default) | serial date number | date character vector | datetime array

Holidays and nontrading-day dates, specified as vector.

All dates in `Holiday` must be the same format: either serial date numbers, or date character vectors, or datetime arrays. (Using serial date numbers improves performance.) The `holidays` function supplies the default vector.

Data Types: `double` | `char` | `datetime`

`Weekend` — Weekend days

`[1 0 0 0 0 0 1]` (Saturday and Sunday form the weekend) (default) | vector of length 7, containing `0` and `1`, where `1` indicates weekend days

Weekend days, specified as a vector of length 7, containing `0` and `1`, where `1` indicates weekend days and the first element of this vector corresponds to Sunday.

Data Types: `double`

Output Arguments

`Busday` — Logical true if a business day

logical `0` or `1`

Logical true if a business day, returned as a logical true (`1`) if `Date` is a business day and logical false (`0`) otherwise.

See Also

`busdate` | `datetime` | `fbusdate` | `holidays` | `lbusdate`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

iscompatible

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 and equal number of data series. However, the values contained in the data series can be different.

Note Data series names are case-sensitive.

See Also

`isequal`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

isequal

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

`eq` | `iscompatible`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

isempty

True for empty financial time series objects

Syntax

```
tf = isempty(fts)
```

Arguments

`fts`	Financial time series object.

Description

`isempty` for financial times series objects is based on the MATLAB `isempty` function. See `isempty`.

`tf = isempty(fts)` returns true (`1`) if `fts` is an empty financial time series object and false (`0`) otherwise. An empty financial times series object has no elements, that is, `length(fts) = 0`.

See Also

`nanmax` | `nanmean` | `nanmedian` | `nanmin` | `nanstd` | `nanvar`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

isfield

Check whether character vector 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`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

issorted

Check whether dates and times are monotonically increasing

Syntax

```
monod = issorted(tsobj)
```

Arguments

`tsobj`	Financial time series object

Description

`monod = issorted(tsobj)` returns `1` if the dates and times in `tsobj` are monotonically increasing or `0` if they are not.

See Also

`sortfts`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

kagi

Kagi chart

Syntax

```
kagi(X)
```

Arguments

X	X is a M-by-2 matrix or a table. If X is a M-by-2 matrix, the first column contains date numbers and the second column is the asset price. If X is a table, the first column of the table contains the dates. The second column contains the asset price data. Dates can be a serial date number, a date character vector, or a datetime array.

Description

`kagi(X)` plots asset price with respect to dates.

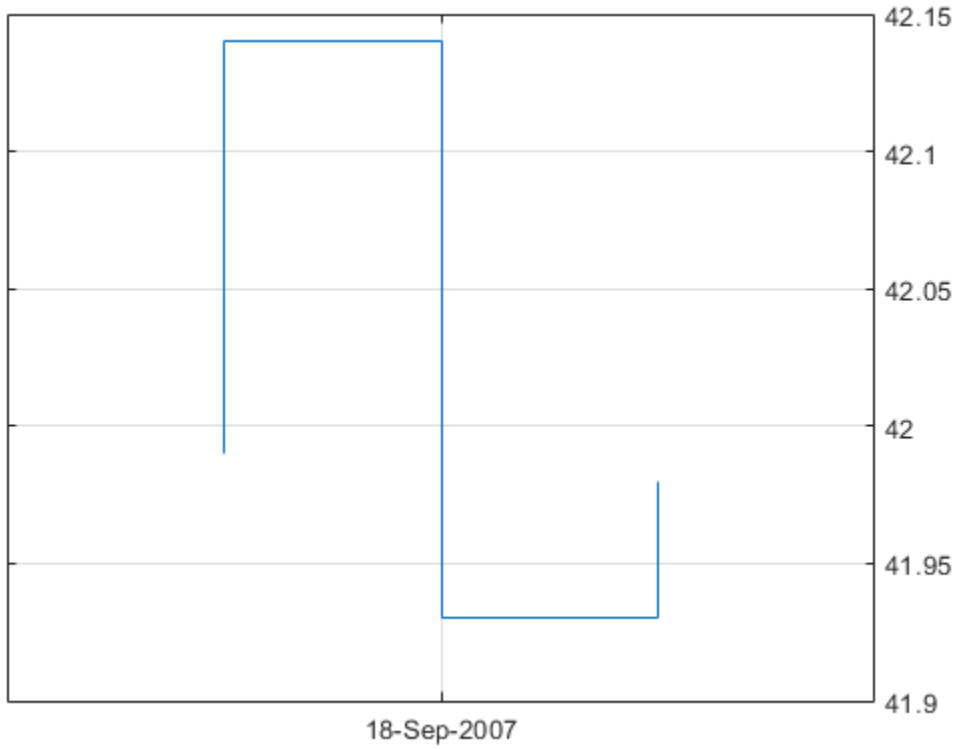
Examples

Create a Kagi Chart

This example shows how to generate a Kagi chart for asset X that is an M-by-2 matrix of date numbers and asset prices.

```
X = [...

733299.00          41.99;...
733300.00          42.14;...
733303.00          41.93;...
733304.00          41.98];
```

```
kagi(X)
```

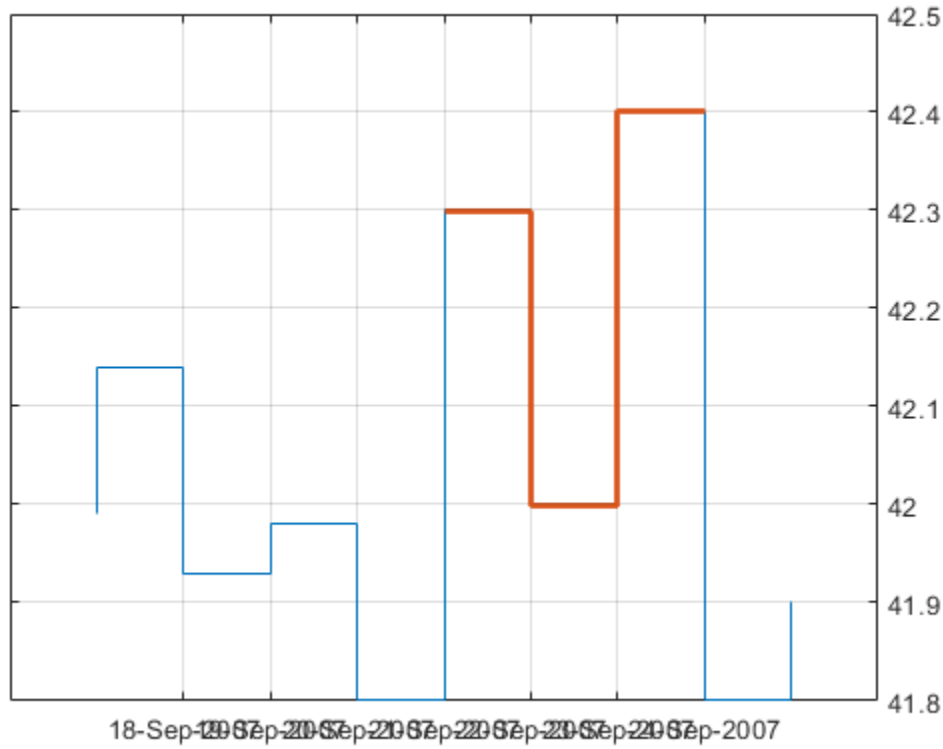


Create a Kagi Chart Using datetime Input

This example shows how to use `datetime` input to generate a Kagi chart for asset `X` that is an `M`-by-2 matrix of date numbers and asset prices.

```
X = [...

733299.00          41.99;...
```

```
733300.00          42.14;...
733303.00          41.93;...
733304.00          41.98;...
733305.00          41.80;...
733306.00          42.30;...
733307.00          42.00;...
733308.00          42.40;...
733309.00          41.80;...
733310.00          41.90];

dates = datetime(X(:,1),'ConvertFrom','datenum','Locale','en_US');
data=X(:,2);
t=table(dates,data);
kagi(t);
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `datetime` | `highlow` | `linebreak` | `movavg` | `pointfig` | `priceandvol` | `renko` | `volarea`

Topics

“Charting Financial Data” on page 2-14

Introduced in R2008a

lagts

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 is 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 later. `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`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

lbusdate

Last business date of month

Syntax

```
Date = lbusdate(Year,Month)
Date = lbusdate(___,Holiday,Weekend,outputType)
```

Description

`Date = lbusdate(Year,Month)` returns the serial date number for the last business date of the given year and month.

`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 character vectors.

`Date = lbusdate(___,Holiday,Weekend,outputType)` returns the serial date number for the last business date of the given year and month using optional input arguments. The optional argument `Holiday` specifies nontrading days.

If neither `Holiday` nor `outputType` are specified, `Date` is returned as a serial date number. If `Holiday` is specified, but not `outputType`, then the type of the holiday variable controls the type of date. If `Holiday` is a serial date number or date character vector, then `Date` is returned as a serial date number.

Examples

Determine the Last Business Date of a Given Year and Month

Determine the `Date` using an input argument for `Year` and `Month`.

```
Date = lbusdate(2001, 5)

Date = 731002

datestr(Date)

ans =
'31-May-2001'
```

Determine the `Date` using the optional input argument for `outputType`.

```
Date = lbusdate(2001, 11,[],[],'datetime')

Date = datetime
   30-Nov-2001
```

Indicate that Saturday is a business day by appropriately setting the `Weekend` argument. May 31, 2003, is a Saturday. Use `lbusdate` to check that this Saturday is actually the last business day of the month.

```
Weekend = [1 0 0 0 0 0 0];
Date = datestr(lbusdate(2003, 5, [], Weekend))

Date =
'31-May-2003'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Year` — Year to determine occurrence of weekday

4-digit integer | vector of 4-digit integers

Year to determine occurrence of weekday, specified as a 4-digit integer or vector of 4-digit integers.

Data Types: `single` | `double`

`Month` — Month to determine occurrence of weekday
integer with value `1` through `12` | vector of integers with values `1` through `12`

Month to determine occurrence of weekday, specified as an integer or vector of integers with values `1` through `12`.

Data Types: `single` | `double`

`Holiday` — Holidays and nontrading-day dates
non-trading day vector is determined by the routine `holidays` (default) | serial date number | date character vector | datetime array

Holidays and nontrading-day dates, specified as vector.

All dates in `Holiday` must be the same format: either serial date numbers, or date character vectors, or datetime arrays. (Using serial date numbers improves performance.) The `holidays` function supplies the default vector.

If `Holiday` is a datetime array, then `Date` is returned as a datetime array. If `outputType` is specified, then its value determines the output type of `Date`. This overrides any influence of `Holiday`.

Data Types: `double` | `char` | `datetime`

`Weekend` — Weekend days
`[1 0 0 0 0 0 1]` (Saturday and Sunday form the weekend) (default) | vector of length 7, containing `0` and `1`, where `1` indicates weekend days

Weekend days, specified as a vector of length 7, containing `0` and `1`, where `1` indicates weekend days and the first element of this vector corresponds to Sunday.

Data Types: `double`

`outputType` — Year to determine days
`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

A character vector specified as either `'datenum'` or `'datetime'`. The output `Date` is in serial date format if `'datenum'` is specified, or datetime format if `'datetime'` is specified. By default the output `Date` is in serial date format, or match the format of `Holiday`, if specified.

Data Types: `char`

Output Arguments

`Date` — Date for the last business date of given year and month

serial date number | date character vector | datetime array

Date for the last business date of a given year and month, returned as a serial date number, date character vector, or datetime array.

If neither `Holiday` nor `outputType` are specified, `Date` is returned as a serial date number. If `Holiday` is specified, but not `outputType`, then the type of the holiday variable controls the type of date:

- If `Holiday` is a serial date number or date character vector, then `Date` is returned as a serial date number
- If `Holiday` is a datetime array, then `Date` is returned as a datetime array.

.

See Also

`busdate` | `datetime` | `eomdate` | `fbusdate` | `holidays` | `isbusday`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

leadts

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`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

length

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

`length` | `size`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

lifetableconv

Convert life table series into life tables with forced termination

Syntax

```
[qx,lx,dx] = lifetableconv(x0,lx0)

[qx,lx,dx] = lifetableconv(x0,y0,y0type)
```

Description

`[qx,lx,dx] = lifetableconv(x0,lx0)` converts life table with ages `x0` and survival counts `lx0` into life tables with termination.

`[qx,lx,dx] = lifetableconv(x0,y0,y0type)` converts life table with ages `x0` and series `y0`, specified by the optional argument `y0type`, into life tables with termination.

Examples

Convert Life Table Series into Life Tables with Forced Termination

Load the life table data file.

```
load us_lifetable_2009
```

Convert life table series into life tables with forced termination.

```
[qx,lx,dx] = lifetableconv(x,lx);
display(qx(1:20,:))
```

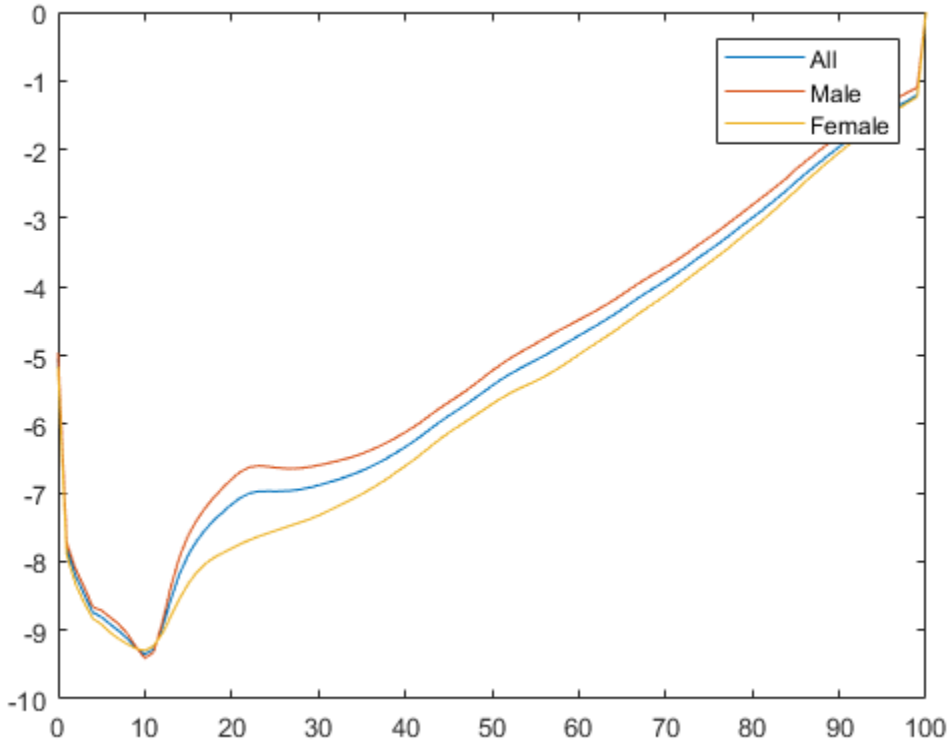
```
    0.0064    0.0070    0.0057
    0.0004    0.0004    0.0004
    0.0003    0.0003    0.0002
    0.0002    0.0002    0.0002
    0.0002    0.0002    0.0001
```

```
    0.0001    0.0002    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0002    0.0002    0.0002
    0.0003    0.0004    0.0002
    0.0004    0.0005    0.0002
    0.0005    0.0006    0.0003
    0.0005    0.0007    0.0003
    0.0006    0.0009    0.0004
    0.0007    0.0010    0.0004

display(lx(1:20,:))

   1.0e+05 *

    1.0000    1.0000    1.0000
    0.9936    0.9930    0.9943
    0.9932    0.9926    0.9939
    0.9930    0.9923    0.9937
    0.9927    0.9920    0.9935
    0.9926    0.9919    0.9933
    0.9924    0.9917    0.9932
    0.9923    0.9916    0.9931
    0.9922    0.9914    0.9930
    0.9921    0.9913    0.9929
    0.9920    0.9912    0.9928
    0.9919    0.9911    0.9927
    0.9918    0.9910    0.9926
    0.9917    0.9909    0.9925
    0.9915    0.9907    0.9923
    0.9912    0.9903    0.9921
    0.9908    0.9898    0.9919
    0.9904    0.9892    0.9916
    0.9899    0.9885    0.9913
    0.9892    0.9876    0.9909

display(dx(1:20,:))

  637.2266  698.8750  572.6328
   40.4063   43.9297   36.7188
```

```
   27.1875   30.0938   24.1406
   20.7656   23.0781   18.3359
   15.9141   17.2109   14.5625
   14.8672   16.3125   13.3516
   13.3672   14.7891   11.8750
   12.1328   13.3828   10.8203
   10.8125   11.6094    9.9844
    9.4609    9.5781    9.3438
    8.6172    8.1328    9.1172
    9.2656    8.8359    9.7188
   12.5938   13.5078   11.6328
   19.1016   22.9844   15.0234
   27.6719   35.5938   19.3516
   36.6328   48.5703   24.0547
   45.0156   60.7109   28.4844
   53.1406   72.8906   32.2813
   60.8984   85.1172   35.2578
   68.3438   97.2266   37.6875
```

Plot the `qx` series and display the legend. The series `qx` is the conditional probability that a person at age *x* will die between age *x* and the next age in the series.

```
plot(x,log(qx))
legend(series)
```



Convert the Life Table dx Series After Fitting and Generating the Life Table Series

Load the life table data file.

```
load us_lifetable_2009
```

Calibrate life table from survival data with the default Heligman-Pollard parametric model.

```
a = lifetablefit(x,lx)
```

```
a =

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0006
    6.2840    6.7636    1.1274
   24.1387   24.2897   52.1149
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1098
```

Generate life table series from the calibrated mortality model.

```
qx = lifetablegen((0:120),a);
display(qx(1:20,:))
```

```
    0.0063    0.0069    0.0057
    0.0005    0.0006    0.0004
    0.0002    0.0003    0.0002
    0.0002    0.0002    0.0002
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0002    0.0002    0.0001
    0.0002    0.0002    0.0002
    0.0002    0.0003    0.0002
    0.0003    0.0004    0.0002
    0.0004    0.0005    0.0002
    0.0005    0.0006    0.0003
    0.0006    0.0008    0.0003
    0.0007    0.0009    0.0003
```

Convert life table series into life tables with forced termination.

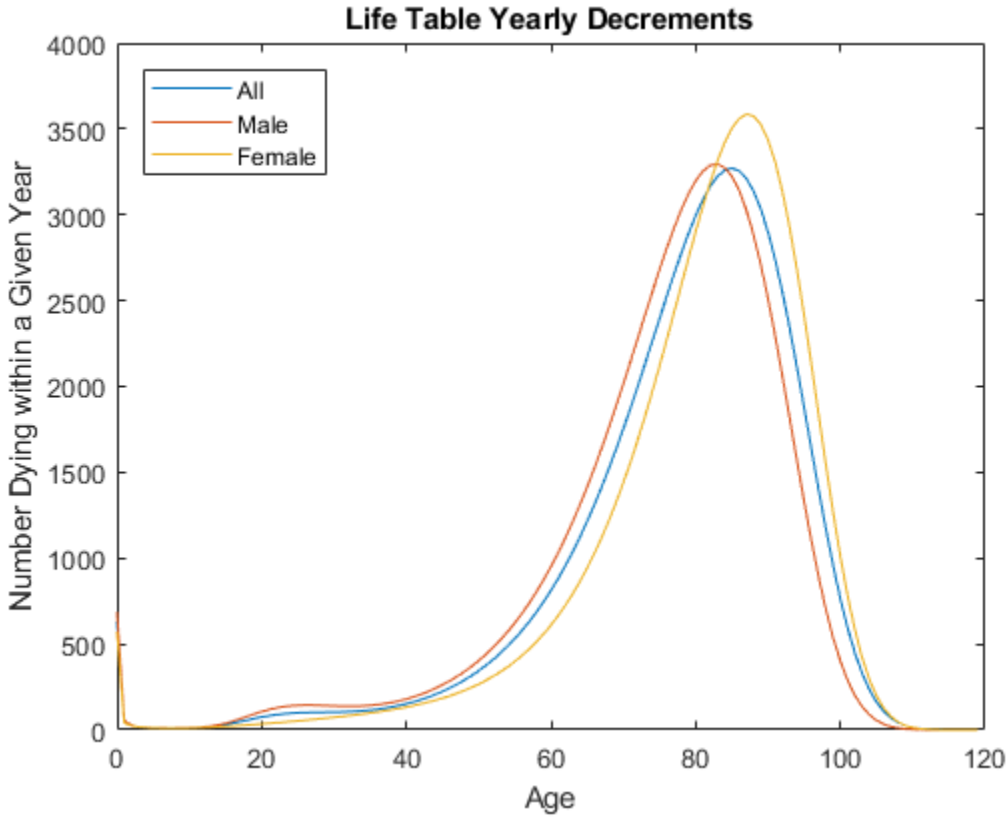
```
[~,~,dx] = lifetableconv((0:120),qx,'qx');
display(dx(1:20,:))
```

```
  630.9953  686.9460  571.6087
   48.7927   55.1030   40.9755
```

```
   24.8017   26.3780   23.6150
   17.0832   17.5879   17.0354
   13.6181   13.8189   13.6205
   11.8661   12.0077   11.6377
   10.9781   11.1573   10.4946
   10.5995   10.8605    9.9493
   10.5757   10.9396    9.8919
   10.8789   11.3612   10.2653
   11.6084   12.2508   11.0339
   12.9922   13.9271   12.1691
   15.3480   16.8834   13.6440
   18.9940   21.6791   15.4312
   24.1395   28.7662   17.5027
   30.8009   38.3209   19.8304
   38.7717   50.1481   22.3869
   47.6536   63.6898   25.1462
   56.9302   78.1258   28.0842
   66.0576   92.5245   31.1798
```

Plot the `dx` series and display the legend. The series `dx` is the number of people who die out of 100,000 alive at birth between age *x* and the next age in the series.

```
plot((0:119),dx(1:end-1,:));
legend(series, 'location', 'northwest');
title('\bfLife Table Yearly Decrements');
xlabel('Age');
ylabel('Number Dying within a Given Year');
```



- “Case Study for Life Tables Analysis” on page 2-56

Input Arguments

`x0` — Increasing ages for raw data

vector of nonnegative integer values

Increasing ages for raw data, specified as nonnegative integer values in an `N0` vector.

The vector of ages *x* must contain nonnegative integer values. If the input series is the discrete survival function `lx`, then the starting age need only be nonnegative. Otherwise, the starting age must be `0`.

Data Types: `double`

`lx0` — Collection of `num` standardized survivor series
matrix

Collection of `num` standardized survivor series, specified as an `N0`-by-`num` matrix. The input `lx0` series is the number of people alive at age *x*, given 100,000 alive at birth. Values of `0` or `NaN` in the input table `lx0` are ignored.

Data Types: `double`

`y0` — Collection of `num` life table series to be converted
matrix

Collection of `num` life table series to be converted, specified as an `N0`-by-`num` matrix. The default `y0` series is `lx0`.

Data Types: `double`

`y0type` — Type of mortality series for input `y0` with default `'lx'`
`'lx'` (default) | character vector with values `'qx'`, `'lx'`, `'dx'`

(Optional) Type of mortality series for input `y0`, specified as a character vector with one of the following values:

- `'qx'` — Input is a table of discrete hazards (`qx`).
- `'lx'` — Input is a table of discrete survival counts (`lx`).
- `'dx'` — Input is a table of discrete decrements (`dx`).

Whereas the output series have forced termination, the input series (`y0`) can have one of several types of termination:

- Natural termination runs out to the last person so that `lx` goes to `0`, `qx` goes to `1`, and `dx` goes to `0`. For more information, see “Natural Termination” on page 18-1092.
- Truncated termination stops at a terminal age so that `lx` is positive, `qx` is less than `1`, and `dx` is positive. Any ages after the terminal age are `NaN` values. For more information, see “Truncated Termination” on page 18-1093.

Data Types: `char`

Output Arguments

`qx` — Discrete hazard function

matrix

Discrete hazard function, returned as an `N0`-by-`num` matrix with forced termination. For more information, see “Forced Termination” on page 18-1091.

The series `qx` is the conditional probability that a person at age x will die between age *x* and the next age in the series.

`lx` — Discrete survival function

matrix

Discrete survival function, returned as an `N0`-by-`num` matrix with forced termination. For more information, see “Forced Termination” on page 18-1091.

The series `lx` is the number of people alive at age *x*, given 100,000 alive at birth.

`dx` — Discrete decrements function

matrix

Discrete decrements function, returned as an `N0`-by-`num` matrix with forced termination. For more information, see “Forced Termination” on page 18-1091.

The series `dx` is the number of people who die out of 100,000 alive at birth, between age *x* and the next age in the series.

Definitions

Forced Termination

Most modern life tables have “forced” termination. Forced termination means that the last row of the life table applies for all persons with ages on or after the last age in the life table.

This sample illustrates forced termination.

United States Life Tables 2009

x	l_x WM	l_x WF	l_x BM	l_x BF	q_x WM	q_x WF	q_x BM	q_x BF
93	9533	18368	6615	15685	0.219553	0.177156	0.200605	0.157858
94	7440	15114	5288	13209	0.23871	0.194852	0.214448	0.170868
95	5664	12169	4154	10952	0.258475	0.213411	0.229177	0.184624
96	4200	9572	3202	8930	0.27881	0.232971	0.24391	0.198992
97	3029	7342	2421	7153	0.299439	0.253337	0.259397	0.214036
98	2122	5482	1793	5622	0.320452	0.27417	0.274958	0.229634
99	1442	3979	1300	4331	0.341886	0.295803	0.291538	0.245671
100+	949	2802	921	3267	1	1	1	1

In this case, the last row of the life table applies for all persons aged 100 or older. Specifically, q_x probabilities are ${}_1q_x$ for ages less than 100 and, technically, ${}_\infty q_x$ for age 100.

Forced termination has terminal age values that apply to all ages after the terminal age so that `lx` is positive, `qx` is `1`, and `dx` is positive. Ages after the terminal age are `NaN` values, although `lx` and `dx` can be `0` and `qx` can be `1` for input series. Forced termination is triggered by a naturally terminating series, the last age in a truncated series, or the first `NaN` value in a series.

Natural Termination

Before 1970, life tables were often published with data that included all ages for which persons associated with a given series were still alive. Tables in this form have "natural" termination. In natural termination, the last row of the life table for each series counts the deaths or probabilities of deaths of the last remaining person at the corresponding age. Tables in this form can be problematic due to the granularity of the data and the fact that groups of series can terminate at distinct ages. Natural termination is illustrated in the following sample of the last few years of a life table.

United States Life Tables 1940

x	l_x WM	l_x WF	l_x BM	l_x BF	q_x WM	q_x WF	q_x BM	q_x BF
103	14	29	59	234	0.428571	0.413793	0.389831	0.333333
104	8	17	36	156	0.5	0.470588	0.416667	0.358974
105	4	9	21	100	0.5	0.444444	0.47619	0.38
106	2	5	11	62	0.5	0.4	0.454545	0.403226
107	1	3	6	37	0	0.666667	0.5	0.432432
108	1	1	3	21	1	0	0.666667	0.47619
109		1	1	11		1	1	0.454545
110				6				0.5
111				3				0.666667
112				1				0
113				1				1
114								

This form for life tables poses a number of issues that go beyond the obvious statistical issues. First, the l_x table on the left terminates at ages 108, 109, 109, and 113 for the four series in the table. Technically, the numbers after these ages are `0`, but can also be `NaN` values because no person is alive after these terminating ages. Second, the probabilities q_x on the right terminate with fluctuating probabilities that go from `0` to `1` in some cases. In this case, however, all probabilities are ${}_1q_x$ probabilities (unlike the forced termination probabilities). You can argue that the probabilities after the ages of termination can be `1` (anyone alive at this age is expected to die in the next year), `0` (the age lies outside the support of the probability distribution), or `NaN` values.

Truncated Termination

Truncated termination occurs with truncation of life tables at an arbitrary age. For example, from 1970–1990, United States life tables truncated at age 85. This format is problematic because life table probabilities must either terminate with probability `1` (forced termination) or discard data that exceeds the terminating age. This sample of the last few years of a life table illustrates truncated termination. The raw data for this table is the l_x series. The q_x series is derived from this series.

United States Life Tables 1980

x	l_x WM	l_x WF	l_x BM	l_x BF	q_x WM	q_x WF	q_x BM	q_x BF
78	38558	61100	25481	45776	0.078064	0.045221	0.082689	0.054789
79	35548	58337	23374	43268	0.084477	0.050037	0.088303	0.059351
80	32545	55418	21310	40700	0.091504	0.055542	0.094603	0.064619
81	29567	52340	19294	38070	0.099165	0.061922	0.101741	0.070607
82	26635	49099	17331	35382	0.107528	0.06935	0.109746	0.077356
83	23771	45694	15429	32645	0.116613	0.078194	0.118608	0.084822
84	20999	42121	13599	29876	0.126339	0.081788	0.128392	0.093018
85	18346	38676	11853	27097				

This life table format poses problems for termination because, for example, over 27% of the population for the fourth l_x series is still alive at age 85. To claim that the probability of dying for all ages after age 85 is 100% might be true but is uninformative. Notwithstanding the statistical issues, however, tables in this form are completed by forced termination.

References

[1] Arias, E. "United States Life Tables." *National Vital Statistics Reports, U.S. Department of Health and Human Services.* Vol. 62, No. 7, 2009.

See Also

`lifetablefit` | `lifetablegen`

Topics

"Case Study for Life Tables Analysis" on page 2-56
"About Life Tables" on page 2-53

Introduced in R2015a

lifetablefit

Calibrate life table from survival data with parametric models

Syntax

```
[a,elx] = lifetablefit(x,lx)
[a,elx] = lifetablefit(___,lifemodel,objtype,interpmethod,a0)
```

Description

`[a,elx] = lifetablefit(x,lx)` calibrates a life table, `x`, from survival data, `lx`, using parametric models.

`[a,elx] = lifetablefit(___,lifemodel,objtype,interpmethod,a0)` calibrates a life table, `x`, from survival data, `lx`, using parametric models using optional arguments for `lifemodel`, `objtype`, `interpmethod`, and `a0`.

Examples

Calibrate Life Table from Survival Data Using a Heligman-Pollard Parametric Model

Load the life table data file.

```
load us_lifetable_2009
```

Calibrate the life table from survival data using the default `heligman-pollard` parametric model.

```
[a,elx] = lifetablefit(x,lx);
display(a)
```

```
a =

    0.0005    0.0006    0.0004
```

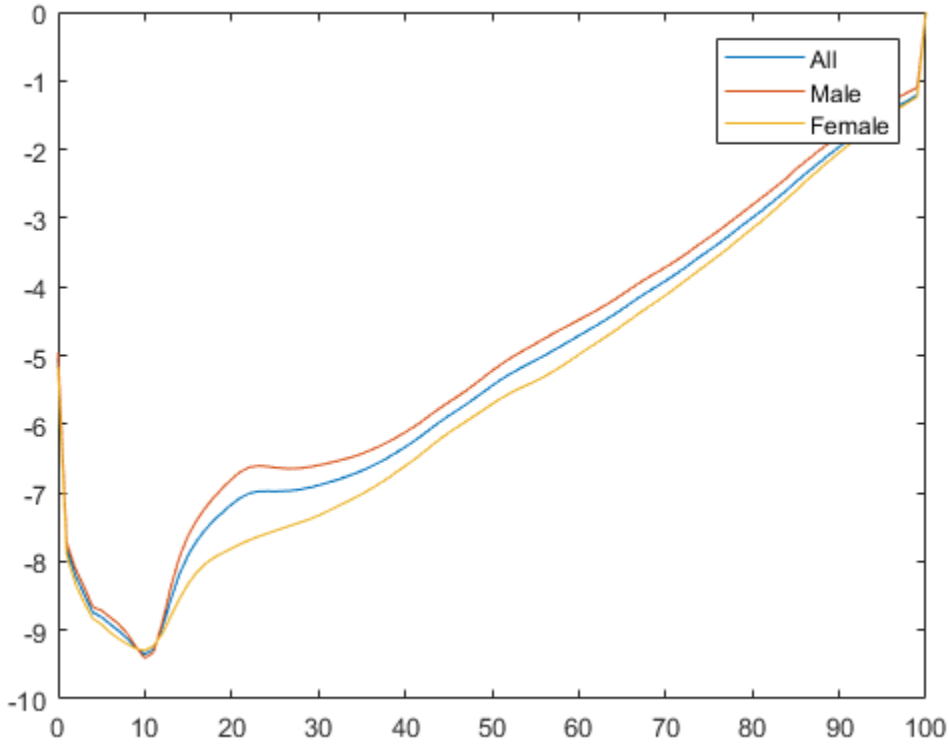
```
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0006
    6.2840    6.7636    1.1274
   24.1387   24.2897   52.1149
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1098
```

```
display(elx(1:20,:))
```

```
   1.0e+05 *

    1.0000    1.0000    1.0000
    0.9937    0.9931    0.9943
    0.9932    0.9926    0.9939
    0.9930    0.9923    0.9936
    0.9928    0.9921    0.9935
    0.9926    0.9920    0.9933
    0.9925    0.9919    0.9932
    0.9924    0.9918    0.9931
    0.9923    0.9917    0.9930
    0.9922    0.9916    0.9929
    0.9921    0.9914    0.9928
    0.9920    0.9913    0.9927
    0.9919    0.9912    0.9926
    0.9917    0.9910    0.9924
    0.9915    0.9908    0.9923
    0.9913    0.9905    0.9921
    0.9910    0.9901    0.9919
    0.9906    0.9896    0.9917
    0.9901    0.9890    0.9914
    0.9895    0.9882    0.9912
```

Plot the `qx` series and display the legend. The series `qx` is the conditional probability that a person at age *x* will die between age *x* and the next age in the series

```
plot(x,log(qx))
legend(series)
```



- “Case Study for Life Tables Analysis” on page 2-56

Input Arguments

x — Increasing ages for raw data

vector of nonnegative integers

Increasing ages for raw data, specified as a `N` vector for nonnegative integers.

Data Types: `double`

`lx` — Collection of `num` discrete survival counts

matrix

Collection of `num` discrete survival counts, specified as an `N`-by-`num` matrix. The input `lx` series is the number of people alive at age x, given 100,000 alive at birth. Values of `0` or `NaN` in the input table `lx` are ignored.

Data Types: `double`

`lifemodel` — Parametric mortality model type

`'heligman-pollard'` (default) | character vector with values `'heligman-pollard'`, `'heligman-pollard-2''heligman-pollard-3'`, `'gompertz'`, `'makeham'`, `'siler'`

(Optional) Parametric mortality model type, specified as a character vector with one of the following values:

- `'heligman-pollard'` — Eight-parameter Heligman-Pollard model (version 1), specified in terms of the discrete hazard function:

  $$\frac{q(x)}{1-q(x)} = A^{(x+B)^C} + D\exp(-E(\log\frac{x}{F})^2) + GH^X$$

  for ages $x \geq 0$, with parameters A, B, C, D, E, F, G, $H \geq 0$.

- `'heligman-pollard-2'` — Eight-parameter Heligman-Pollard model (version 2), specified in terms of the discrete hazard function:

  $$\frac{q(x)}{1-q(x)} = A^{(x+B)^C} + D\exp(-E(\log\frac{x}{F})^2) + \frac{GH^X}{1+GH^X}$$

  for ages $x \geq 0$, with parameters A, B, C, D, E, F, G, $H \geq 0$.

- `'heligman-pollard-3'` — Eight-parameter Heligman-Pollard model (version 3), specified in terms of the discrete hazard function:

  $$q(x) = A^{(x+B)^C} + D\exp(-E(\log\frac{x}{F})^2) + GH^X$$

  for ages $x \geq 0$, with parameters A, B, C, D, E, F, G, $H \geq 0$.

- `'gompertz'` — Two-parameter Gompertz model, specified in terms of the continuous hazard function:

  ```
  h(x) = A exp(Bx)
  ```

  for ages $x \geq 0$, with parameters A, $B \geq 0$.

- `'makeham'` — Three-parameter Gompertz-Makeham model, specified in terms of the continuous hazard function:

  ```
  h(x) = A exp(Bx) + C
  ```

  for ages $x \geq 0$, with parameters$A$, B, $C \geq 0$.
- `'siler'` — Five-parameter Siler model, specified in terms of the continuous hazard function:

  ```
  h(x) = A exp(Bx) + C + D exp(-Ex)
  ```

  for ages $x \geq 0$, with parameters A, B, C, D, $E \geq 0$.

Data Types: `char`

`objtype` — Objective for nonlinear least-squares estimation

`'ratio'` (default) | character vector with values `'ratio'` `'logratio'`

(Optional) Objective for nonlinear least-squares estimation, specified as a character vector with the following values:

- `'ratio'` — Given raw data q_x and model estimates $\hat{q}_x$ for $x = 1, \ldots, N$, the first objective (which is the preferred objective) has the form

  $$\Phi = \sum_{x=1}^{N} \left(1 - \frac{\hat{q}_x}{q_x}\right)^2$$

- `'logratio'` — Given raw data q_x and model estimates $\hat{q}_x$ for $x = 1, \ldots, N$, the second objective has the form

  $$\Phi = \sum_{x=1}^{N} \left(\log(q_x) - \log(q_x)\right)^2$$

Data Types: `char`

`interpmethod` — Interpolation method to use for abridged life table inputs

`'cubic'` (default) | character vector with values `'cubic'`, `'linear'`, `'none'`

(Optional) Interpolation method to use for abridged life table inputs, specified as a character vector with the following values:

- `'cubic'` — Cubic interpolation that uses `'pchip'` method in `interp1`.
- `'linear'` — Linear interpolation.
- `'none'` — No interpolation.

Note If the ages in `x` are not consecutive years and interpolation is set to `'none'`, then the estimates for the parameters are suitable only for the age vector `x`.

If you use the parameter estimates to compute life table values for arbitrary years, interpolate using the default `'cubic'` method.

Interpolation with abridged life tables forms internal interpolated full life tables, which usually improves model fits.

Data Types: `char`

`a0` — Initial parameter estimate to be applied to all series

vector

(Optional) Initial parameter estimate to be applied to all series, specified as a `numparam` vector. This vector must conform to the number of parameters in the model specified using the `lifemodel` argument.

Data Types: `double`

Output Arguments

`a` — Parameter estimates for each `num` series

matrix

Parameter estimates for each `num` series, returned as a `numparam`-by-`num` matrix.

`elx` — Estimated collection of `num` standardized survivor series

matrix

Estimated collection of `num` standardized survivor series, returned as an `N`-by-`num` matrix. The `elx` output series is the number of people alive at age *x*, given 100,000 alive at birth. Values of `0` or `NaN` in the input table `lx` are ignored.

References

[1] Arias, E. “United States Life Tables.” *National Vital Statistics Reports, U.S. Department of Health and Human Services.* Vol. 62, No. 7, 2009.

[2] Carriere, F. “Parametric Models for Life Tables.” *Transactions of the Society of Actuaries.* Vol. 44, 1992, pp. 77–99.

[3] Gompertz, B. “On the Nature of the Function Expressive of the Law of Human Mortality, and on a New Mode of Determining the Value of Life Contingencies.” *Philosophical Transactions of the Royal Society.* Vol. 115, 1825, pp. 513–582.

[4] Heligman, L. M. .A., and J. H. Pollard. “The Age Pattern of Mortality.” *Journal of the Institute of Actuaries* Vol. 107, Pt. 1, 1980, pp. 49–80.

[5] Makeham, W .M. “On the Law of Mortality and the Construction of Annuity Tables.” *Journal of the Institute of Actuaries* Vol. 8, 1860 . pp. 301–310.

[6] Siler, W. “A Competing-Risk Model for Animal Mortality.” *Ecology* Vol. 60, pp. 750–757, 1979.

[7] Siler, W. “Parameters of Mortality in Human Populations with Widely Varying Life Spans.” *Statistics in Medicine* Vol. 2, 1983, pp. 373–380.

See Also

`lifetableconv` | `lifetablegen`

Topics

“Case Study for Life Tables Analysis” on page 2-56
“About Life Tables” on page 2-53

Introduced in R2015a

lifetablegen

Generate life table series from calibrated mortality model

Syntax

```
[qx,lx,dx] = lifetablegen(x,a)
[qx,lx,dx] = lifetablegen(x,a,lifemodel)
```

Description

`[qx,lx,dx] = lifetablegen(x,a)` generates a life table series from a calibrated mortality model.

`[qx,lx,dx] = lifetablegen(x,a,lifemodel)` generates a life table series from a calibrated mortality model using the optional argument for `lifemodel`.

Examples

Generate Life Table Series from a Calibrated Mortality Model for Heligman-Pollard

Load the life table data file.

```
load us_lifetable_2009
```

Calibrate the life table from survival data using the default `heligman-pollard` parametric model.

```
a = lifetablefit(x, lx)
```

```
a =

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
```

```
    0.0007    0.0011    0.0006
    6.2840    6.7636    1.1274
   24.1387   24.2897   52.1149
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1098
```

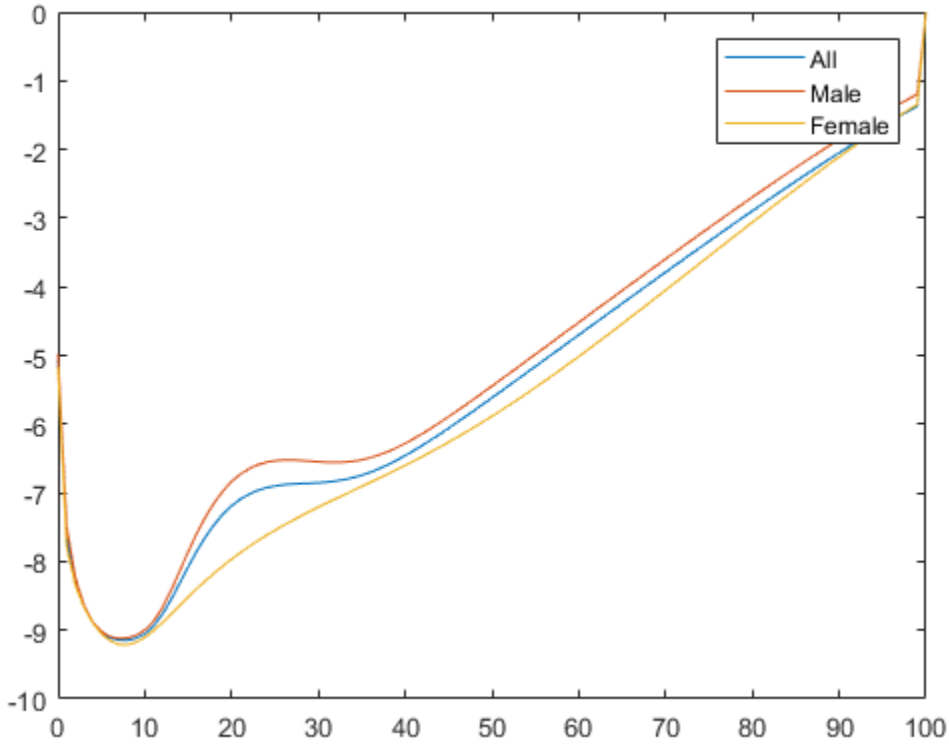
Generate a life table series from the calibrated mortality model.

```
qx = lifetablegen(x,a);
display(qx(1:20,:))
```

```
    0.0063    0.0069    0.0057
    0.0005    0.0006    0.0004
    0.0002    0.0003    0.0002
    0.0002    0.0002    0.0002
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0002    0.0002    0.0001
    0.0002    0.0002    0.0002
    0.0002    0.0003    0.0002
    0.0003    0.0004    0.0002
    0.0004    0.0005    0.0002
    0.0005    0.0006    0.0003
    0.0006    0.0008    0.0003
    0.0007    0.0009    0.0003
```

Plot the `qx` series and display the legend. The series `qx` is the conditional probability that a person at age *x* will die between age *x* and the next age in the series.

```
plot(x,log(qx))
legend(series)
```



Prepare Life Table Data Using the qx Series and Generate the qx Life Table

Load the life table data file.

```
load us_lifetable_2009
```

Convert the life table series into life tables with forced termination.

```
[~, lx] = lifetableconv(x, qx, 'qx');
```

Calibrate the life table from survival data using the default `heligman-pollard` parametric model.

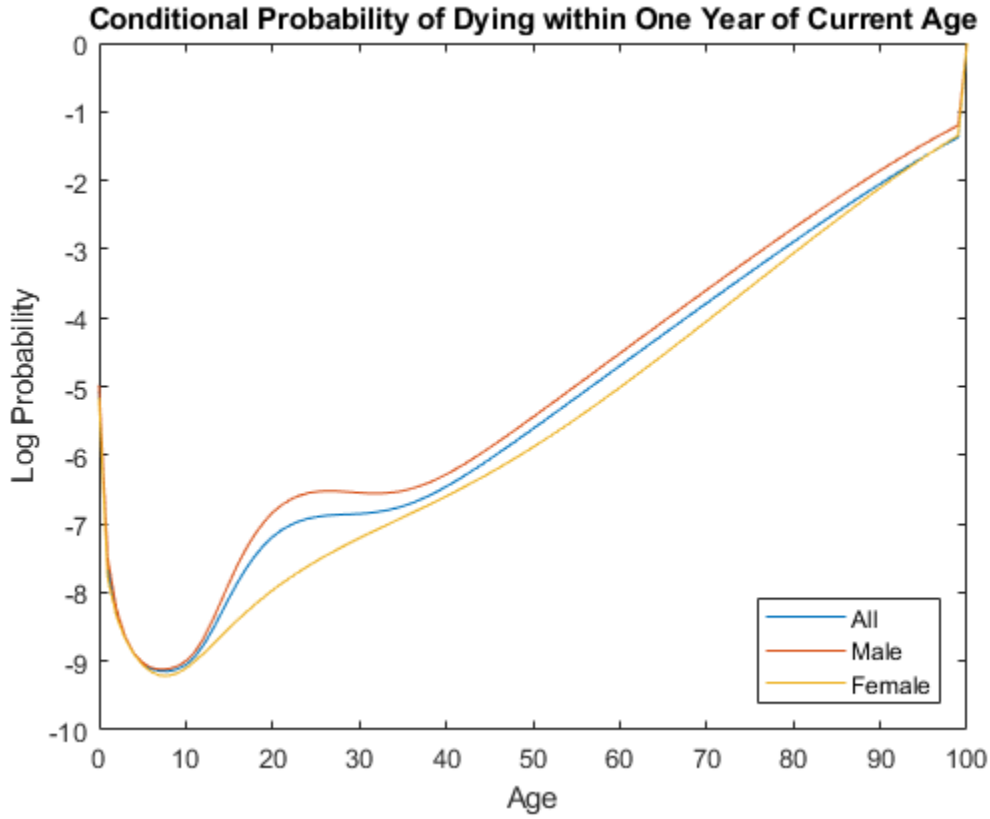
```
a = lifetablefit(x, lx)

a =

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0007
    6.2854    6.7638    1.1032
   24.1384   24.2895   53.2068
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1100
```

Generate a life table series from the calibrated mortality model.

```
qx = lifetablegen((0:100), a)

qx =

    0.0063    0.0069    0.0057
    0.0005    0.0006    0.0004
    0.0002    0.0003    0.0002
    0.0002    0.0002    0.0002
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
    0.0001    0.0001    0.0001
```

Plot the `qx` series and display the legend. The series `qx` is the conditional probability that a person at age *x* will die between age *x* and the next age in the series.

```
plot((0:100), log(qx));
legend(series, 'location', 'southeast');
title('Conditional Probability of Dying within One Year of Current Age');
xlabel('Age');
ylabel('Log Probability');
```



- “Case Study for Life Tables Analysis” on page 2-56

Input Arguments

`x` — Increasing ages for raw data

vector of nonnegative integers

Increasing ages for raw data, specified as a `N` vector of nonnegative integer values. The ages must start at `0` (birth).

Data Types: `double`

`a` — Model parameters for `num` models

matrix

Model parameters for `num` models, specified as a `numparam`-by-`num` matrix, where the number of parameters (`numparam`) depends on the model specified using the `lifemodel` argument.

Data Types: `double`

`lifemodel` — Parametric mortality model type

`'heligman-pollard'` (default) | character vector with values `'heligman-pollard'`, `'heligman-pollard-2''heligman-pollard-3'`, `'gompertz'`, `'makeham'`, `'siler'`

(Optional) Parametric mortality model type, specified as a character vector with one of the following values:

- `'heligman-pollard'` — Eight-parameter Heligman-Pollard model (version 1), specified in terms of the discrete hazard function:

  $$\frac{q(x)}{1-q(x)} = A^{(x+B)^C} + D\exp(-E(\log\frac{x}{F})^2) + GH^X$$

  for ages $x \geq 0$, with parameters A, B, C, D, E, F, G, $H \geq 0$.

- `'heligman-pollard-2'` — Eight-parameter Heligman-Pollard model (version 2), specified in terms of the discrete hazard function:

  $$\frac{q(x)}{1-q(x)} = A^{(x+B)^C} + D\exp(-E(\log\frac{x}{F})^2) + \frac{GH^X}{1+GH^X}$$

  for ages $x \geq 0$, with parameters A, B, C, D, E, F, G, $H \geq 0$.

- `'heligman-pollard-3'` — Eight-parameter Heligman-Pollard model (version 3), specified in terms of the discrete hazard function:

  $$q(x) = A^{(x+B)^C} + D\exp(-E(\log\frac{x}{F})^2) + GH^X$$

  for ages $x \geq 0$, with parameters A, B, C, D, E, F, G, $H \geq 0$.

- `'gompertz'` — Two-parameter Gompertz model, specified in terms of the continuous hazard function:

  ```
  h(x) = A exp(Bx)
  ```

  for ages $x \geq 0$, with parameters A, $B \geq 0$.

- `'makeham'` — Three-parameter Gompertz-Makeham model, specified in terms of the continuous hazard function:

  ```
  h(x) = A exp(Bx) + C
  ```

  for ages $x \geq 0$, with parameters$A$, B, $C \geq 0$.
- `'siler'` — Five-parameter Siler model, specified in terms of the continuous hazard function:

  ```
  h(x) = A exp(Bx) + C + D exp(-Ex)
  ```

  for ages $x \geq 0$, with parameters A, B, C, D, $E \geq 0$.

Data Types: `char`

Output Arguments

`qx` — Conditional probabilities of dying for `N` ages and `num` series

matrix

Conditional probabilities of dying for `N` ages and `num` series, returned as an `N`-by-`num` matrix. The series `qx` is the conditional probability that a person at age x will die between age x and the next age in the series. For the last age, `qx` represents probabilities or counts for all ages after the last age.

The last row of the `N`-by-`num` output for `qx` is the values for all ages on or after the last age in x (due to “Forced Termination” on page 18-1109). Therefore, the last row of `qx` contains `1` (100% probability of dying on or after the last age).

`lx` — Survival counts for `N` ages and `num` series

matrix

Survival counts for `N` ages and `num` series, returned as an `N`-by-`num` matrix. The series `lx` is the number of people alive at age x, given 100,000 alive at birth.

`dx` — Decrement counts for `N` ages and `num` series

matrix

Decrement counts for `N` ages and `num` series, returned as an `N`-by-`num` matrix. The series `dx` is the number of people out of 100,000 alive at birth who die between age x and the

next age in the series. For the last age, `dx` represent probabilities or counts for all ages after the last age.

The last row of the `N`-by-`num` output for `dx` are values for all ages on or after the last age in x (due to "Forced Termination" on page 18-1109). Therefore, the last row of `dx` contains the remaining count of 100,000 people alive at birth who have not died by the last age.

Definitions

Forced Termination

Most modern life tables have "forced" termination. Forced termination means that the last row of the life table applies for all persons with ages on or after the last age in the life table.

This sample illustrates forced termination.

United States Life Tables 2009

x	l_x WM	l_x WF	l_x BM	l_x BF	q_x WM	q_x WF	q_x BM	q_x BF
93	9533	18368	6615	15685	0.219553	0.177156	0.200605	0.157858
94	7440	15114	5288	13209	0.23871	0.194852	0.214448	0.170868
95	5664	12169	4154	10952	0.258475	0.213411	0.229177	0.184624
96	4200	9572	3202	8930	0.27881	0.232971	0.24391	0.198992
97	3029	7342	2421	7153	0.299439	0.253337	0.259397	0.214036
98	2122	5482	1793	5622	0.320452	0.27417	0.274958	0.229634
99	1442	3979	1300	4331	0.341886	0.295803	0.291538	0.245671
100+	949	2802	921	3267	1	1	1	1

In this case, the last row of the life table applies for all persons aged 100 or older. Specifically, q_x probabilities are ${}_1q_x$ for ages less than 100 and, technically, ${}_\infty q_x$ for age 100.

Forced termination has terminal age values that apply to all ages after the terminal age so that `lx` is positive, `qx` is `1`, and `dx` is positive. Ages after the terminal age are `NaN` values, although `lx` and `dx` can be `0` and `qx` can be `1` for input series. Forced termination is triggered by a naturally terminating series, the last age in a truncated series, or the first `NaN` value in a series.

References

[1] Arias, E. “United States Life Tables.” *National Vital Statistics Reports, U.S. Department of Health and Human Services.* Vol. 62, No. 7, 2009.

[2] Carriere, F. “Parametric Models for Life Tables.” *Transactions of the Society of Actuaries.* Vol. 44, 1992, pp. 77–99.

[3] Gompertz, B. “On the Nature of the Function Expressive of the Law of Human Mortality, and on a New Mode of Determining the Value of Life Contingencies.” *Philosophical Transactions of the Royal Society.* Vol. 115, 1825, pp. 513–582.

[4] Heligman, L. M. .A., and J. H. Pollard. “The Age Pattern of Mortality.” *Journal of the Institute of Actuaries* Vol. 107, Pt. 1, 1980, pp. 49–80.

[5] Makeham, W .M. “On the Law of Mortality and the Construction of Annuity Tables.” *Journal of the Institute of Actuaries* Vol. 8, 1860 . pp. 301–310.

[6] Siler, W. “A Competing-Risk Model for Animal Mortality.” *Ecology* Vol. 60, pp. 750–757, 1979.

[7] Siler, W. “Parameters of Mortality in Human Populations with Widely Varying Life Spans.” *Statistics in Medicine* Vol. 2, 1983, pp. 373–380.

See Also

`lifetableconv` | `lifetablefit`

Topics

“Case Study for Life Tables Analysis” on page 2-56
“About Life Tables” on page 2-53

Introduced in R2015a

linebreak

Line break chart

Syntax

```
linebreak(X)
```

Arguments

`X`	`X` is a `M` -by-`2` matrix or table. If `X` is a `M` -by-`2` matrix, the first column contains date numbers and the second column is the asset price. If `X` is a table, the first column of the table contains the dates. The second column contains the asset price data. Dates can be either a serial date number, a date character vector, or a datetime array.

Description

`linebreak(X)` plots asset price with respect to dates.

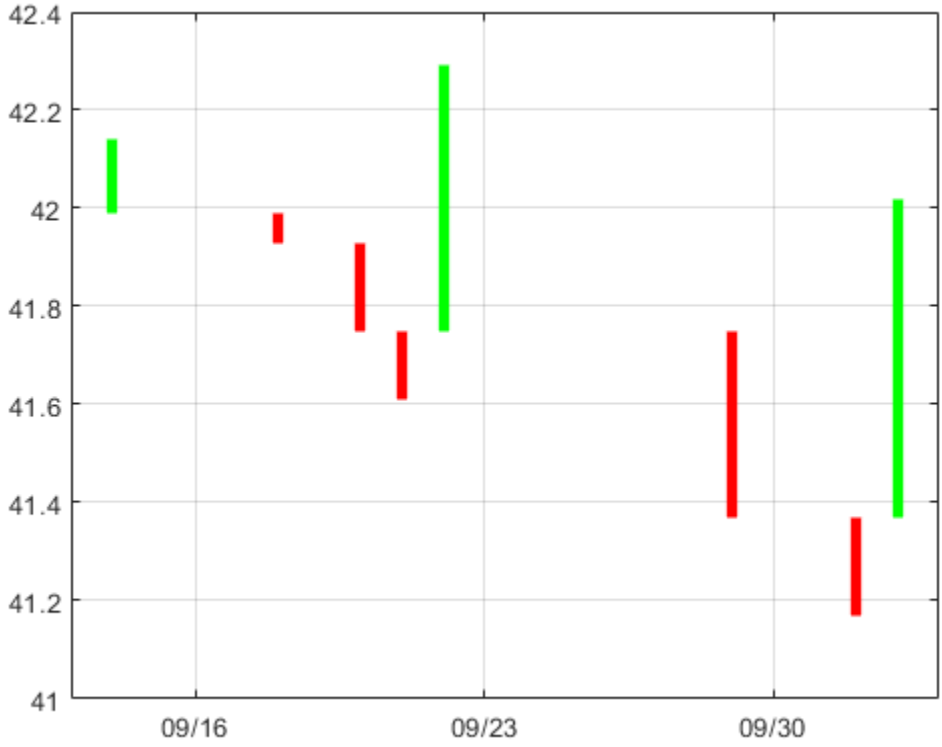
Examples

Create a Line Break Chart

This example shows how to generate a line break chart for asset X that is an M-by-2 matrix of date numbers and asset prices.

```
X = [...

733299.00         41.99;...
733300.00         42.14;...
733303.00         41.93;...
```

```
733304.00         41.98;...
733305.00         41.75;...
733306.00         41.61;...
733307.00         42.29;...
733310.00         42.19;...
733311.00         41.82;...
733312.00         41.93;...
733313.00         41.81;...
733314.00         41.37;...
733317.00         41.17;...
733318.00         42.02]

X =

   1.0e+05 *

    7.3330    0.0004
    7.3330    0.0004
    7.3330    0.0004
    7.3330    0.0004
    7.3331    0.0004
    7.3331    0.0004
    7.3331    0.0004
    7.3331    0.0004
    7.3331    0.0004
    7.3331    0.0004


linebreak(X)
```

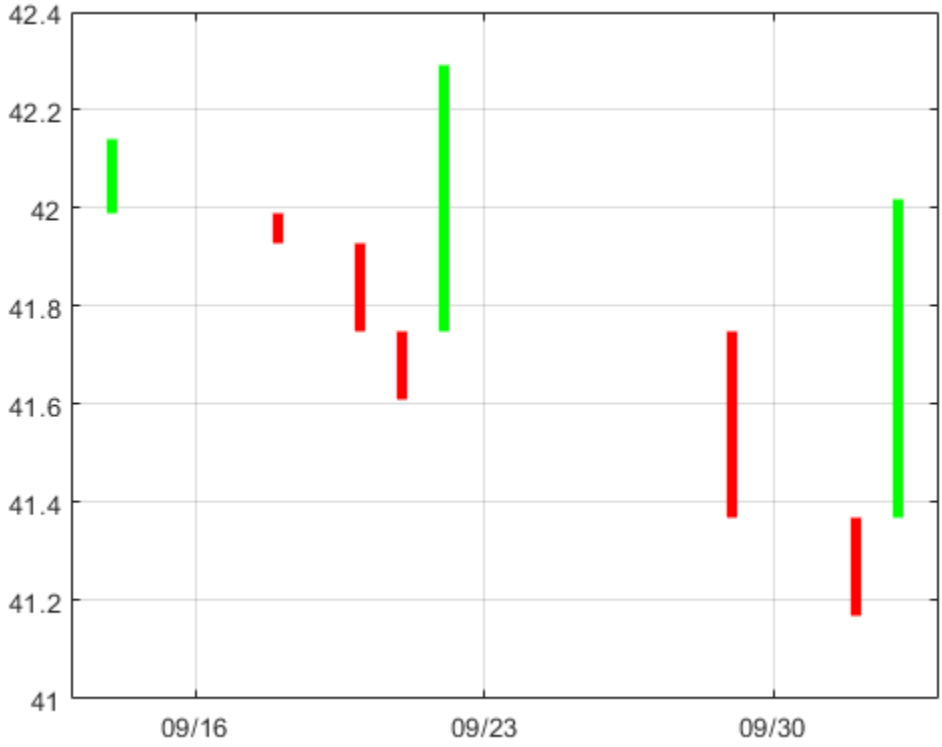


Create a Line Break Chart Using datetime Inputs

This example shows how to use `datetime` input to generate a line break chart for asset `X` that is an `M`-by-2 matrix of date numbers and asset prices.

```
X = [...

733299.00          41.99;...
733300.00          42.14;...
733303.00          41.93;...
733304.00          41.98;...
```

```
733305.00          41.75;...
733306.00          41.61;...
733307.00          42.29;...
733310.00          42.19;...
733311.00          41.82;...
733312.00          41.93;...
733313.00          41.81;...
733314.00          41.37;...
733317.00          41.17;...
733318.00          42.02];

dates = datetime(X(:,1),'ConvertFrom','datenum','Locale','en_US');
data = X(:,2);
t=table(dates,data);
linebreak(t)
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `datetime` | `highlow` | `kagi` | `movavg` | `pointfig` | `priceandvol` | `renko` | `volarea`

Topics

“Charting Financial Data” on page 2-14

Introduced in R2008a

llow

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.
`'ParameterName'`	The valid parameter name is: • `LowName`: low prices series name
`ParameterValue`	The parameter value is a character vector that represents the valid parameter name.

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.

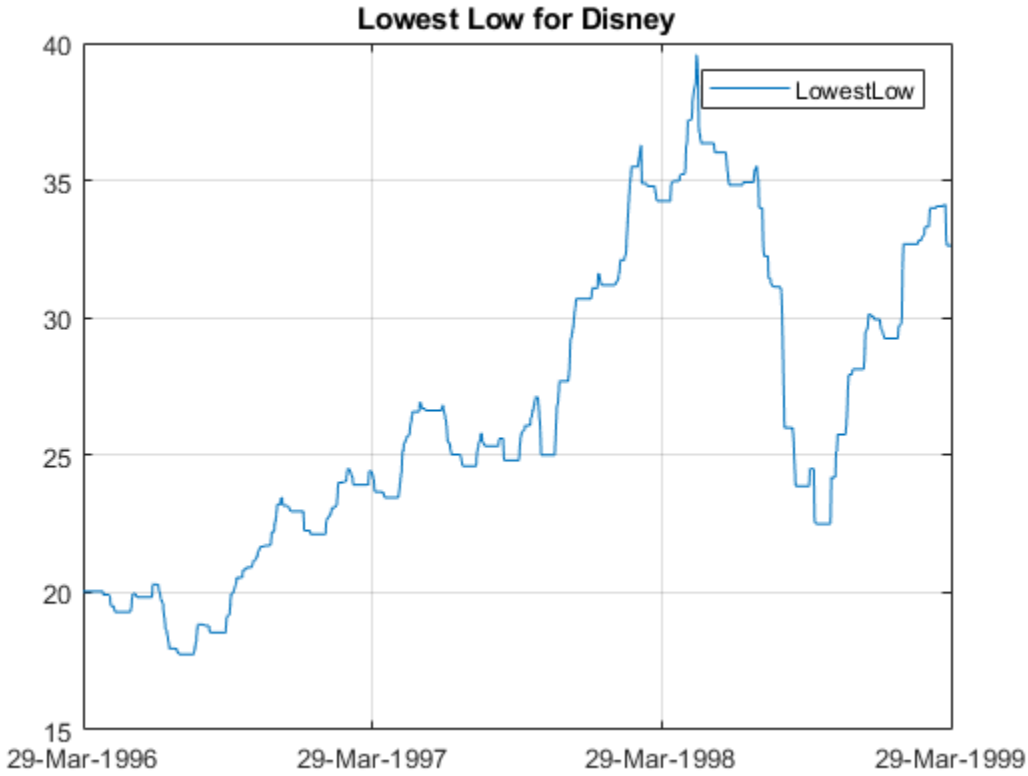
`llvts = llow(tsobj,nperiods,'ParameterName',ParameterValue, ...)` specifies the name for the required data series when it is different from the default name. The parameter value is a character vector that represents the valid parameter name.

Examples

Compute the Lowest Low Price

This example shows how to compute the lowest low price for Disney stock and plot the results.

```
load disney.mat
dis_LLow = llow(dis);
plot(dis_LLow)
title('Lowest Low for Disney')
```



- “Technical Analysis Examples” on page 16-4

See Also

`hhigh`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

log

Natural logarithm

Syntax

```
newfts = log(tsobj)
```

Description

`newfts = log(tsobj)` calculates the natural logarithm (log base e) of the data series in a financial time series object `tsobj`. It returns another time series object newfts containing the natural logarithms.

See Also

`exp` | `log10` | `log2`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

log10

Common logarithm

Syntax

```
newfts = log10(tsobj)
```

Description

`newfts = log10(tsobj)` calculates the common logarithm (base 10) of all the data in the data series of the financial time series object `tsobj` and returns the result in the object `newfts`.

See Also

`exp` | `log` | `log2`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

log2

Base 2 logarithm

Syntax

```
newfts = log2(tsobj)
```

Description

`newfts = log2(tsobj)` calculates the base 2 logarithm of the data series in a financial time series object `tsobj`. It returns another time series object `newfts` containing the logarithms.

See Also

`exp` | `log` | `log10`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

lpm

Compute sample lower partial moments of data

Syntax

```
lpm(Data)

lpm(Data,MAR)

lpm(Data,MAR,Order)

Moment = lpm(Data,MAR,Order)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` observations of `NUMSERIES` asset returns.
`MAR`	(Optional) Scalar minimum acceptable return (default `MAR` = `0`). This is a cutoff level of return such that all returns above `MAR` contribute nothing to the lower partial moment.
`Order`	(Optional) Either a scalar or a `NUMORDERS` vector of nonnegative integer moment orders. If no order specified, default `Order` = `0`, which is the shortfall probability. Although this function works for noninteger orders and, in some cases, for negative orders, this falls outside customary usage.

Description

Given `NUMSERIES` assets with `NUMSAMPLES` returns in a `NUMSAMPLES`-by-`NUMSERIES` matrix `Data`, a scalar minimum acceptable return `MAR`, and one or more nonnegative moment orders in a `NUMORDERS` vector `Order`, `lpm` computes lower partial moments relative to `MAR` for each asset in a `NUMORDERS x NUMSERIES` matrix `Moment`.

The output `Moment` is a `NUMORDERS x NUMSERIES` matrix of lower partial moments with `NUMORDERS` `Order`s and `NUMSERIES` series, that is, each row contains lower partial moments for a given order.

Note To compute upper partial moments, reverse the signs of both `Data` and `MAR` (do not reverse the sign of the output). This function computes sample lower partial moments from data. To compute expected lower partial moments for multivariate normal asset returns with a specified mean and covariance, use `elpm`. With `lpm`, you can compute various investment ratios such as Omega ratio, Sortino ratio, and Upside Potential ratio, where:

- `Omega = lpm(-Data, -MAR, 1) / lpm(Data, MAR, 1)`
- `Sortino = (mean(Data) - MAR) / sqrt(lpm(Data, MAR, 2))`
- `Upside = lpm(-Data, -MAR, 1) / sqrt(lpm(Data, MAR, 2))`

Examples

See “Sample Lower Partial Moments” on page 7-15.

References

Vijay S. Bawa. "Safety-First, Stochastic Dominance, and Optimal Portfolio Choice." *Journal of Financial and Quantitative Analysis.* Vol. 13, No. 2, June 1978, pp. 255–271.

W. V. Harlow. "Asset Allocation in a Downside-Risk Framework." *Financial Analysts Journal.* Vol. 47, No. 5, September/October 1991, pp. 28–40.

W. V. Harlow and K. S. Rao. "Asset Pricing in a Generalized Mean-Lower Partial Moment Framework: Theory and Evidence." *Journal of Financial and Quantitative Analysis.* Vol. 24, No. 3, September 1989, pp. 285–311.

Frank A. Sortino and Robert van der Meer. "Downside Risk." *Journal of Portfolio Management.* Vol. 17, No. 5, Spring 1991, pp. 27–31.

See Also

`elpm`

Topics

“Sample Lower Partial Moments” on page 7-15
“Performance Metrics Overview” on page 7-2

Introduced in R2006b

lweekdate

Date of last occurrence of weekday in month

Syntax

```
LastDate = lweekdate(Weekday,Year,Month)
LastDate = lweekdate( ___ ,NextDay,outputType)
```

Description

`LastDate = lweekdate(Weekday,Year,Month)` returns the date number for the last occurrence of `Weekday` in the given year and month.

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.

`LastDate = lweekdate( ___ ,NextDay,outputType)` returns the date of last occurrence of weekday in month using the optional arguments for `NextDay` and `outputType`.

The type of the output for `LastDate` depends on the input `outputType`. If this variable is `'datenum'`, `LastDate` is a serial date number. If `outputType` is `'datetime'`, then `LastDate` is a datetime array. By default, `outputType` is set to `'datenum'`.

Use the function `datestr` to convert serial date numbers to formatted date character vectors.

Examples

Determine the Date of Last Occurrence of Weekday in a Month

Determine the last Monday in June 2001.

```
LastDate = lweekdate(2, 2001, 6); datestr(LastDate)

ans =
'25-Jun-2001'
```

Determine the last Monday in a week that also contains a Friday in June 2001 returned as a datetime array.

```
LastDate = lweekdate(2, 2001, 6,[],'datetime')

LastDate = datetime
   25-Jun-2001
```

Determine 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'
```

Determine the last Monday in May for 2001, 2002, and 2003:

```
Year = [2001:2003];
LastDate = lweekdate(2, Year, 5);
datestr(LastDate)

ans = 3x11 char array
    '28-May-2001'
    '27-May-2002'
    '26-May-2003'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Weekday` — Weekday whose date you seek

integer with value 1 through 7 | vector of integers with values 1 through 7

Weekday whose date you seek, specified as an integer or a vector of integers from `1` through `7`.

- `1` — Sunday
- `2` — Monday
- `3` — Tuesday
- `4` — Wednesday
- `5` — Thursday
- `6` — Friday
- `7` — Saturday

Data Types: `single` | `double`

`Year` — Year to determine occurrence of weekday
4-digit integer | vector of 4-digit integers

Year to determine occurrence of weekday, specified as a 4-digit integer or vector of 4-digit integers.

Data Types: `single` | `double`

`Month` — Month to determine occurrence of weekday
integer with value `1` through `12` | vector of integers with values `1` through `12`

Month to determine occurrence of weekday, specified as an integer or vector of integers with values `1` through `12`.

Data Types: `single` | `double`

`NextDay` — Weekday that must occur after `Weekday` in same week
`0` = ignore (default) | integer with value `0` through `7` | vector of integers with values `0` through `7`

Weekday that must occur after `Weekday` in same week, specified as an integer or a vector of integers from `0` through `7`, where `0` = ignore (default) and `1` through `7` are the same as for `Weekday`.

Data Types: `single` | `double`

`outputType` — Year to determine days
`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

A character vector specified as either `'datenum'` or `'datetime'`. The output `LastDate` is in serial date format if `'datenum'` is specified, or datetime format if `'datetime'` is specified. By default the output `LastDate` is in serial date format.

Data Types: `char`

Output Arguments

`LastDate` — Date for last occurrence of `Weekday` in given year and month

serial date number | date character vector

Date for last occurrence of `Weekday` in given year and month, returned as a serial date number or date character vector.

The type of the output for `LastDate` depends on the optional input argument `outputType`. If this variable is `'datenum'`, `LastDate` is a serial date number. If `outputType` is `'datetime'`, then `LastDate` is a datetime array. By default, `outputType` is set to `'datenum'`.

See Also

`datetime` | `eomdate` | `lbusdate` | `nweekdate`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

m2xdate

MATLAB date to Excel serial date number

Syntax

```
DateNum = m2xdate(MATLABDateNumber,Convention)
```

Description

`DateNum = m2xdate(MATLABDateNumber,Convention)` converts MATLAB serial date numbers, date character vectors, or datetime arrays to Excel serial date numbers. MATLAB date numbers start with `1` = January 1, 0000 A.D., hence there is a difference of 693960 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with Spreadsheet Link™ software.

Examples

Convert MATLAB Serial Date Numbers Using 1900 Date System

This example shows how to convert MATLAB serial date numbers using the 1900 date system. Given MATLAB date numbers for Christmas 2001 through 2004, convert them to Excel date numbers in the 1900 system.

```
DateNum = datenum(2001:2004, 12, 25);
ExDate = m2xdate(DateNum)

ExDate =

       37250       37615       37980       38346
```

Convert MATLAB Serial Date Numbers Using 1900 Date System With a datetime Array

This example shows how to convert MATLAB® date numbers using a datetime array with the 1900 date system. Given MATLAB date numbers for Christmas 2001 through 2004, convert them to Excel date numbers in the 1904 system.

```
DateNum = datetime(2001:2004, 12, 25,'Locale','en_US');
ExDate = m2xdate(DateNum)

ExDate =

       37250       37615       37980       38346
```

Convert MATLAB Serial Date Numbers Using 1904 Date System

This example shows how to convert MATLAB serial date numbers using the 1904 date system. Given MATLAB date numbers for Christmas 2001 through 2004, convert them to Excel date numbers in the 1904 system.

```
DateNum = datenum(2001:2004, 12, 25);
ExDate = m2xdate(DateNum, 1)

ExDate =

       35788       36153       36518       36884
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`MATLABDateNumber` — MATLAB dates

serial date number | date character vector | datetime array

MATLAB dates, specified as a scalar or vector of MATLAB serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Convention` — Flag for Excel date system

`0` (Excel 1900 date system is in effect) (default) | numeric with value `0` or `1`

Flag for Excel date system, specified as a scalar or vector as a numeric with a value `0` or `1`.

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.

Due to a software limitation in Excel software, the year 1900 is considered a leap year. As a result, all DATEVALUE's reported by Excel software between Jan. 1, 1900 and Feb. 28, 1900 (inclusive) differs from the values reported by 1. For example:

- In Excel software, Jan. 1, 1900 = `1`
- In MATLAB, Jan. 1, 1900 – `693960` (for 1900 date system) = `2`

```
datenum('Jan 1, 1900') - 693960

ans =

     2
```

Data Types: `logical`

Output Arguments

`DateNum` — Excel serial date number

array of serial date numbers

Excel serial date number, returned as an array of serial date numbers in Excel serial date number form.

See Also

`datenum` | `datestr` | `datetime` | `x2mdate`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

macd

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, and 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 must indicate whether each row is a set of observations (`dim = 2`) or each column is a set of observations (`dim = 1`, the default).

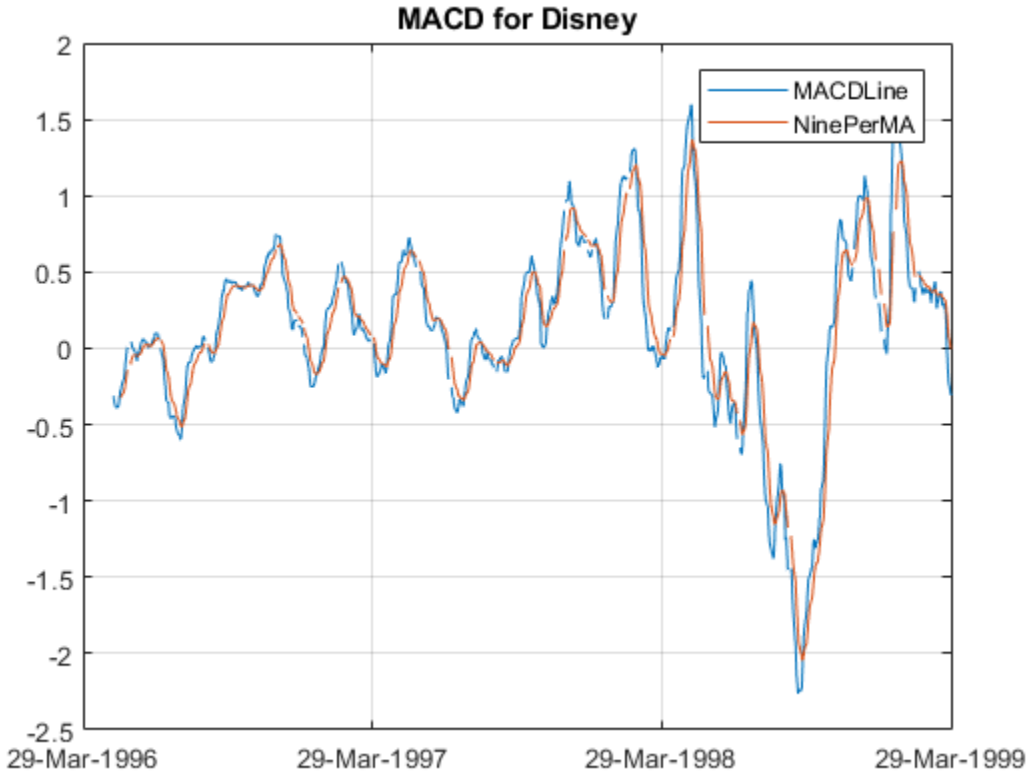
`macdts = macd(tsobj,series_name)` calculates the MACD line from the financial time series `tsobj`, and the nine-period exponential moving average from the MACD line. The MACD is calculated for the closing price series in `tsobj`, presumed to have been named `Close`. The result is stored in the financial 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 Moving Average Convergence/Divergence (MACD)

This example shows how to compute the MACD for Disney stock and plot the results.

```
load disney.mat
dis_CloseMACD = macd(dis);
dis_OpenMACD = macd(dis, 'OPEN');
plot(dis_CloseMACD);
plot(dis_OpenMACD);
title('MACD for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

[1] Achelis, Steven B. *Technical Analysis From A To Z.* Second Edition. McGraw-Hill, 1995, pp. 166–168.

See Also

`adline` | `willad`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

max

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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

maxdrawdown

Compute maximum drawdown for one or more price series

Syntax

```
MaxDD = maxdrawdown(Data)

MaxDD = maxdrawdown(Data,Format)

[MaxDD,MaxDDIndex] = maxdrawdown(Data,Format)
```

Arguments

`Data`	`T`-by-`N` matrix with `T` samples of `N` total return price series (also known as total equity).
`Format`	(Optional) MATLAB character vector indicating format of data. Possible values are:
	`'return'` (default) — Maximum drawdown in terms of maximum percentage drop from a peak.
	`'arithmetic'` — Maximum drawdown of an arithmetic Brownian motion with drift (differences of data from peak to trough) using the equation $dX(t) = \mu dt + \sigma dW(t)$.
	`'geometric'` — Maximum drawdown of a geometric Brownian motion with drift (differences of log of data from peak to trough) using the equation $dS(t) = \mu_0 S(t) dt + \sigma_0 S(t) dW(t)$

Description

`MaxDD = maxdrawdown(Data,Format)` computes maximum drawdown for each series in an `N`-vector `MaxDD` and identifies start and end indexes of maximum drawdown periods for each series in a `2`-by-`N` matrix `MaxDDIndex`.

To summarize the outputs of `maxdrawdown`:

- `MaxDD` is a `1`-by-`N` vector with maximum drawdown for each of `N` time series.
- `MaxDDIndex` is a `2`-by-`N` vector of start and end indexes for each maximum drawdown period for each total equity time series, where the first row contains the start indexes and the second row contains the end indexes of each maximum drawdown period.

Notes

- Drawdown is the percentage drop in total returns from the start to the end of a period. If the total equity time series is increasing over an entire period, drawdown is 0. Otherwise, it is a positive number. Maximum drawdown is an ex-ante proxy for downside risk that computes the largest drawdown over all intervals of time that can be formed within a specified interval of time.
- Maximum drawdown is sensitive to quantization error.

Examples

See “Maximum Drawdown” on page 7-18.

References

Christian S. Pederson and Ted Rudholm-Alfvin. "Selecting a Risk-Adjusted Shareholder Performance Measure." *Journal of Asset Management*. Vol. 4, No. 3, 2003, pp. 152–172.

See Also

`emaxdrawdown`

Topics

“Maximum Drawdown” on page 7-18
“Performance Metrics Overview” on page 7-2

Introduced in R2006b

mean

Arithmetic average

Syntax

```
tsmean = mean(tsobj)
```

Description

`tsmean = mean(tsobj)` computes the arithmetic mean of all data in all series in the financial time series object (`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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

medprice

Median price

Syntax

```
mprc = medprice(highp,lowp)

mprc = medprice([highp lowp])

mprcts = medprice(tsobj)

mprcts = medprice(tsobj,'ParameterName',ParameterValue, ...)
```

Arguments

`highp`	High price (vector)
`lowp`	Low price (vector)
`tsobj`	Financial time series object
`'ParameterName'`	Valid parameter names are: • `HighName` — high prices series name • `LowName` — low prices series name
`ParameterValue`	Parameter values are the character vectors that represent the valid parameter names.

Description

`mprc = medprice(highp,lowp)` calculates the median prices `mprc` from the high (`highp`) and low (`lowp`) prices. The median price is the average of the high and low price for each period.

`mprc = medprice([highp lowp])` accepts a two-column matrix as the input rather than two individual vectors. The columns of the matrix represent the high and low prices, in that order.

`mprcts = medprice(tsobj)` calculates the median prices of a financial time series object `tsobj`. The object must minimally contain the series `High` and `Low`. The median price is the average of the high and low price each period. `mprcts` is a financial time series object with the same dates as `tsobj` and the data series `MedPrice`.

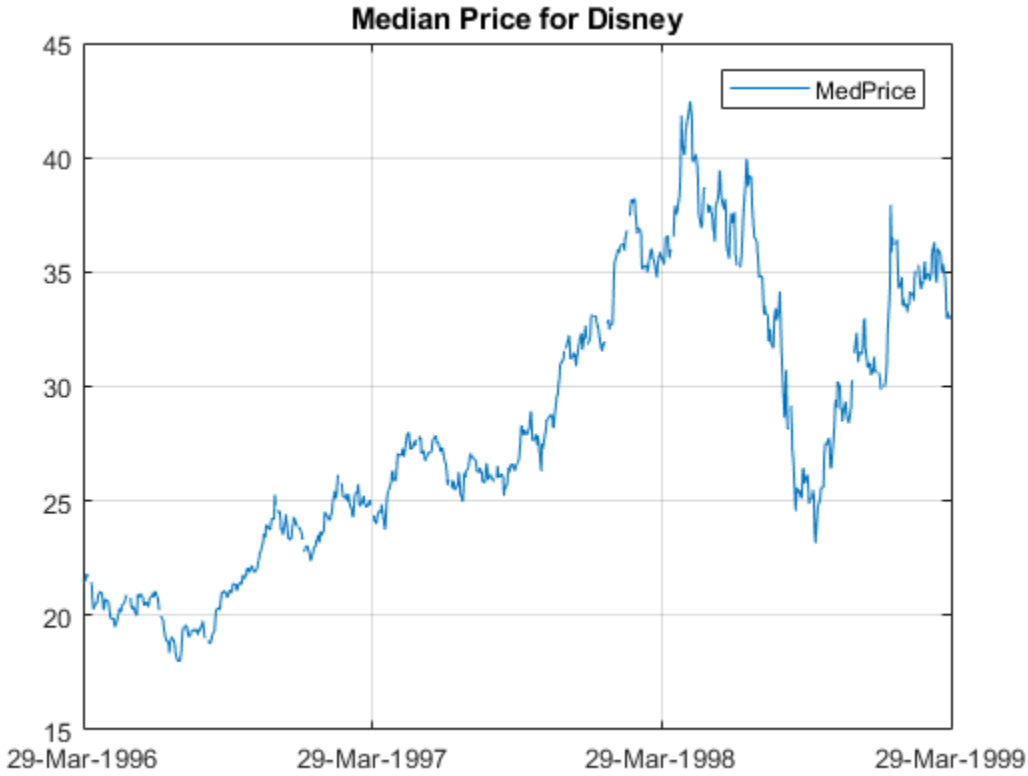
`mprcts = medprice(tsobj,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Median Price

This example shows how to compute the median price for Disney stock and plot the results.

```
load disney.mat
dis_MedPrice = medprice(dis);
plot(dis_MedPrice)
title('Median Price for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 177–178.

See Also

`typprice` | `wclose`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

merge

Merge multiple financial time series objects

Syntax

```
newfts = merge(fts1,fts2)

newfts = merge(fts1,fts2, ..., ftsx)

newfts = merge(fts1,fts2, ..., ftsx,'PARAM1',VALUE1,'PARAM2',VALUE2, ...)
```

Arguments

`fts1, fts2, ...`	Comma-separated list of financial time series objects to merge. **Note** Multiple Financial Time Series objects can be merged at once. The merged objects must appear in a comma separated list before the optional inputs. The order of the inputs is significant.
`'DateSetMethod'`	(Optional) Merge *`method`*. Valid merge values are: `'union'` — (Default) Returns the combined values of all merged objects. `'intersection'` — Returns the values common to all merged objects. `RefObj` — Maps all values to a reference time contained in a Financial Time Series object (`RefObj`) or vector of date numbers.

`'DataSetMethod'`	(Optional) Merge *`method`*. Valid merge values are: `'closest'` — (Default) Returns data based on the order of the inputs. However, the first missing data point (`NaN` value) of a date will be replaced by the closest non-`NaN` data point that appears on the same date of subsequent merged objects. `'order'` — Returns data based strictly on the order of the inputs.
`'SortColumns'`	(Optional) Sorts columns. Valid merge values are: `True/1` — (Default) Sorts the columns based on the headers (series names). The headers are sorted in alphabetical order. `False/0` — Columns are not sorted.

Description

`newfts = merge(fts1,fts2, ...,
ftsx,'PARAM1',VALUE1,'PARAM2',VALUE2', ...)` merges multiple financial time series objects. The optional parameter and value pair argument specifies the values contained in the output financial time series object `ftsout`.

Examples

Create Three Financial Time Series Objects and Merge into a Single Object

Create three financial time series objects and merged into a new time series object `t123`.

```
dates = {'jan-01-2001'; 'jan-02-2001'; 'jan-03-2001'; ...
         'jan-04-2001'; 'jan-06-2001'};
data = [1; 1; 1; 1; 1];
t1 = fints(dates, data);

dates = {'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001';
         'jan-05-2001'};
data = [2; 2; 2; 2];
```

```
t2 = fints(dates, data);

dates = {'jan-03-2001'; 'jan-04-2001'; 'jan-05-2001';
         'jan-06-2001'};
data = [3; 3; 3; 3];
t3 = fints(dates, data);

t123 = merge(t1, t2, t3)


t123 =

    desc:   ||  ||
    freq:  Unknown (0)

    'dates:  (6)'    'series1:  (6)'
    '01-Jan-2001'    [            1]
    '02-Jan-2001'    [            1]
    '03-Jan-2001'    [            1]
    '04-Jan-2001'    [            1]
    '05-Jan-2001'    [            2]
    '06-Jan-2001'    [            1]
```

If you change the order of input time series, the output may contain different data when duplicate dates exist. Here, for example, is the result of using the same three time series defined above but with the order changed.

```
merge(t3, t2, t1)


ans =

    desc:   ||  ||
    freq:  Unknown (0)

    'dates:  (6)'    'series1:  (6)'
    '01-Jan-2001'    [            1]
    '02-Jan-2001'    [            2]
    '03-Jan-2001'    [            3]
    '04-Jan-2001'    [            3]
    '05-Jan-2001'    [            3]
    '06-Jan-2001'    [            3]
```

`t123` contains all 1's except on `'05-Jan-2001'` because `t1` appears first in the list of inputs and takes precedence. The same logic can be applied to `t321`. By changing the

order of inputs, you can overwrite old financial time series data with new data by placing the new time series ahead of the old one in the list of inputs to the `merge` function.

Merge Financial Time Series Objects with Different Headers (Series Names)

Merge time series objects with different headers into a new time series object `t45`.

```
dates = {'jan-01-2001'; 'jan-02-2001'; 'jan-03-2001'; ...
'jan-04-2001'; 'jan-06-2001'};
data = [1; 1; 1; 1; 1];
t4 = fints(dates, data, 'ts4');

dates = {'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001'; 'jan-05-2001'};
data = [2; 2; 2; 2];
t5 = fints(dates, data, 'ts5');

t45 = merge(t4, t5)
```

```
t45 =

    desc:   ||
    freq:  Unknown (0)

    'dates:  (6)'    'ts4:  (6)'    'ts5:  (6)'
    '01-Jan-2001'    [        1]    [      NaN]
    '02-Jan-2001'    [        1]    [        2]
    '03-Jan-2001'    [        1]    [        2]
    '04-Jan-2001'    [        1]    [        2]
    '05-Jan-2001'    [      NaN]    [        2]
    '06-Jan-2001'    [        1]    [      NaN]
```

Merge Two Financial Index Series and Keep Intersecting Dates

Merge two index series into the final merged object (`t12`) and keep the intersecting dates.

```
dates = {'jan-01-2001'; 'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001'; 'jan-06-2001'};
data = [1; 1; 1; 1; 1];
t1 = fints(dates, data,'A')
```

```
t1 =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (5)'    'A:  (5)'
    '01-Jan-2001'    [      1]
    '02-Jan-2001'    [      1]
    '03-Jan-2001'    [      1]
    '04-Jan-2001'    [      1]
    '06-Jan-2001'    [      1]
```

```
dates = {'jan-02-2001'; 'jan-03-2001'; 'jan-04-2001'; 'jan-05-2001'};
data = [2; 2; 2; 2];
t2 = fints(dates, data,'B')
```

```
t2 =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (4)'    'B:  (4)'
    '02-Jan-2001'    [      2]
    '03-Jan-2001'    [      2]
    '04-Jan-2001'    [      2]
    '05-Jan-2001'    [      2]
```

```
t12 = merge(t1, t2,'DateSetMethod','Intersection')
```

```
t12 =

    desc:   ||
    freq:  Unknown (0)

    'dates:  (3)'    'A:  (3)'    'B:  (3)'
    '02-Jan-2001'    [      1]    [      2]
```

```
'03-Jan-2001'    [      1]    [      2]
'04-Jan-2001'    [      1]    [      2]
```

- “Merge Financial Time Series Objects” on page 13-12
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`horzcat` | `vertcat`

Topics

“Merge Financial Time Series Objects” on page 13-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

min

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`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

minus

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

`plus` | `rdivide` | `times`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

minute

Minute of date or time

Syntax

```
Minute = minute(Date)
Minute = minute(___,F)
```

Description

`Minute = minute(Date)` returns the minute of date or time given a serial date number or a date character vector.

`Minute = minute(___,F)` returns the minute of date or time given a serial date number or a date character vector, `Date`, using format defined by the optional input `F`. `Date` can be a character array where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`.

Examples

Determine the Minutes of the Date for Various Dates

Find the minutes of the day for `Date` using a serial date number.

```
Minute = minute(731204.5591223380)
```

```
Minute = 25
```

Find the minutes of the day for `Date` using a date character vector format.

```
Minute = minute('19-dec-2001, 13:25:08.17')
```

```
Minute = 25
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Date` — Date to determine minute

serial date number | date character vector | cell array of date character vectors

Date to determine minute, specified as a serial date number or date character vector.

`Date` can be an array of date character vectors, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`

Data Types: `single` | `double` | `char` | `cell`

`F` — Date format symbol

character vector designating date format

Date format symbol, specified as a character vector to designate the date format symbol for input argument `Date`. For more information on supported date character vector formats, see `datestr`. Note, formats with `'Q'` are not accepted.

Data Types: `char`

Output Arguments

`Minute` — Minute of date or time

serial date number | datetime array

Minute of date or time, returned as a serial date number or date character vector.

See Also

`datevec` | `hour` | `second`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

mirr

Modified internal rate of return

Syntax

```
Return = mirr(CashFlow,FinRate,Reinvest)
```

Arguments

`CashFlow`	Vector of cash flows. The first entry is the initial investment.
`FinRate`	Finance rate for negative cash flow values. Enter as a decimal fraction.
`Reinvest`	Reinvestment rate for positive cash flow values, as a decimal fraction.

Description

`Return = mirr(CashFlow,FinRate,Reinvest)` calculates the modified internal rate of return for a series of periodic cash flows. This function calculates only positive rates of return; for nonpositive rates of return, `Return = 0`.

Examples

This cash flow represents the yearly income from an initial investment of $100,000. The finance rate is 9% and the reinvestment rate is 12%.

Year 1	$20,000
Year 2	($10,000)
Year 3	$30,000
Year 4	$38,000

Year 5	$50,000

To calculate the modified internal rate of return on the investment

```
Return = mirr([-100000 20000 -10000 30000 38000 50000], 0.09,...
0.12)
```

returns

```
Return =
          0.0832 (8.32%)
```

References

Brealey and Myers. *Principles of Corporate Finance.* Chapter 5.

See Also

`annurate` | `effrr` | `irr` | `nomrr` | `pvvar` | `xirr`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

month

Month of date

Syntax

```
[MonthNum,MonthString] = month(Date)
[MonthNum,MonthString] = month(___,F)
```

Description

`[MonthNum,MonthString] = month(Date)` returns the month of date given a serial date number or a date character vector.

`[MonthNum,MonthString] = month(___,F)` returns the month of date given a serial date number or a date character vector, `Date`, using format defined by the optional input `F`. `Date` can be a character array where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`.

Examples

Determine the Month for Various Dates

Find the month for `Date` using a serial date number.

```
[MonthNum, MonthString] = month(730368)
```

```
MonthNum = 9
```

```
MonthString =
'Sep'
```

Find the month for `Date` using a date character vector format.

```
[MonthNum, MonthString] = month('05-Sep-1999')

MonthNum = 9

MonthString =
'Sep'
```

Use the optional `F` argument to designate a country-specific date format for a given `Date`.

```
[MonthNum, MonthString] = month('1999/05/09','yyyy/dd/mm')

MonthNum = 9

MonthString =
'Sep'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Date` — Date to determine month

serial date number | date character vector | cell array of date character vectors

Date to determine month, specified as a serial date number or date character vector.

`Date` can be an array of date character vectors, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`

Data Types: `single` | `double` | `char` | `cell`

`F` — Date format symbol

character vector designating date format

Date format symbol, specified as a character vector to designate the date format symbol for input argument `Date`. For more information on supported date character vector formats, see `datestr`. Note, formats with `'Q'` are not accepted.

Data Types: `char`

Output Arguments

`MonthNum` — Numeric representation of the month
nonnegative integer

Month of date, returned as a nonnegative integer.

`MonthString` — Three letter abbreviation for month
character vector

Three letter abbreviation for month, returned as a character vector.

See Also

`datevec` | `day` | `year`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

months

Number of whole months between dates

Syntax

```
MyMonths = months(StartDate,EndDate)
MyMonths = months( ___ ,EndMonthFlag)
```

Description

`MyMonths = months(StartDate,EndDate)` returns the number of whole months between `StartDate` and `EndDate`. If `EndDate` is earlier than `StartDate`, `MyMonths` is negative.

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 character vector dates, then `EndDate` must be an *n*-row character array of character vector dates or a single date. `MyMonths` is then an `N`-by-`1` vector of numbers.

`MyMonths = months( ___ ,EndMonthFlag)` returns the number of whole months between `StartDate` and `EndDate` using an optional argument for `EndMonthFlag`. If `EndDate` is earlier than `StartDate`, `MyMonths` is negative.

Examples

Determine the Number of Whole Months Between Dates

Find the number of whole months using date character vectors.

```
MyMonths = months('may 31 2000', 'jun 30 2000', 1)
```

```
MyMonths = 1
```

Find the number of whole months using date character vectors when the optional `EndMonthFlag` = `0`.

```
MyMonths = months('may 31 2000','jun 30 2000', 0)
```

```
MyMonths = 0
```

Find the number of whole months using a cell array of date character vectors.

```
Dates = ['mar 31 2002'; 'apr 30 2002'; 'may 31 2002'];
MyMonths = months(Dates, 'jun 30 2002')
```

```
MyMonths =

     3
     2
     1
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Starting date for number of whole months between dates

serial date number | date character vector | cell array of date character vectors

Starting date for number of whole months between dates, specified as a serial date number or date character vector.

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 character vector dates, then `EndDate` must be an *n*-row character array of character vector dates or a single date. `MyMonths` is then an *n*-by-`1` vector of numbers.

Data Types: `single` | `double` | `char` | `cell`

`EndDate` — Ending date for number of whole months between dates

serial date number | date character vector | cell array of date character vectors

Ending date for number of whole months between dates, specified as a serial date number or date character vector.

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 character vector dates, then `EndDate` must be an *n*-row character array of character vector dates or a single date. `MyMonths` is then an *n*-by-`1` vector of numbers.

Data Types: `single` | `double` | `char` | `cell`

`EndMonthFlag` — Flag for end-of-month rule
`1` (default) | nonnegative integer with values `0` or `1`

Flag for end-of-month rule, specified as a nonnegative integer with values `0` or `1`.

If `StartDate` and `EndDate` are end-of-month dates and `EndDate` has fewer days than `StartDate`, `EndMonthFlag = 1`. In this case, `EndDate` is treated as the end of a whole month, while `EndMonthFlag = 0` does not.

Data Types: `logical`

Output Arguments

`MyMonths` — Number of whole months between dates
nonnegative integer

Number of whole months between dates, returned as a nonnegative integer.

See Also

`yearfrac`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

movavg

Leading and lagging moving averages chart

Syntax

```
movavg(Asset,Lead,Lag,Alpha)

[Short,Long] = movavg(Asset,Lead,Lag,Alpha)
```

Arguments

`Asset`	Security data, a vector of time-series prices.
`Lead`	Number of samples to use in leading average calculation. A positive integer. `Lead` must be less than or equal to `Lag`.
`Lag`	Number of samples to use in the lagging average calculation. A positive integer.
`Alpha`	(Optional) Control parameter that determines the type of moving averages. `0` = simple moving average (default), `0.5` = square root weighted moving average, `1` = linear moving average, `2` = square weighted moving average, and so on. To calculate the exponential moving average, set `Alpha ='e'`. **Note** When `Alpha = 'e'`, the value of the moving average depends on all previous data points (due to the iterative calculation). In this case, the `Lead` and `Lag` parameters are used to calculate the weighting factor for their respective averages (which is different from the number of samples).

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.

Note When using `movavg` syntax with output arguments, zero padding is used at the edges of the data. If you use `movavg` without output arguments, there is no zero padding in the data for the plot.

Examples

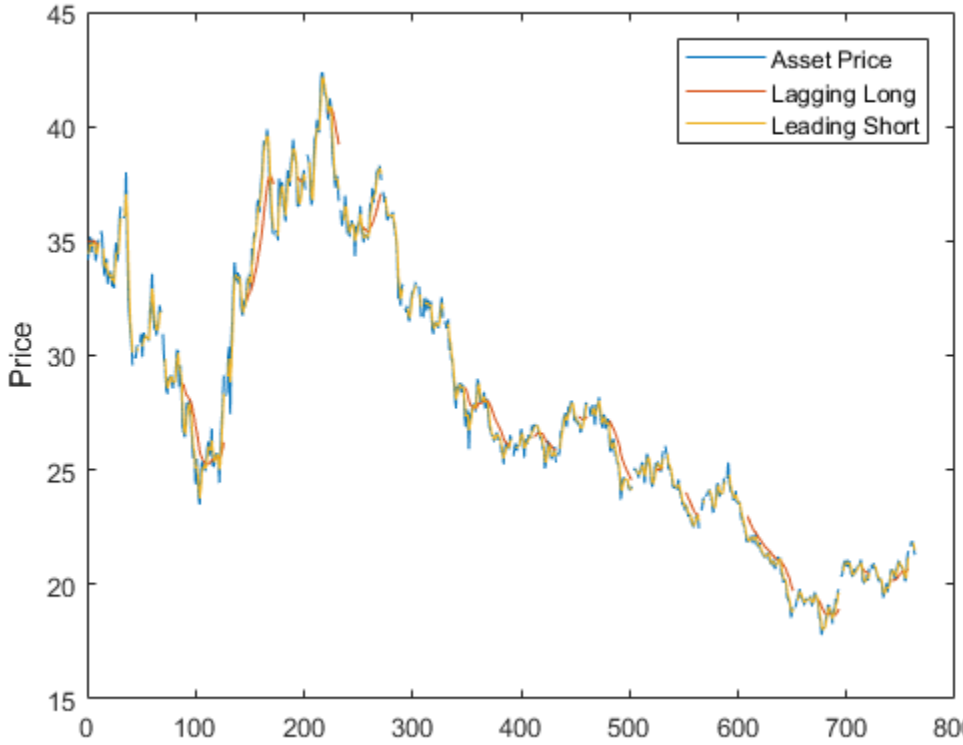
Compute the Moving Average for DIS Closing Prices

Load the DIS closing prices using `disney.mat`.

```
load disney.mat
```

Use `movavg` to plot the leading and lagging moving averages for DIS.

```
movavg(dis_CLOSE,3,20,1);ylabel('Price')
legend('Asset Price','Lagging Long','Leading Short')
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `dateaxis` | `highlow` | `pointfig`

Topics

“Charting Financial Data” on page 2-14

Introduced before R2006a

mrdivide

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 (`tsobj`) 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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

mtimes

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 (`tsobj`) 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` and `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

`minus` | `mrdivide` | `plus` | `times`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

mvnrfish

Fisher information matrix for multivariate normal or least-squares regression

Syntax

```
Fisher = mvnrfish(Data,Design,Covariance,MatrixFormat,CovarFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `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) Character vector 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.
`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. The covariance matrix is a full matrix. • `'diagonal'` — The covariance matrix is a diagonal matrix.

Description

`Fisher = mvnrfish(Data,Design,Covariance,MatrixFormat,CovarFormat)` computes a Fisher information matrix based on current maximum likelihood or least-squares parameter estimates.

`Fisher` is a `TOTALPARAMS`-by-`TOTALPARAMS` Fisher information matrix. The size of `TOTALPARAMS` depends on `MatrixFormat` and on current parameter estimates. If `MatrixFormat = 'full'`,

```
TOTALPARAMS = NUMPARAMS + NUMSERIES * (NUMSERIES + 1)/2
```

If `MatrixFormat = 'paramonly'`,

```
TOTALPARAMS = NUMPARAMS
```

Note `mvnrfish` operates slowly if you calculate the full Fisher information matrix.

Examples

See “Multivariate Normal Linear Regression” on page 9-2.

See Also

`mvnrmle` | `mvnrstd`

Topics

“Multivariate Normal Regression With Missing Data” on page 9-17
“Multivariate Normal Regression Without Missing Data” on page 9-17
“Least-Squares Regression With Missing Data” on page 9-18
“Least-Squares Regression Without Missing Data” on page 9-18
“Fisher Information” on page 9-6
“Multivariate Normal Linear Regression” on page 9-2
“Least-Squares Regression” on page 9-5

Introduced in R2006a

mvnrmle

Multivariate normal regression (ignore missing data)

Syntax

```
[Parameters,Covariance,Resid,Info] = mvnrmle(Data,Design,MaxIterations,TolParam,TolObj,
```

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`	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
	$\|Param_k - Param_{k-1}\| < TolParam \times (1 + \|Param_k\|)$
	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 $\|Obj_k - Obj_{k-1}\| < TolObj \times (1 + \|Obj_k\|)$ 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.
`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. Compute the full covariance matrix. • `'diagonal'` — Force the covariance matrix to be a diagonal matrix.

Description

`[Parameters,Covariance,Resid,Info] = mvnrmle(Data,Design,MaxIterations,TolParam,TolObj,Covar0,CovarFormat)` estimates a multivariate normal regression model without missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

for samples $k = 1, \ldots,$ `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 contain each value of the objective function at each iteration of the estimation algorithm. The last value in this vector, `Obj(end)`, is the terminal estimate of the objective function. If you do maximum likelihood estimation, the objective function is the log-likelihood function.
- `Info.PrevParameters` – `NUMPARAMS`-by-`1` column vector of estimates for the model parameters from the iteration just before the terminal iteration.

- `Info.PrevCovariance` — `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance parameters from the iteration just before the terminal iteration.

Notes

`mvnrmle` does not accept an initial parameter vector, because the parameters are estimated directly from the first iteration onward.

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES` = `1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES` > `1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

These points concern how `Design` handles missing data:

- Although `Design` should not have `NaN` values, ignored samples due to `NaN` values in `Data` are also ignored in the corresponding `Design` array.
- If `Design` is a `1`-by-`1` cell array, which has a single `Design` matrix for each sample, no `NaN` values are permitted in the array. A model with this structure must have `NUMSERIES` ≥ `NUMPARAMS` with `rank(Design{1}) = NUMPARAMS`.
- Two functions for handling missing data, `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.

Examples

See “Multivariate Normal Regression” on page 9-17, “Least-Squares Regression” on page 9-17, “Covariance-Weighted Least Squares” on page 9-18, “Feasible Generalized Least Squares” on page 9-19, and “Seemingly Unrelated Regression” on page 9-20.

References

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data.*, 2nd Edition. John Wiley & Sons, Inc., 2002.

Xiao-Li Meng and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika.* Vol. 80, No. 2, 1993, pp. 267–278.

See Also

`ecmmvnrmle` | `mvnrobj` | `mvnrstd` | `mvregress`

Topics

"Multivariate Normal Regression With Missing Data" on page 9-17
"Multivariate Normal Regression Without Missing Data" on page 9-17
"Capital Asset Pricing Model with Missing Data"
"Multivariate Normal Linear Regression" on page 9-2

Introduced in R2006a

mvnrobj

Log-likelihood function for multivariate normal regression without missing data

Syntax

```
Objective = mvnrobj(Data,Design,Parameters,Covariance,CovarFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `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.

`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. The covariance matrix is a full matrix. • `'diagonal'` — The covariance matrix is a diagonal matrix.

Description

`Objective = mvnrobj(Data,Design,Parameters,Covariance,CovarFormat)` computes the log-likelihood function based on current maximum likelihood parameter estimates without missing data. `Objective` is a scalar that contains the log-likelihood function.

Notes

You can configure `Design` as a matrix if `NUMSERIES = 1` or as a cell array if `NUMSERIES ≥ 1`.

- If `Design` is a cell array and `NUMSERIES` = `1`, each cell contains a `NUMPARAMS` row vector.
- If `Design` is a cell array and `NUMSERIES` > `1`, each cell contains a `NUMSERIES`-by-`NUMPARAMS` matrix.

Although `Design` should not have `NaN` values, ignored samples due to `NaN` values in `Data` are also ignored in the corresponding `Design` array.

Examples

See “Multivariate Normal Regression” on page 9-17, “Least-Squares Regression” on page 9-17, “Covariance-Weighted Least Squares” on page 9-18, “Feasible Generalized Least Squares” on page 9-19, and “Seemingly Unrelated Regression” on page 9-20.

See Also

`ecmmvnrmle` | `ecmmvnrobj` | `mvnrmle`

Topics

"Multivariate Normal Regression Without Missing Data" on page 9-17
"Multivariate Normal Linear Regression" on page 9-2

Introduced in R2006a

mvnrstd

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters,StdCovariance] = mvnrstd(Data,Design,Covariance,CovarFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `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.

`CovarFormat`	(Optional) Character vector that specifies the format for the covariance matrix. The choices are: • `'full'` — Default method. The covariance matrix is a full matrix. • `'diagonal'` — The covariance matrix is a diagonal matrix.

Description

`[StdParameters,StdCovariance] = mvnrstd(Data,Design,Covariance,CovarFormat)` evaluates standard errors for a multivariate normal regression model without missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

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.

Examples

See “Multivariate Normal Regression” on page 9-17, “Least-Squares Regression” on page 9-17, “Covariance-Weighted Least Squares” on page 9-18, “Feasible Generalized Least Squares” on page 9-19, and “Seemingly Unrelated Regression” on page 9-20.

References

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data.* 2nd Edition. John Wiley & Sons, Inc., 2002.

See Also

`ecmmvnrmle` | `ecmmvnrstd` | `mvnrmle`

Topics

“Multivariate Normal Regression Without Missing Data” on page 9-17
“Multivariate Normal Regression Without Missing Data” on page 9-17
“Multivariate Normal Linear Regression” on page 9-2

Introduced in R2006a

nancov

Covariance ignoring NaNs

Syntax

```
c = nancov(X)
```

```
c = nancov(...,'pairwise')
```

Arguments

X	Financial times series object.
Y	Financial times series object.

Description

nancov for financial times series objects is based on the Statistics and Machine Learning Toolbox function nancov. See nancov.

c = nancov(X), if X is a financial time series object with one series and returns the sample variance of the values in X, treating NaNs as missing values. For a financial time series object containing more than one series, where each row is an observation and each series a variable, nancov(X) is the covariance matrix computing using rows of X that do not contain any NaN values. nancov(X,Y), where X and Y are financial time series objects with the same number of elements, is equivalent to nancov([X(:) Y(:)]).

nancov(X) or nancov(X,Y) normalizes by (N-1) if N >1, where N is the number of observations after removing missing values. This makes nancov the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For N = 1, cov normalizes by N.

nancov(X,1) or nancov(X,Y,1) normalizes by N and produces the second moment matrix of the observations about their mean. nancov(X,Y,0) is the same as nancov(X,Y), and nancov(X,0) is the same as nancov(X).

`c = nancov(...,'pairwise')` computes `c(i,j)` using rows with no `NaN` values in columns *i*or *j*. The result may not be a positive definite matrix. `c = nancov(..., 'complete')` is the default, and it omits rows with any `NaN` values, even if they are not in column *i* or *j*. The mean is removed from each column before calculating the result.

Examples

To generate random data having nonzero covariance between column `4` and the other columns:

```
x = randn(30, 4);                % uncorrelated data
x(:, 4) = sum(x, 2);             % introduce correlation
x(2, 3) = NaN;                   % introduce one missing value
f = fints((today:today+29)', x);  % create a fints object using x
c = nancov(f)                    % compute sample covariance

c =

    1.6898   -0.0005    0.3612    1.9143
   -0.0005    1.0833   -0.5513    0.6059
    0.3612   -0.5513    1.0369    0.7570
    1.9143    0.6059    0.7570    4.4895
```

See Also

`cov` | `nanvar` | `var`

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

nanmax

Maximum ignoring NaNs

Syntax

```
m = nanmax(X)

[m,ndx] = nanmax(X)

m = nanmax(X,Y)

[m,ndx] = nanmax(X,[],DIM)
```

Arguments

X	Financial times series object.
Y	Financial times series object or scalar.
DIM	Dimension of X.

Description

nanmax for financial times series objects is based on the Statistics and Machine Learning Toolbox function nanmax. See nanmax.

m = nanmax(X) returns the maximum of a financial time series object X with NaNs treated as missing. m is the largest non-NaN element in X.

[m,ndx] = nanmax(X) returns the indices of the maximum values in X. If the values along the first nonsingleton dimension contain multiple maximal elements, the index of the first one is returned.

m = nanmax(X,Y) returns an array the same size as X and Y with the largest elements taken from X or Y. Only Y can be a scalar double.

`[m,ndx] = nanmax(X,[],DIM)` operates along the dimension `DIM`.

Examples

To compute `nanmax` for the following dates:

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
f.series3(2) = nan;

[nmax, maxidx] = nanmax(f)

nmax =
     4     5     6

maxidx =
     3     2     1
```

See Also

`max` | `nanmean` | `nanmedian` | `nanmin` | `nanstd` | `nanvar`

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

nanmean

Mean ignoring NaNs

Syntax

```
m = nanmean(X)

m = nanmean(X,DIM)
```

Arguments

X	Financial times series object.
DIM	Dimension along which the operation is conducted.

Description

nanmean for financial times series objects is based on the Statistics and Machine Learning Toolbox function nanmean. See nanmean.

m = nanmean(X) returns the sample mean of a financial time series object X, treating NaNs as missing values. m is a row vector containing the mean value of the non-NaN elements in each series.

m = nanmean(X,DIM) takes the mean along dimension DIM of X.

Examples

To compute nanmean for the following dates:

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
```

```
f.series3(2) = nan;

nmean = nanmean(f)

nmean =

    3.5000    3.0000    4.0000
```

See Also

mean | nanmax | nanmin | nanstd | nansum | nanvar

Topics

"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

nanmedian

Median ignoring NaNs

Syntax

```
m = nanmedian(X)

m = nanmedian(X,DIM)
```

Arguments

X	Financial times series object.
DIM	Dimension along which the operation is condcuted.

Description

nanmedian for financial times series objects is based on the Statistics and Machine Learning Toolbox function nanmedian. See nanmedian.

m = nanmedian(X) returns the sample median of a financial time series object X, treating NaNs as missing values. m is a row vector containing the median value of non-NaN elements in each column.

m = nanmedian(X,DIM) takes the median along the dimension DIM of X.

Examples

To compute nanmedian for the following dates:

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007';'04-Jan-2007'};
f = fints(dates, magic(4));
f.series1(1) = nan;
f.series2(2) = nan;
f.series3([1 3]) = nan;
```

```
nmedian = nanmedian(f)

nmedian =
    5.0000    7.0000   12.5000   10.0000
```

See Also

mean | nanmax | nanmin | nanstd | nansum | nanvar

Topics

"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

nanmin

Minimum ignoring NaNs

Syntax

```
m = nanmin(X)

[m,ndx] = nanmin(X)

m = nanmin(X,Y)

[m,ndx] = nanmin(X,[],DIM)
```

Arguments

X	Financial times series object.
Y	Financial times series object or scalar.
DIM	Dimension along which the operation is conducted.

Description

nanmin for financial times series objects is based on the Statistics and Machine Learning Toolbox function nanmin. See nanmin.

m = nanmin(X) returns the minimum of a financial time series object X with NaNs treated as missing. m is the smallest non-NaN element in X.

[m,ndx] = nanmin(X) returns the indices of the minimum values in X. If the values along the first nonsingleton dimension contain multiple elements, the index of the first one is returned.

m = nanmin(X,Y) returns an array the same size as X and Y with the smallest elements taken from X or Y. Only Y can be a scalar double.

`[m,ndx] = nanmin(X, [], DIM)` operates along the dimension `DIM`.

Examples

To compute `nanmin` for the following dates:

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
f.series3(2) = nan;

[nmin, minidx] = nanmin(f)

nmin =
     3     1     2
minidx =
     2     1     3
```

See Also

`mean` | `nanmax` | `nanstd` | `nanvar`

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

nanstd

Standard deviation ignoring NaNs

Syntax

```
y = nanstd(X)

y = nanstd(X,1)

y = nanstd(X,FLAG,DIM)
```

Arguments

X	Financial times series object.
FLAG	Normalization flag.
DIM	Dimension along which the operation is conducted.

Description

nanstd for financial times series objects is based on the Statistics and Machine Learning Toolbox function nanstd. See nanstd.

y = nanstd(X) returns the sample standard deviation of the values in a financial time series object X, treating NaNs as missing values. y is the standard deviation of the non-NaN elements of X.

nanstd normalizes y by (N – 1), where N is the sample size. This is the square root of an unbiased estimator of the variance of the population from which X is drawn, as long as X consists of independent, identically distributed samples and data are missing at random.

y = nanstd(X,1) normalizes by N and produces the square root of the second moment of the sample about its mean. nanstd(X,0) is the same as nanstd(X).

`y = nanstd(X,flag,dim)` takes the standard deviation along the dimension dim of `X`. Set the value of `flag` to `0` to normalize the result by `n – 1`; set the value of `flag` to `1` to normalize the result by `n`.

Examples

To compute `nanstd` for the following dates:

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
f.series3(2) = nan;

nstd = nanstd(f)

nstd =

          0.71          2.83          2.83
```

See Also

`nanmax` | `nanmean` | `nanmedian` | `nanmin` | `nanvar` | `std`

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

nansum

Sum ignoring NaNs

Syntax

```
y = nansum(X)

y = nansum(X,DIM)
```

Arguments

X	Financial time series object.
DIM	Dimension along which the operation is conducted.

Description

nansum for financial times series objects is based on the Statistics and Machine Learning Toolbox function nansum. See nansum.

y = nansum(X) returns the sum of a financial time series object X, treating NaNs as missing values. y is the sum of the non-NaN elements in X.

y = nansum(X,DIM) takes the sum along dimension DIM of X.

Examples

To compute nansum for the following dates:

```
dates = {'01-Jan-2007';'02-Jan-2007';'03-Jan-2007'};
f = fints(dates, magic(3));
f.series1(1) = nan;
f.series2(3) = nan;
f.series3(2) = nan;
```

```
nsum = nansum(f)

nsum =
     7     6     8
```

See Also

nanmax | nanmean | nanmedian | nanmin | nanstd | nanvar

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

nanvar

Variance ignoring NaNs

Syntax

```
y = nanvar(X)

y = nanvar(X,1)

y = nanvar(X,W)

y = nanvar(X,W,DIM)
```

Arguments

X	Financial times series object.
W	Weight vector.
DIM	Dimension along which the operation is conducted.

Description

nanvar for financial times series objects is based on the Statistics and Machine Learning Toolbox function nanvar. See nanvar.

y = nanvar(X) returns the sample variance of the values in a financial time series object X, treating NaNs as missing values. y is the variance of the non-NaN elements of each series in X.

nanvar normalizes y by N – 1 if N > 1, where N is the sample size of the non-NaN elements. This is an unbiased estimator of the variance of the population from which X is drawn, as long as X consists of independent, identically distributed samples, and data are missing at random. For N = 1, y is normalized by N.

`y = nanvar(X,1)` normalizes by `N` and produces the second moment of the sample about its mean. `nanvar(X, 0)` is the same as `nanvar(X)`.

`y = nanvar(X,W)` computes the variance using the weight vector `W`. The length of `W` must equal the length of the dimension over which `nanvar` operates, and its non-`NaN` elements must be nonnegative. Elements of `X` corresponding to `NaN` elements of `W` are ignored.

`y = nanvar(X,W,DIM)` takes the variance along dimension `DIM` of `X`.

Examples

To compute `nanvar`:

```
f = fints((today:today+1)', [4 -2 1; 9  5 7])
f.series1(1) = nan;
f.series3(2) = nan;

nvar = nanvar(f)

nvar =
         0   24.5000         0
```

See Also

`nanmax` | `nanmean` | `nanmedian` | `nanmin` | `nanstd` | `var`

Topics

"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

negvolidx

Negative volume index

Syntax

```
nvi = negvolidx(closep,tvolume,initnvi)

nvi = negvolidx([closep tvolume],initnvi)

nvits = negvolidx(tsobj)

vits = 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.
`'ParameterName'`	Valid parameter names are: • `CloseName`: closing prices series name • `VolumeName`: volume traded series name
`ParameterValue`	Parameter values are the character vectors that represent the valid parameter names.

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.

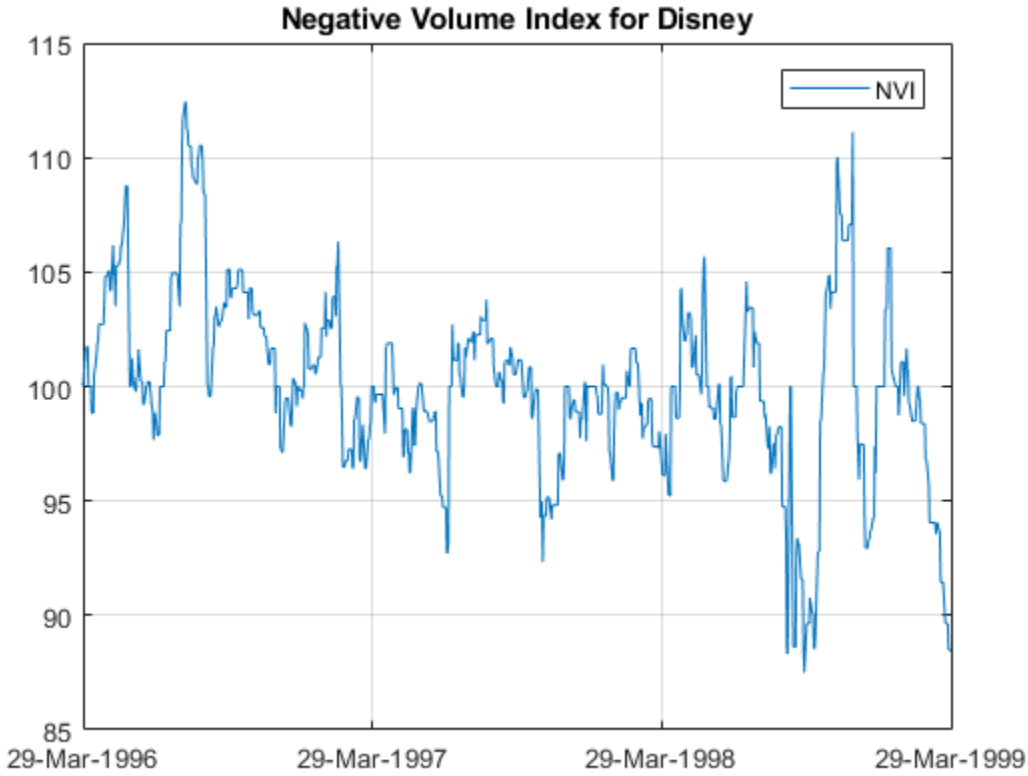
`nvits = negvolidx(tsobj,initnvi,'ParameterName',ParameterValue, ...)` accepts parameter name/ parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Negative Volume Index

This example shows how to compute the negative volume index for Disney stock and plot the results.

```
load disney.mat
dis_NegVol = negvolidx(dis);
plot(dis_NegVol)
title('Negative Volume Index for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 193–194.

See Also

`onbalvol` | `posvolidx`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

nomrr

Nominal rate of return

Syntax

```
Return = nomrr(Rate,NumPeriods)
```

Arguments

`Rate`	Effective annual percentage rate. Enter as a decimal fraction.
`NumPeriods`	Number of compounding periods per year, an integer.

Description

`Return = nomrr(Rate,NumPeriods)` calculates the nominal rate of return.

Examples

Calculate the Nominal Rate of Return

This example shows how to calculate the nominal rate of return based on an effective annual percentage rate of 9.38% compounded monthly.

```
Return = nomrr(0.0938, 12)
```

```
Return = 0.0900
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`effrr` | `irr` | `mirr` | `taxedrr` | `xirr`

Topics

"Analyzing and Computing Cash Flows" on page 2-21

Introduced before R2006a

nweekdate

Date of specific occurrence of weekday in month

Syntax

```
Date = nweekdate(n,Weekday,Year,Month)
Date = nweekdate(___,Same,outputType)
```

Description

`Date = nweekdate(n,Weekday,Year,Month)` returns the date number for the specific occurrence of the weekday in the given `Year` and `Month`.

Any input argument can contain multiple values, but if so, all other input arguments 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 character vectors.

`Date = nweekdate(___,Same,outputType)` returns the date number for the specific occurrence of the weekday in the given `Year` and `Month` and also contains the optional arguments for weekday `Same` and `outputType`.

Examples

Determine the Date of a Specific Occurrence of a Weekday in a Month

Find the first Thursday in May 2001.

```
Date = nweekdate(1, 5, 2001, 5);
datestr(Date)
```

```
ans =
'03-May-2001'
```

Find the first Thursday in May 2001 returned as a datatime array.

```
Date = nweekdate(1, 5, 2001, 5,[],'datetime')
```

```
Date = datetime
   03-May-2001
```

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'
```

Find the third Monday in February for 2001, 2002, and 2003.

```
Year = [2001:2003];
Date = nweekdate(3, 2, Year, 2);
datestr(Date)
```

```
ans = 3x11 char array
    '19-Feb-2001'
    '18-Feb-2002'
    '17-Feb-2003'
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`n` — Nth occurrence of weekday in a month

integer with value `1` through `5` | vector of integers with values `1` through `5`

Nth occurrence of the weekday in a month, specified as an integer or a vector of integers from `1` through `5`.

If `n` is larger than the last occurrence of `Weekday`, the output `Date` = `0`.

Data Types: `single` | `double`

`Weekday` — Weekday whose date you seek

integer with value `1` through `7` | vector of integers with values `1` through `7`

Weekday whose date you seek, specified as an integer or a vector of integers from `1` through `7`.

- `1` — Sunday
- `2` — Monday
- `3` — Tuesday
- `4` — Wednesday
- `5` — Thursday
- `6` — Friday
- `7` — Saturday

Data Types: `single` | `double`

`Year` — Year to determine occurrence of weekday

4-digit integer | vector of 4-digit integers

Year to determine occurrence of weekday, specified as a 4-digit integer or vector of 4-digit integers.

Data Types: `single` | `double`

`Month` — Month to determine occurrence of weekday

integer with value `1` through `12` | vector of integers with values `1` through `12`

Month to determine occurrence of weekday, specified as an integer or vector of integers with values `1` through `12`.

Data Types: `single` | `double`

`Same` — Weekday that must occur in same week with `Weekday`

`0` = ignore (default) | integer with value `0` through `7` | vector of integers with values `0` through `7`

Weekday that must occur in the same week with `Weekday`, specified as an integer or a vector of integers from `0` through `7`, where `0` = ignore (default) and `1` through `7` are as for `Weekday`.

Data Types: `single` | `double`

`outputType` — Year to determine days
`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

A character vector specified as either `'datenum'` or `'datetime'`. The output `Date` is in serial date format if `'datenum'` is specified, or datetime format if `'datetime'` is specified. By default the output `Date` is in serial date format.

Data Types: `single` | `double`

Output Arguments

`Date` — Date of specific occurrence of weekday in month
serial date number | date character vector

Date of specific occurrence of weekday in month, returned as a serial date number or date character vector.

The type of the output for `Date` depends on the input `outputType`. If this variable is `'datenum'`, `Date` is a serial date number. If `outputType` is `'datetime'`, then `Date` is a datetime array. By default, `outputType` is set to `'datenum'`.

See Also

`datetime` | `fbusdate` | `lbusdate` | `lweekdate`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

nyseclosures

New York Stock Exchange closures from 1885 to 2070

Syntax

```
Closures = nyseclosures
[Closures,SatTransition] = nyseclosures(StartDate,EndDate,
WorkWeekFormat)
```

Description

`Closures = nyseclosures` returns a vector of serial date numbers for all known or anticipated closures from January 1, 1885 to December 31, 2070.

Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures from 1885 to 1952 are based on a 6-day workweek. `nyseclosures` contains all holiday and special non-trading days for the New York Stock Exchange from 1885 through 2070 based on a six-day work week (always closed on Sundays).

`[Closures,SatTransition] = nyseclosures(StartDate,EndDate, WorkWeekFormat)`, using optional input arguments, returns a vector of serial date numbers corresponding to market closures between `StartDate` and `EndDate`, inclusive.

Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures from 1885 to 1952 are based on a 6-day workweek. `nyseclosures` contains all holiday and special non-trading days for the New York Stock Exchange from 1885 through 2070 based on a six-day work week (always closed on Sundays). Use `WorkWeekFormat` to modify the list of dates.

Examples

Find NYSE Closures

Find the NYSE closures for 1899:

```
datestr(nyseclosures('1-jan-1899','31-dec-1899'),'dd-mmm-yyyy ddd')
```

```
ans = 16x15 char array
    '02-Jan-1899 Mon'
    '11-Feb-1899 Sat'
    '13-Feb-1899 Mon'
    '22-Feb-1899 Wed'
    '31-Mar-1899 Fri'
    '29-May-1899 Mon'
    '30-May-1899 Tue'
    '03-Jul-1899 Mon'
    '04-Jul-1899 Tue'
    '04-Sep-1899 Mon'
    '29-Sep-1899 Fri'
    '30-Sep-1899 Sat'
    '07-Nov-1899 Tue'
    '25-Nov-1899 Sat'
    '30-Nov-1899 Thu'
    '25-Dec-1899 Mon'
```

Find the NYSE closures for 1899 using a datetime array:

```
[Closures,SatTransition] = nyseclosures(datetime('1-jan-1899','Locale','en_US'),'30-Jur
```

```
Closures = 7x1 datetime array
   02-Jan-1899
   11-Feb-1899
   13-Feb-1899
   22-Feb-1899
   31-Mar-1899
   29-May-1899
   30-May-1899


SatTransition = datetime
   29-Sep-1952
```

Find the NYSE closure dates using the `'Archaic'` value for `WorkWeekFormat`:

```
datestr(nyseclosures('1-sep-1952','31-oct-1952','a'),1)
```

```
ans = 10x11 char array
    '01-Sep-1952'
    '06-Sep-1952'
    '13-Sep-1952'
    '20-Sep-1952'
    '27-Sep-1952'
    '04-Oct-1952'
    '11-Oct-1952'
    '13-Oct-1952'
    '18-Oct-1952'
    '25-Oct-1952'
```

The exchange was closed on Saturdays for much of 1952 before the official transition to a 5-day workweek.

- "Handle and Convert Dates" on page 2-2

Input Arguments

`StartDate` — Start date

start of default date range, January 1, 1885 (default) | serial date number | date character vector | datetime object

Start date, specified using a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

end of default date range, December 31, 2070 (default) | serial date number | date character vector | datetime object

End date, specified using a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`WorkWeekFormat` — Method to handle the workweek

`'Implicit'` (default) | date character vector with values `'Modern'`, `'Implicit'`, or `'Archaic'`

Method to handle the workweek, specified using a date character vector with values `'Modern'`, `'Implicit'`, or `'Archaic'`. This function accepts the first letter for each method as input and is not case-sensitive. Acceptable values are:

- `'Modern'` — 5-day workweek with all Saturday trading days removed.
- `'Implicit'` — 6-day workweek until 1952 and 5-day week afterward (no need to exclude Saturdays).
- `'Archaic'` — 6-day workweek throughout and Saturdays treated as closures after 1952.

Data Types: `char`

Output Arguments

`Closures` — Market closures between `StartDate` and `EndDate`, inclusive

vector

Market closures between the `StartDate` and `EndDate`, inclusive, returned as a vector of dates.

If `StartDate` or `EndDate` are all either serial date numbers or date character vectors, both `Closures` and `SatTransition` are returned as serial date numbers. If either `StartDate` or `EndDate` are datetime arrays, both `Closures` and `SatTransition` are returned as datetime arrays.

If both `StartDate` and `EndDate` are not specified or are empty, `Closures` contains all known or anticipated closures from January 1, 1885 to December 31, 2070 based on a `WorkWeekFormat` of `'implicit'`.

`SatTransition` — Date of transition for New York Stock Exchange from 6-day workweek to 5-day workweek

serial date number | datetime array

Date of transition for the New York Stock Exchange from a 6-day workweek to a 5-day workweek, returned as the date September 29, 1952 (serial date number `713226`).

If `StartDate` or `EndDate` are all either serial date numbers or date character vectors, both `Closures` and `SatTransition` are returned as serial date numbers. If either

`StartDate` or `EndDate` are datetime arrays, both `Closures` and `SatTransition` are returned as datetime arrays.

Definitions

holidays

The `holidays` function is based on a modern 5-day workweek.

This function contains all holidays and special nontrading days for the New York Stock Exchange from January 1, 1885 to December 31, 2070.

Since the New York Stock Exchange was open on Saturdays before September 29, 1952, exact closures from 1885 to 2070 should include Saturday trading days. To capture these dates, use `nyseclosures`. The results from `holidays` and `nyseclosures` are identical if the `WorkWeekFormat` in `nyseclosures` is `'Modern'`.

See Also

`busdate` | `createholidays` | `datetime` | `fbusdate` | `holidays` | `isbusday` | `lbusdate`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 15-2
“UICalendar User Interface” on page 15-4

Introduced before R2006a

onbalvol

On-Balance Volume (OBV)

Syntax

```
obv = onbalvol(closep,tvolume)

obv = onbalvol([closeptvolume])

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([closeptvolume])` 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

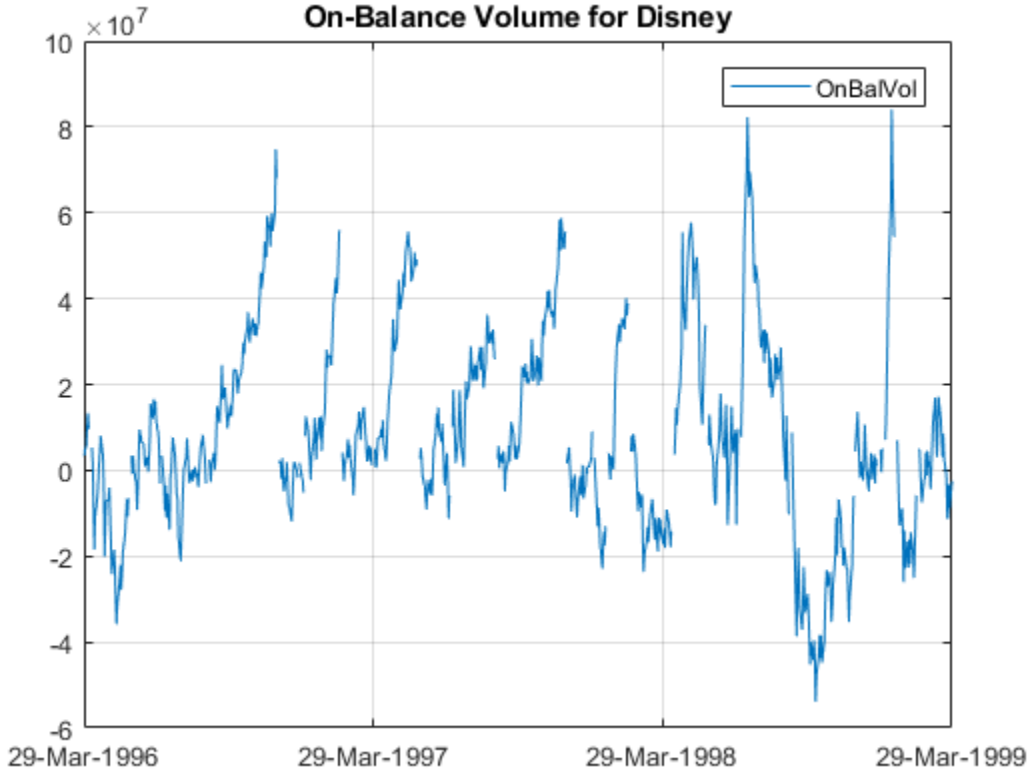
Parameter values are the character vectors that represent the valid parameter names.

Examples

Calculate the On-Balance Volume (OBV)

This example shows how to calculate the OBV for Disney stock and plot the results.

```
load disney.mat
dis_OnBalVol = onbalvol(dis);
plot(dis_OnBalVol)
title('On-Balance Volume for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 207–209.

See Also

`negvolidx`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

opprofit

Option profit

Syntax

```
Profit = opprofit(AssetPrice,Strike,Cost,PosFlag,OptType)
```

Description

`Profit = opprofit(AssetPrice,Strike,Cost,PosFlag,OptType)` returns the profit of an option.

Examples

Calcualte the Profit of an Option

This example shows how to return the profit of an option. For example, consider buying (going long on) a call option with a strike price of $90 on an underlying asset with a current price of $100 for a cost of $4.

```
Profit = opprofit(100, 90, 4, 0, 0)
```

```
Profit = 6
```

- “Pricing and Analyzing Equity Derivatives” on page 2-48
- “Greek-Neutral Portfolios of European Stock Options” on page 10-19
- “Plotting Sensitivities of an Option” on page 10-31
- “Plotting Sensitivities of a Portfolio of Options” on page 10-34

Input Arguments

`AssetPrice` — Asset price
numeric

Asset price, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`Strike` — Strike or exercise price
numeric

Strike or exercise price, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`Cost` — Cost of option
numeric

Cost of the option, specified as a scalar or a `NINST`-by-`1` vector.

Data Types: `double`

`PosFlag` — Option position
`0` = long | `1` = short

Option position, specified as a scalar or a `NINST`-by-`1` vector using the values `0` (long) or `1` (short).

Data Types: `logical`

`OptType` — Option type
`0` = call option | `1` = put option

Option type, specified as a scalar or a `NINST`-by-`1` vector using the values `0` (call option) or `1` (put option).

Data Types: `logical`

Output Arguments

`Profit` — Option profit
vector

Option profit, returned as a scalar or a `NINST`-by-`1` vector.

See Also

`binprice` | `blsprice`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-48
"Greek-Neutral Portfolios of European Stock Options" on page 10-19
"Plotting Sensitivities of an Option" on page 10-31
"Plotting Sensitivities of a Portfolio of Options" on page 10-34

Introduced before R2006a

payadv

Periodic payment given number of advance payments

Syntax

```
Payment = payadv(Rate,NumPeriods,PresentValue,FutureValue,Advance)
```

Arguments

`Rate`	Lending or borrowing rate per period. Enter as a decimal fraction. Must be greater than or equal to `0`.
`NumPeriods`	Number of periods in the life of the instrument.
`PresentValue`	Present value of the instrument.
`FutureValue`	Future value or target value to be attained after `NumPeriods` periods.
`Advance`	Number of advance payments. If the payments are made at the beginning of the period, add `1` to `Advance`.

Description

`Payment = payadv(Rate,NumPeriods,PresentValue,FutureValue,Advance)` returns the periodic payment given a number of advance payments.

Examples

Compute the Periodic Payment

This example shows how to compute the periodic payment, given a number of advance payments. For example, the present value of a loan is $1000.00 and it will be paid in full

in 12 months. The annual interest rate is 10% and three payments are made at closing time.

```
Payment = payadv(0.1/12, 12, 1000, 0, 3)

Payment = 85.9389
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

amortize | payodd | payper

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

payodd

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

Compute the Payment for a Loan or Annuity With an Odd First Period

This example shows how to return the payment for a loan or annuity with an odd first period. For example, consider a two-year loan for $4000 that has an annual interest rate of 11% and the first payment will be made in 36 days.

```
Payment = payodd(0.11/12, 24, 4000, 0, 36)
```

```
Payment = 186.7731
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`amortize` | `payadv` | `payper`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

payper

Periodic payment of loan or annuity

Syntax

```
Payment = payper(Rate,NumPeriods,PresentValue,FutureValue,Due)
```

Arguments

`Rate`	Interest rate per period. Enter as a decimal fraction.
`NumPeriods`	Number of payment periods in the life of the instrument.
`PresentValue`	Present value of the instrument.
`FutureValue`	(Optional) Future value or target value to be attained after `NumPeriods` periods. Default = `0`.
`Due`	(Optional) When payments are due: `0` = end of period (default), or `1` = beginning of period.

Description

`Payment = payper(Rate,NumPeriods,PresentValue,FutureValue,Due)` returns the periodic payment of a loan or annuity.

Examples

Compute the Periodic Payment of a Loan or Annuity

This example shows how to find the monthly payment for a three-year loan of $9000 with an annual interest rate of 11.75%.

```
Payment = payper(0.1175/12, 36, 9000, 0, 0)
```

```
Payment = 297.8553
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`amortize` | `fvfix` | `payadv` | `payodd` | `pvfix`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

payuni

Uniform payment equal to varying cash flow

Syntax

```
Series = payuni(CashFlow,Rate)
```

Arguments

`CashFlow`	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
`Rate`	Periodic interest rate. Enter as a decimal fraction.

Description

`Series = payuni(CashFlow,Rate)` returns the uniform series value of a varying cash flow.

Examples

This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 8%.

Year 1	$2000
Year 2	$1500
Year 3	$3000
Year 4	$3800
Year 5	$5000

To calculate the uniform series value

```
Series = payuni([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
Series =

         429.63
```

See Also

`fvfix` | `fvvar` | `irr` | `pvfix` | `pvvar`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

pcalims

Linear inequalities for individual asset allocation

As an alternative to `pcalims`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

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` is 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 stock to 0.5 and CSCO to 0.8, while letting the maximum weight in INTC float.

Asset	IBM	INTC	CSCO
Minimum Weight	0	0	0
Maximum Weight	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
Minimum Weight	0	0	0
Maximum Weight	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` | `portstats`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Selection and Risk Aversion" on page 3-9
"Active Returns and Tracking Error Efficient Frontier" on page 3-43
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

pcgcomp

Linear inequalities for asset group comparison constraints

As an alternative to `pcgcomp`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

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`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Selection and Risk Aversion" on page 3-9
"Active Returns and Tracking Error Efficient Frontier" on page 3-43
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

pcglims

Linear inequalities for asset group minimum and maximum allocation

As an alternative to `pcglims`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

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`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Selection and Risk Aversion" on page 3-9
"Active Returns and Tracking Error Efficient Frontier" on page 3-43
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

pcpval

Linear inequalities for fixing total portfolio value

As an alternative to `pcpval`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

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`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Selection and Risk Aversion” on page 3-9
“Active Returns and Tracking Error Efficient Frontier” on page 3-43
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

peravg

Periodic average of FINTS object

Syntax

```
avgfts = peravg(tsobj)

avgfts = peravg(tsobj,numperiod)

avgfts = peravg(tsobj,daterange)
```

Arguments

`tsobj`	Financial time series object
`numperiod`	(Optional) Integer specifying the number of data points over which each periodic average should be averaged
`daterange`	(Optional) Time period over which the data is averaged

Description

`peravg` calculates periodic averages of a financial time series object. Periodic averages are calculated from the values per period defined. If the period supplied is a character vector, it is assumed as a range of date character vector. If the period is entered as numeric, the number represents the number of data points (financial time series periods) to be included in a period for the calculation. For example, if you enter `'01/01/98::01/01/99'` as the period input argument, `peravg` returns the average of the time series between those dates, inclusive. However, if you enter the number `5` as the period input, `peravg` returns a series of averages from the time series data taken 5 date points (financial time series periods) at a time.

`avgfts = peravg(tsobj,numperiod)` returns a structure `avgfts` that contains the periodic (per `numperiod` periods) average of the financial time series object. `avgfts` has field names identical to the data series names of `tsobj`.

`avgfts = peravg(tsobj,daterange)` returns a structure `avgfts` that contains the periodic (as specified by `daterange`) average of the financial time series object. `avgfts` has field names identical to the data series names of `tsobj`.

Note `peravg` calculates periodic averages of a FINTS object. Periodic averages are calculated from the values per period defined. If the period supplied is a character vector, it is assumed as a range of date character vectors. If the period is entered as numeric, the number represents the number of data points to be included in a period for the calculation.

Examples

If you enter `01-Jan-2001::03-Jan-2001` as the period input argument, `peravg` returns the average of the time series between those dates, inclusive. However, if you enter the number `5` as the period input, `peravg` returns a series of averages from the time series data, taken 5 date points at a time.

```
%% Create the FINTS object %%
dates = ['01-Jan-2001';'01-Jan-2001'; '02-Jan-2001'; ...
'02-Jan-2001'; '03-Jan-2001';'03-Jan-2001'];
times = ['11:00';'12:00';'11:00';'12:00';'11:00';'12:00'];
dates_times = cellstr([dates, repmat(' ', size(dates, 1), 1), times]);
data  = [(1:6)', 2*(1:6)'];
myFts = fints(dates_times, data, {'Data1', 'Data2'}, 1, 'My first FINTS')

%% Create the FINTS object %%
[p, pFts] = peravg(myFts, 3)

 p =

  Data1: [2 5]
  Data2: [4 10]

pFts =

  desc:  My first FINTS
  freq:  Daily (1)

  'dates:  (2)'    'times:  (2)'    'Data1:  (2)'    'Data2:  (2)'
  '02-Jan-2001'    '11:00'          [          2]    [          4]
  '03-Jan-2001'    '12:00'          [          5]    [         10]

[p, pFts] = peravg(myFts,'01-Jan-2001 12:00::03-Jan-2001 11:00')
```

```
p =
   Data1: 3.5000
   Data2: 7

 pFts =

   desc:  My first FINTS
   freq:  Daily (1)

   'dates:  (1)'    'times:  (1)'    'Data1:  (1)'    'Data2:  (1)'
   '03-Jan-2001'    '11:00'          [     3.5000]    [          7]
```

See Also

`mean` | `tsmovavg`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

periodicreturns

Periodic total returns from total return prices

Syntax

```
TotalReturn = periodicreturns(TotalReturnPrices)
```

```
TotalReturn = periodicreturns(TotalReturnPrices,Period)
```

Arguments

`TotalReturnPrices`	`TotalReturnPrices` can be the number of observations (`NUMOBS`) by number of assets (`NASSETS + 1`) matrix of total return prices for a given security. Column 1 contains MATLAB serial date numbers. The remaining columns contain total return price data. In addition, `TotalReturnPrices` can also be a table where the first column of the table represents the dates (as either serial date numbers, date character vectors, or datetime arrays) while the other columns represent the returns data. If a table is used, `TotalReturn` is returned as a table.
`Period`	(Optional) Periodicity flag used to compute total returns: `'d'` = daily values (default) `'w'` = weekly values `'m'` = monthly values *n* = rolling return periodic values, where *n* is an integer

Description

`TotalReturn = periodicreturns(TotalReturnPrices)` calculates the daily total returns from a daily total return price series.

`TotalReturn = periodicreturns(TotalReturnPrices,Period)` calculates the total returns for a periodicity that you specify from a daily total return price series.

If `TotalReturnPrices` input is a matrix, `TotalReturn` is a `NUMOBS`-by-`NASSETS + 1` matrix containing month-end dates and return values. Each row represents an observation. Column 1 contains month-end dates in MATLAB serial date number format. The remaining columns contain monthly return values.

Note Although input returns can have dates in either ascending or descending order, output total returns in `TotalReturn` have dates in ascending order, with the earliest date in the first row `TotalReturn`, and the most recent date in the last row of `TotalReturn`.

If `TotalReturnPrices` input is a table where the first column of the table represents the dates (as either serial date numbers, date character vectors, or datetime arrays) while the other columns represent the returns data, `TotalReturn` is returned as a table.

Examples

Compute TotalReturn Using datetime Input for TotalReturnPrices

Compute `TotalReturn` returned as a table using `datetime` input in a table for `TotalReturnPrices`.

```
Dates = datetime(2015,1,1:10,'Locale','en_US')';
Prices = [0.01 0.03 0.1  -0.05  0.02 0.07 0.03 -0.01 -0.02 0.01]';
TotalReturnPrices = table(Dates,Prices);
TotalReturn = periodicreturns(TotalReturnPrices)
```

```
TotalReturn=9x2 table
       Dates         Prices
    ___________    ________

    02-Jan-2015           2
    03-Jan-2015      2.3333
    04-Jan-2015        -1.5
    05-Jan-2015        -1.4
    06-Jan-2015         2.5
    07-Jan-2015    -0.57143
    08-Jan-2015     -1.3333
    09-Jan-2015           1
```

```
10-Jan-2015        -1.5
```

- “Portfolio Construction Examples” on page 3-7

See Also

`totalreturnprice`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

plot

Plot data series

Syntax

```
plot(tsobj)

hp = plot(tsobj)

plot(tsobj,linefmt)

hp = plot(tsobj,linefmt)

plot(..., volumename,bar)

hp = plot(..., volumename,bar)
```

Arguments

`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, barh`.

Description

`plot(tsobj)` plots the data series contained in the object `tsobj`. Each data series is a line. `plot` automatically generates a legend and dates on the *x*-axis. Grid is turned on by default. `plot` uses the default color order as if plotting a matrix.

The `plot` command automatically creates subplots when multiple time series are encountered, and they differ greatly on their decimal scales. For example, subplots are generated if one time series data set is in the 10s and another is in the 10,000s.

`hp = plot(tsobj)` also 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`. 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 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)` also 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 off the legend, enter `legend off` at the MATLAB command line. Once you turn it off, the legend is deleted. To turn it back on, recreate it using the `legend` command as if you are creating it for the first time. To turn off the grid, enter `grid off`. To turn it back on, enter `grid on`.

See Also

`candle` | `chartfts` | `grid` | `highlow` | `legend` | `plot`

Topics

“Charting Financial Data” on page 2-14

Introduced before R2006a

plotFrontier

Plot efficient frontier

Use the `plotFrontier` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to plot the efficient frontier for a portfolio object.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
[prsk,pret] = plotFrontier(obj)

[prsk,pret] = plotFrontier(obj,varargin)
```

Description

`[prsk,pret] = plotFrontier(obj)` plots the efficient frontier for a portfolio object.

`[prsk,pret] = plotFrontier(obj,varargin)` plot the efficient frontier for a portfolio object with multiple types of input methods. There are four ways to use `plotFrontier`:

- Method 1 — Given a portfolio object `obj`, estimate the efficient frontier with default number of 10 portfolios on the frontier.
- Method 2 — Given a portfolio object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier.
- Method 3 — Given a portfolio object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios. This method assumes that you provide valid efficient portfolios as input.
- Method 4 — Given a portfolio object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier. This method assumes that you provide valid inputs for efficient portfolio risks and returns.

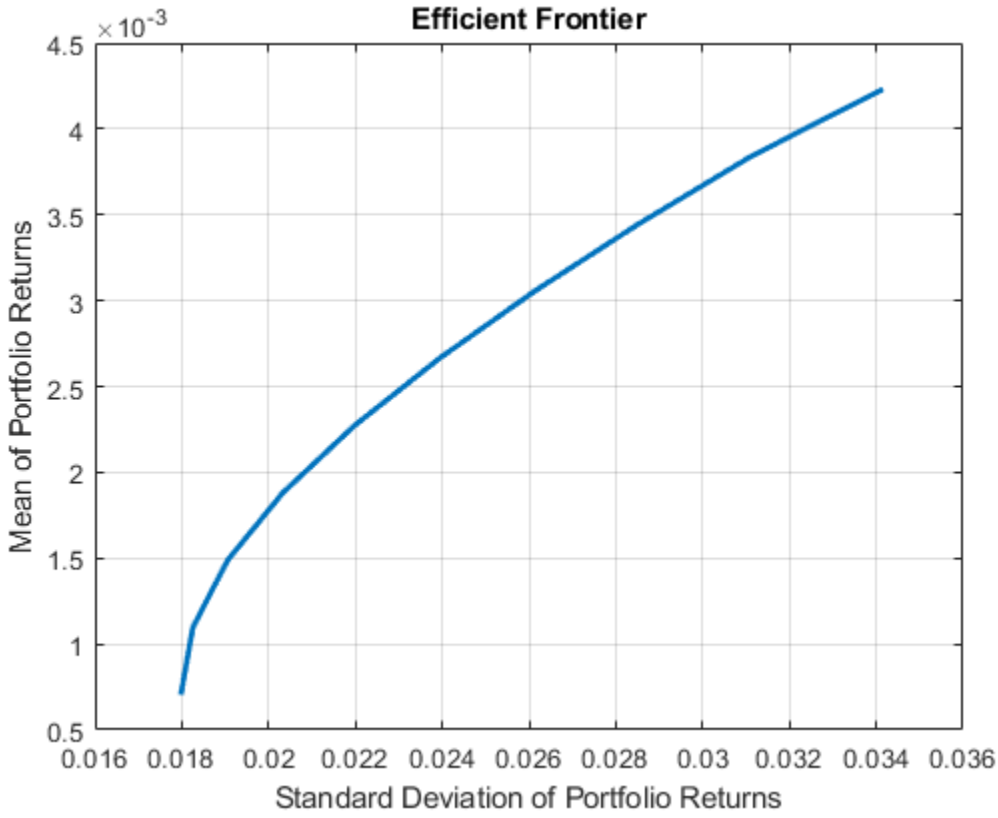
Note `plotFrontier` handles multiple input formats as described above. Given an asset universe with `NumAssets` assets and an efficient frontier with `NumPorts` portfolios, remember that portfolio weights are `NumAsset`-by-`NumPorts` matrices and that portfolio risks and returns are `NumPorts` column vectors.

Examples

Plot the Efficient Frontier for the Portfolio Object

Given a portfolio `p`, plot the efficient frontier.

```
load CAPMuniverse

p = Portfolio('AssetList',Assets(1:12));
p = estimateAssetMoments(p, Data(:,1:12),'missingdata',true);
p = setDefaultConstraints(p);
plotFrontier(p);
```

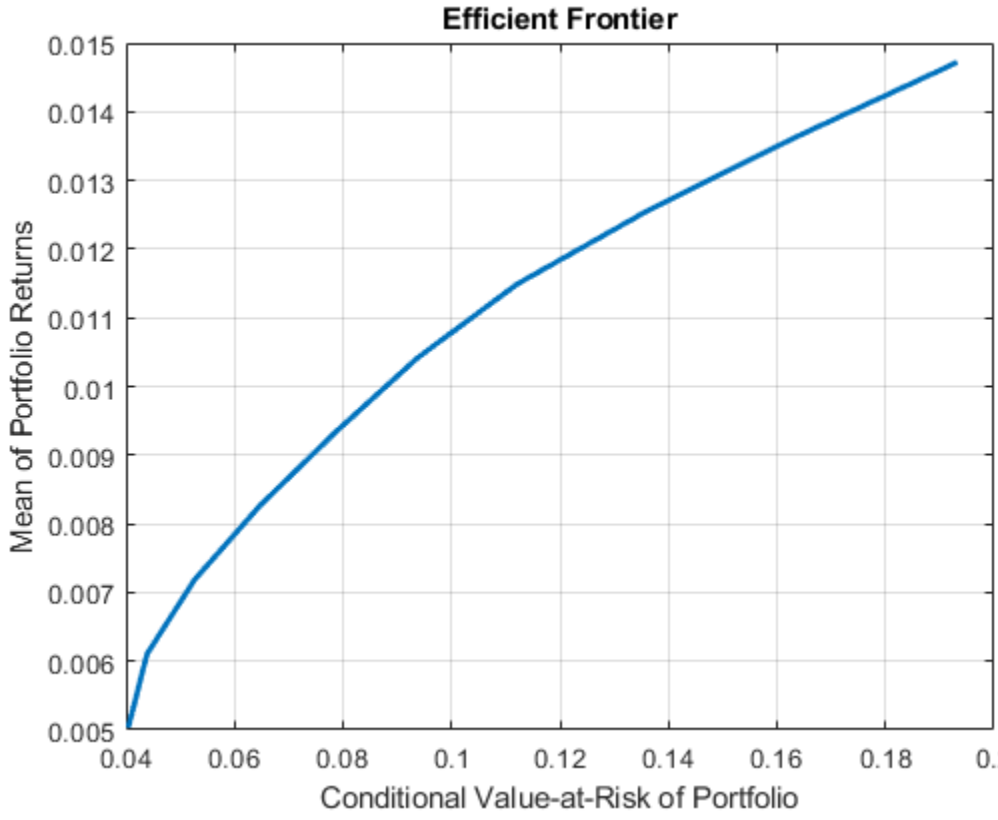


Plot the Efficient Frontier for the PortfolioCVaR Object

Given a PortfolioCVaR `p`, plot the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;
```

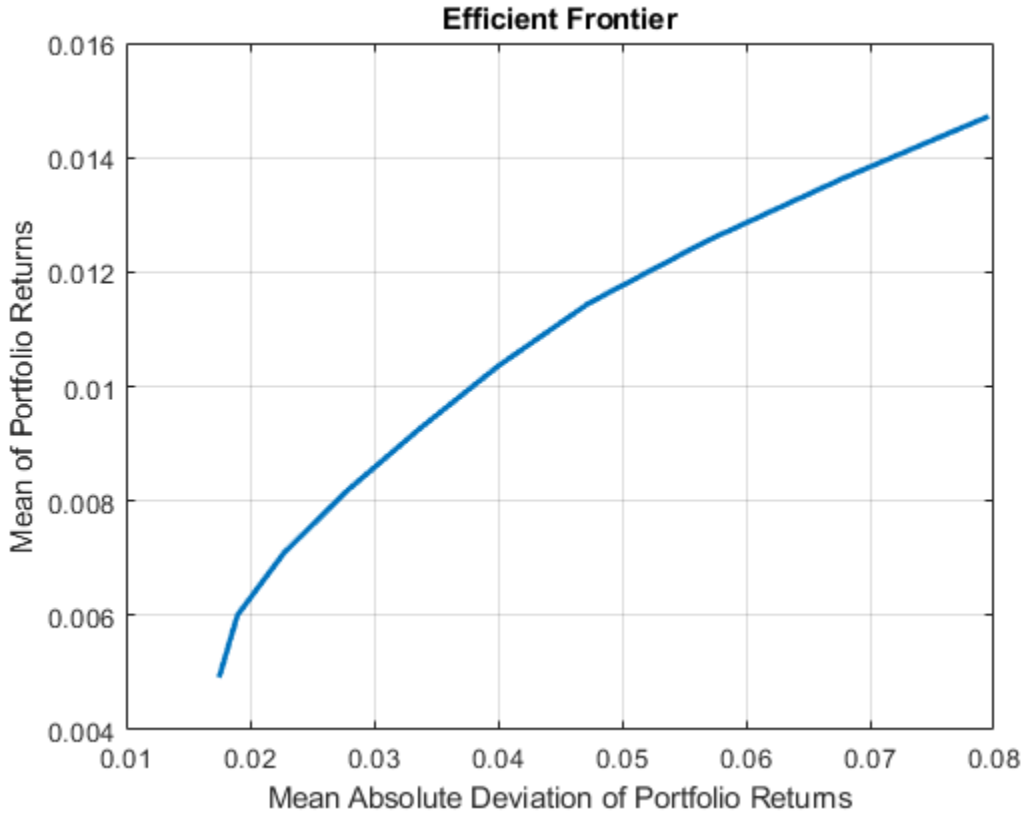
```
AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```



Plot Efficient Frontier for PortfolioMAD Object

Given a PortfolioMAD `p`, plot the efficient frontier.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

plotFrontier(p);
```



- "Plotting the Efficient Frontier for a Portfolio Object" on page 4-132
- "Plotting the Efficient Frontier for a PortfolioCVaR Object" on page 5-123
- "Plotting the Efficient Frontier for a PortfolioMAD Object" on page 6-115
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio
object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`varargin` — Optional input methods

vector for `NumPorts`, `PortRisk`, `PortReturn`, or `PortWeights`

Optional input methods, specified as `varargin` can be `NumPorts`, `PortRisk`, `PortReturn`, or `PortWeights` depending on which of these four input methods you use:

- Method 1 — Given a portfolio object `obj`, estimate the efficient frontier with the default number of 10 portfolios on the frontier:

  ```
  [prsk, pret] = plotFrontier(obj)
  ```

- Method 2 — Given a portfolio object `obj`, estimate the efficient frontier with a specified number of portfolios `NumPorts` on the frontier:

  ```
  [prsk, pret] = plotFrontier(obj,NumPorts)
  ```

- Method 3 — Given a portfolio object `obj` with estimated efficient portfolios in `PortWeights`, plot the efficient frontier with those portfolios:

  ```
  [prsk, pret] = plotFrontier(obj,PortWeights)
  ```

  The `plotFrontier` function assumes that you provide valid efficient portfolios as inputs.

- Method 4 — Given a portfolio object `obj` with estimated portfolio risks (`PortRisk`) and returns (`PortReturn`), plot the efficient frontier:

  ```
  [prsk, pret] = plotFrontier(obj,PortRisk,PortReturn)
  ```

  The `plotFrontier` function assumes that you provide valid efficient portfolio risks and returns as inputs.

Data Types: `double`

Output Arguments

`prsk` — Estimated efficient portfolio risks (standard deviation of returns

vector

Estimated efficient portfolio risks (standard deviation of returns, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note

- If the portfolio object has a name in the `Name` property, the name is displayed as the title of the plot. Otherwise, the plot is just labeled "Efficient Frontier."
- If the portfolio object has an initial portfolio in the `InitPort` property, the initial portfolio is plotted and labeled.
- If portfolio risks and returns are inputs, make sure that risks come first in the calling sequence. In addition, if portfolio risks and returns are not sorted in ascending order, this method performs the sort. On output, the sorted moments are returned.

`pret` — Estimated efficient portfolio returns

vector

Estimated efficient portfolio returns, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note

- If the portfolio object has a name in the `Name` property, the name is displayed as the title of the plot. Otherwise, the plot is labeled "Efficient Frontier."
- If the portfolio object has an initial portfolio in the `InitPort` property, the initial portfolio is plotted and labeled.
- If portfolio risks and returns are inputs, make sure that risks come first in the calling sequence. In addition, if portfolio risks and returns are not sorted in ascending order, this method performs the sort. On output, the sorted moments are returned.

Tips

You can also use dot notation to plot the efficient frontier.

```
[prsk, pret] = obj.plotFrontier;
```

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

Topics

"Plotting the Efficient Frontier for a Portfolio Object" on page 4-132
"Plotting the Efficient Frontier for a PortfolioCVaR Object" on page 5-123
"Plotting the Efficient Frontier for a PortfolioMAD Object" on page 6-115
"Portfolio Optimization Examples" on page 4-147

Introduced in R2011a

plus

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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

pointfig

Point and figure chart

Syntax

```
pointfig(Asset)
```

Description

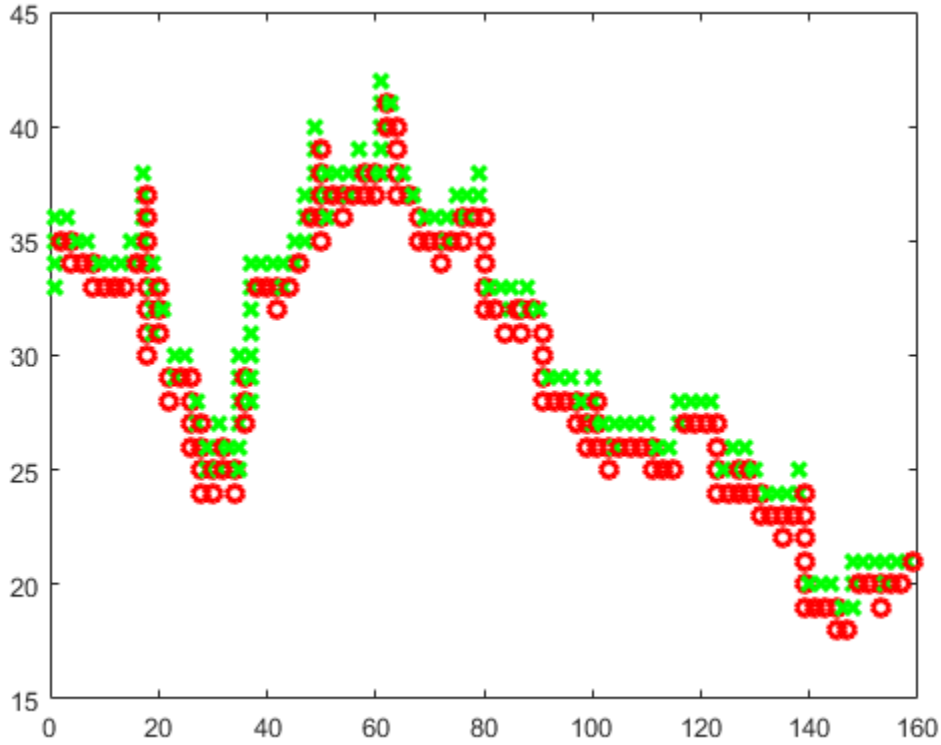
`pointfig(Asset)` plots a point and figure chart for a vector of price data `Asset`. Upward price movements are plotted as `X`'s and downward price movements are plotted as `O`'s.

Examples

Create a Point and Figure Chart for an Equity's Closing Prices

Using the price data for the equity DIS, plot a point and figure chart for the closing prices.

```
load disney;
pointfig(dis_CLOSE)
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `dateaxis` | `highlow` | `movavg`

Topics

“Charting Financial Data” on page 2-14

Introduced before R2006a

portalloc

Optimal capital allocation to efficient frontier portfolios

Syntax

```
[RiskyRisk,RiskyReturn,RiskyWts,RiskyFraction,OverallRisk,OverallReturn] = portalloc(Pc
```

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 from 2.0 through 4.0 (Default = `3`). **Note** Consider that a less risk-averse investor would be expected to accept much greater risk and, consequently, a more risk-averse investor would accept less risk for a given level of return. Therefore, making the `RiskAversion`argument higher reflects the risk-return tradeoff in the data.

Description

`[RiskyRisk,RiskyReturn,RiskyWts,RiskyFraction,OverallRisk,OverallReturn] = portalloc(PortRisk,PortReturn,PortWts, RisklessRate,BorrowRate,RiskAversion)` computes the optimal risky portfolio, and the optimal allocation of funds between the risky portfolio and the risk-free asset.

`RiskyRisk` is the standard deviation of the optimal risky portfolio.

`RiskyReturn` is the expected return of the optimal risky portfolio.

`RiskyWts` is a `1`-by-`NASSETS` vector of weights allocated to the optimal risky portfolio. The total of all weights in the portfolio is 1.

`RiskyFraction` is the fraction of the complete portfolio allocated to the risky portfolio.

`OverallRisk` is the standard deviation of the optimal overall portfolio.

`OverallReturn` is the expected rate of return of the optimal overall portfolio.

`portalloc` generates a plot of the optimal capital allocation if you invoke it without output arguments.

Examples

Compute the Optimal Risky Portfolio

This example shows how to compute the optimal risky portfolio by generating the efficient frontier from the asset data and then finding the optimal risky portfolio and allocate capital. The risk-free investment return is 8%, and the borrowing rate is 12%.

```
ExpReturn = [0.1 0.2 0.15];

ExpCovariance = [0.005   -0.010    0.004
                -0.010    0.040   -0.002
                 0.004   -0.002    0.023];
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn,...
ExpCovariance);
```

```
RisklessRate  = 0.08;
BorrowRate    = 0.12;
RiskAversion  = 3;

[RiskyRisk, RiskyReturn, RiskyWts, RiskyFraction, ...
OverallRisk, OverallReturn] = portalloc(PortRisk, PortReturn,...
PortWts, RisklessRate, BorrowRate, RiskAversion)

RiskyRisk = 0.1283

RiskyReturn = 0.1788

RiskyWts =

    0.0265    0.6023    0.3712


RiskyFraction = 1.1898

OverallRisk = 0.1527

OverallReturn = 0.1899
```

- “Portfolio Construction Examples” on page 3-7

References

Bodie, Kane, and Marcus. *Investments.* Second Edition. Chapters 6 and 7.

See Also

`portrand` | `portstats`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

portalpha

Compute risk-adjusted alphas and returns for one or more assets

Syntax

```
portalpha(Asset,Benchmark)

portalpha(Asset,Benchmark,Cash)

portalpha(Asset,Benchmark,Cash,Choice)

Alpha = portalpha(Asset,Benchmark,Cash,Choice)

[Alpha,RAReturn] = portalpha(Asset,Benchmark,Cash,Choice)
```

Arguments

`Asset`	`NUMSAMPLES x NUMSERIES` matrix with `NUMSAMPLES` observations of asset returns for `NUMSERIES` asset return series.
`Benchmark`	`NUMSAMPLES` vector of returns for a benchmark asset. The periodicity must be the same as the periodicity of `Asset`. For example, if `Asset` is monthly data, then `Benchmark` should be monthly returns.
`Cash`	(Optional) Either a scalar return for a riskless asset or a vector of asset returns to be a proxy for a “riskless” asset. In either case, the periodicity must be the same as the periodicity of `Asset`. For example, if `Asset` is monthly data, then `Cash` must be monthly returns. If no value is supplied, the default value for `Cash` returns is 0.

<table>
<tr><td>Choice</td><td>(Optional) A character vector, or cell array of character vectors to indicate one or more measures to be computed from among a number of risk-adjusted alphas and return measures. The number of choices selected in Choice is NUMCHOICES. The list of choices is given in the following table:
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>'xs'</td><td>Excess Return (no risk adjustment)</td></tr>
<tr><td>'sml'</td><td>Security Market Line</td></tr>
<tr><td>'capm'</td><td>Jensen's Alpha</td></tr>
<tr><td>'mm'</td><td>Modigliani & Modigliani</td></tr>
<tr><td>'gh1'</td><td>Graham-Harvey 1</td></tr>
<tr><td>'gh2'</td><td>Graham-Harvey 2</td></tr>
<tr><td>'all'</td><td>Compute all measures</td></tr>
</table>
Choice is specified by using the code from the table (for example, to select the Modigliani & Modigliani measure, Choice = 'mm'). A single choice is either a character vector or a scalar cell array with a single code from the table.

Multiple choices can be selected with a cell array of character vectors for choice codes (for example, to select both Graham-Harvey measures, Choice = {'gh1','gh2'}). To select all choices, specify Choice = 'all'. If no value is supplied, the default choice is to compute the excess return with Choice = 'xs'. Choice is not case-sensitive.</td></tr>
</table>

Description

Given NUMSERIES assets with NUMSAMPLES returns in a NUMSAMPLES-by-NUMSERIES matrix Asset, a NUMSAMPLES vector of Benchmark returns, and either a scalar Cash return or a NUMSAMPLES vector of Cash returns, compute risk-adjusted alphas and returns for one or more methods specified by Choice.

To summarize the outputs of portalpha:

- `Alpha` is a `NUMCHOICES`-by-`NUMSERIES` matrix of risk-adjusted alphas for each series in `Asset` with each row corresponding to a specified measure in `Choice`.
- `RAReturn` is a `NUMCHOICES`-by-`NUMSERIES` matrix of risk-adjusted returns for each series in `Asset` with each row corresponding to a specified measure in `Choice`.

Note `NaN` values in the data are ignored and, if `NaN`s are present, some results could be unpredictable. Although the alphas are comparable across measures, risk-adjusted returns depend on whether the `Asset` or `Benchmark` is levered or unlevered to match its risk with the alternative. If `Choice = 'all'`, the order of rows in `Alpha` and `RAReturn` follows the order in the table. In addition, `Choice = 'all'` overrides all other choices.

Examples

See “Risk-Adjusted Return” on page 7-12.

References

John Lintner. "The Valuation of Risk Assets and the Selection of Risky Investments in Stocks Portfolios and Capital Budgets." *Review of Economics and Statistics.* Vol. 47, No. 1, February 1965, pp. 13–37.

John R. Graham and Campbell R. Harvey. "Market Timing Ability and Volatility Implied in Investment Newsletters' Asset Allocation Recommendations." *Journal of Financial Economics.* Vol. 42, 1996, pp. 397–421.

Franco Modigliani and Leah Modigliani. "Risk-Adjusted Performance: How to Measure It and Why." *Journal of Portfolio Management.* Vol. 23, No. 2, Winter 1997, pp. 45–54.

Jan Mossin. "Equilibrium in a Capital Asset Market." *Econometrica.* Vol. 34, No. 4, October 1966, pp. 768–783.

William F. Sharpe. "Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk." *Journal of Finance.* Vol. 19, No. 3, September 1964, pp. 425–442.

See Also

`inforatio` | `sharpe`

Topics

“Performance Metrics Illustration” on page 7-4
“Performance Metrics Overview” on page 7-2

Introduced in R2006b

portcons

Portfolio constraints

As an alternative to `portcons`, use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow" on page 4-21.

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, ..., 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.

Constraint Type	Description	Values
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.
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. **Note** For more information using `Custom`, see "Specifying Group Constraints" on page 3-39.

Examples

Constrain a portfolio of three assets:

`Asset`	IBM	HPQ	XOM
`Group`	A	A	B
`Minimum Weight`	0	0	0
`Maximum Weight`	0.5	0.9	0.8

```
NumAssets = 3;
PVal = 1; % Scale portfolio value to 1.
AssetMin = 0;
```

```
AssetMax = [0.5 0.9 0.8];
GroupA = [1 1 0];
GroupB = [0 0 1];
AtoBmax = 1.5 % Value of assets in Group A at most 1.5 times value
              % in group B.

ConSet = portcons('PortValue', PVal, NumAssets,'AssetLims',...
AssetMin, AssetMax, NumAssets, 'GroupComparison',GroupA, NaN,...
AtoBmax, GroupB)

ConSet =

    1.0000    1.0000    1.0000    1.0000
   -1.0000   -1.0000   -1.0000   -1.0000
    1.0000         0         0    0.5000
         0    1.0000         0    0.9000
         0         0    1.0000    0.8000
   -1.0000         0         0         0
         0   -1.0000         0         0
         0         0   -1.0000         0
    1.0000    1.0000   -1.5000         0
```

For instance, one possible solution for portfolio weights that satisfy the constraints is 30% in IBM, 30% in HPQ, and 40% in XOM.

See Also

`pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portopt`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Selection and Risk Aversion” on page 3-9
“Active Returns and Tracking Error Efficient Frontier” on page 3-43
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

Portfolio

Portfolio object for mean-variance portfolio optimization and analysis

Description

The Portfolio object implements mean-variance portfolio optimization. Portfolio objects support functions that are specific to mean-variance portfolio optimization.

The main workflow for portfolio optimization is to create an instance of a `Portfolio` object that completely specifies a portfolio optimization problem and to operate on the `Portfolio` object using supported functions to obtain and analyze efficient portfolios. A mean-variance optimization problem is completely specified with the following three elements:

- A universe of assets with estimates for the prospective mean and covariance of asset total returns for a period of interest.
- A portfolio set that specifies the set of portfolio choices in terms of a collection of constraints.
- A model for portfolio return and risk, which, for mean-variance optimization, is either the gross or net mean of portfolio returns and the standard deviation of portfolio returns.

After you specify these three elements in an unambiguous way, you can solve and analyze portfolio optimization problems. The simplest mean-variance portfolio optimization problem has:

- A mean and covariance of asset total returns
- Nonnegative weights for all portfolios that sum to `1` (the summation constraint is known as a budget constraint)
- Built-in models for portfolio return and risk that use the mean and covariance of asset total returns

Given mean and covariance of asset returns in the variables `AssetMean` and `AssetCovar`, this problem is completely specified by:

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar,...
'LowerBound', 0, 'UpperBudget',1, 'LowerBudget',1)
```

or equivalently by:

```
p = Portfolio;
p = setAssetMoments(p, AssetMean, AssetCovar);
p = setDefaultConstraints(p);
```

For more information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21 and for more detailed information on the theoretical basis for mean-variance optimization, see “Portfolio Optimization Theory” on page 4-3.

Creation

To create a `Portfolio` object, use the `Portfolio` function. For more details on working with a Portfolio object, see:

- “Portfolio Object Properties and Functions” on page 4-23
- “Working with Portfolio Objects” on page 4-23
- “Setting and Getting Properties” on page 4-24
- “Displaying Portfolio Objects” on page 4-25
- “Saving and Loading Portfolio Objects” on page 4-25
- “Estimating Efficient Portfolios and Frontiers” on page 4-25
- “Arrays of Portfolio Objects” on page 4-25
- “Subclassing Portfolio Objects” on page 4-26
- “Conventions for Representation of Data” on page 4-26

Properties

Portfolio	Manage Portfolio object for mean-variance portfolio optimization and analysis

Object Functions

setAssetList	Set up list of identifiers for assets
setInitPort	Set up initial or current portfolio
setDefaultConstraints	Set up portfolio constraints with nonnegative weights that sum to 1

getAssetMoments	Obtain mean and covariance of asset returns from Portfolio object
setAssetMoments	Set moments (mean and covariance) of asset returns for Portfolio object
estimateAssetMoments	Estimate mean and covariance of asset returns from data
setCosts	Set up proportional transaction costs
addEquality	Add linear equality constraints for portfolio weights to existing constraints
addGroupRatio	Add group ratio constraints for portfolio weights to existing group ratio constraints
addGroups	Add group constraints for portfolio weights to existing group constraints
addInequality	Add linear inequality constraints for portfolio weights to existing constraints
getBounds	Obtain bounds for portfolio weights from portfolio object
getBudget	Obtain budget constraint bounds from portfolio object
getCosts	Obtain buy and sell transaction costs from portfolio object
getEquality	Obtain equality constraint arrays from portfolio object
getGroupRatio	Obtain group ratio constraint arrays from portfolio object
getGroups	Obtain group constraint arrays from portfolio object
getInequality	Obtain inequality constraint arrays from portfolio object
getOneWayTurnover	Obtain one-way turnover constraints from portfolio object
setGroups	Set up group constraints for portfolio weights
setInequality	Set up linear inequality constraints for portfolio weights
setBounds	Set up bounds for portfolio weights
setBudget	Set up budget constraints
setCosts	Set up proportional transaction costs
setDefaultConstraints	Set up portfolio constraints with nonnegative weights that sum to 1
setEquality	Set up linear equality constraints for portfolio weights
setGroupRatio	Set up group ratio constraints for portfolio weights
setInitPort	Set up initial or current portfolio
setOneWayTurnover	Set up one-way portfolio turnover constraints
setTurnover	Set up maximum portfolio turnover constraint
setTrackingPort	Set up benchmark portfolio for tracking error constraint
setTrackingError	Set up maximum portfolio tracking error constraint
checkFeasibility	Check feasibility of input portfolios against portfolio object
estimateBounds	Estimate global lower and upper bounds for set of portfolios
estimateFrontier	Estimate specified number of optimal portfolios on the efficient frontier

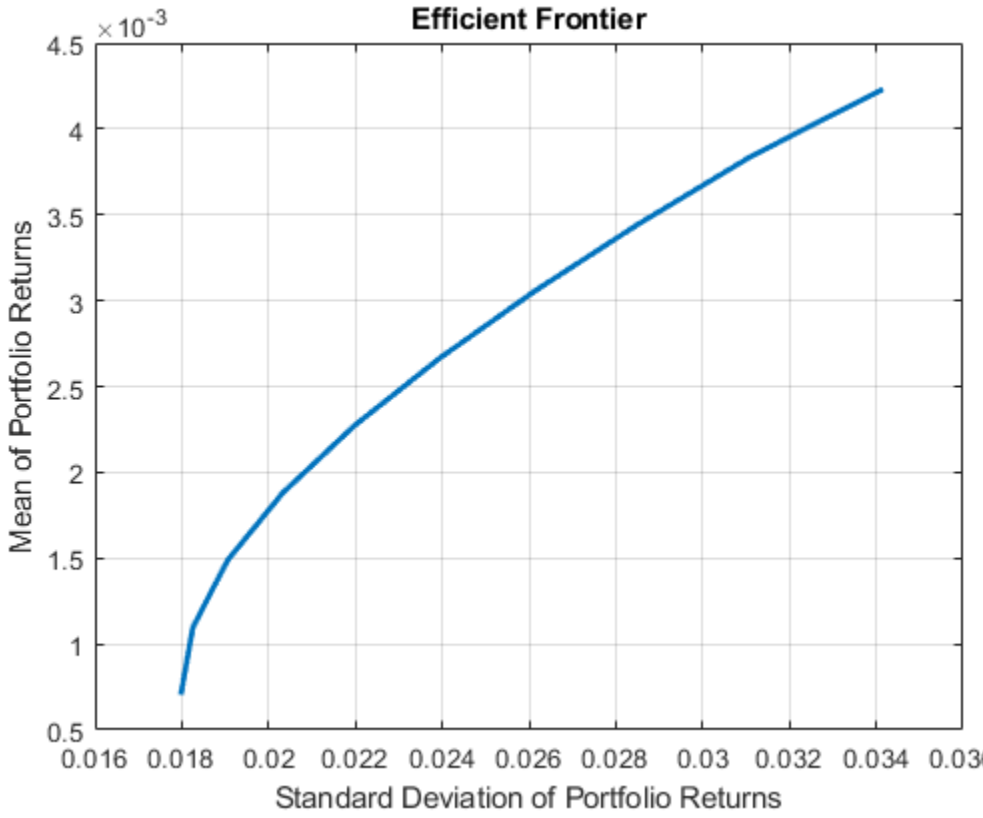
estimateFrontierByReturn	Estimate optimal portfolios with targeted portfolio returns
estimateFrontierByRisk	Estimate optimal portfolios with targeted portfolio risks
estimateFrontierLimits	Estimate optimal portfolios at endpoints of efficient frontier
plotFrontier	Plot efficient frontier
estimateMaxSharpeRatio	Estimate efficient portfolio to maximize Sharpe ratio for Portfolio object
estimatePortMoments	Estimate moments of portfolio returns for Portfolio object
estimatePortReturn	Estimate mean of portfolio returns
estimatePortRisk	Estimate portfolio risk according to risk proxy associated with corresponding object
setSolver	Choose main solver and specify associated solver options for portfolio optimization

Examples

Create a `Portfolio` Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = Portfolio('AssetList',Assets(1:12));
p = estimateAssetMoments(p, Data(:,1:12),'missingdata',true);
p = setDefaultConstraints(p);
plotFrontier(p);
```




```
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1        Port2        Port3       Port4       Port5
    AAPL     0.017926     0.058247     0.097816    0.12955     0
```

```
AMZN            0           0           0          0    0
CSCO            0           0           0          0    0
DELL    0.0041906           0           0          0    0
EBAY            0           0           0          0    0
GOOG      0.16144     0.35678     0.55228    0.75116    1
HPQ      0.052566    0.032302    0.011186          0    0
IBM       0.46422     0.36045     0.25577    0.11928    0
INTC            0           0           0          0    0
MSFT      0.29966     0.19222    0.082949          0    0
ORCL            0           0           0          0    0
YHOO            0           0           0          0    0
```

- "Creating the Portfolio Object" on page 4-28
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

References

[1] For a complete list of references for the Portfolio object, see "Portfolio Optimization" on page A-7.

See Also

`PortfolioCVaR` | `PortfolioMAD`

Topics

"Creating the Portfolio Object" on page 4-28
"Working with Portfolio Constraints Using Defaults" on page 4-67
"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Asset Allocation Case Study" on page 4-175
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

"Portfolio Object Workflow" on page 4-21

Introduced in R2011a

Portfolio

Create Portfolio object for mean-variance portfolio optimization

Use the `Portfolio` function to create a `Portfolio` object for mean-variance portfolio optimization. For more information, see `Portfolio`.

You can use the `Portfolio` function in several ways. To set up a portfolio optimization problem in a Portfolio object, the simplest syntax is:

```
p = Portfolio;
```

This syntax creates a Portfolio object, `p`, such that all object properties are empty.

The `Portfolio` function also accepts collections of argument name-value pair arguments for properties and their values. The `Portfolio` function accepts inputs for properties with the general syntax:

```
p = Portfolio('property1', value1, 'property2', value2, ... );
```

If a Portfolio object exists, the syntax permits the first (and only the first argument) of the `Portfolio` function to be an existing object with subsequent argument name-value pair arguments for properties to be added or modified. For example, given an existing Portfolio object in `p`, the general syntax is:

```
p = Portfolio(p, 'property1', value1, 'property2', value2, ... );
```

Input argument names are not case-sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 18-1304). The `Portfolio` function tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process to formulate a problem. In addition, a Portfolio object is a value object so that, given portfolio `p`, the following code creates two objects, `p` and `q`, that are distinct:

```
q = Portfolio(p, ...)
```

After creating a `Portfolio` object, you can use the associated object functions to set portfolio constraints, analyze the efficient frontier, and validate the portfolio model.

For details on this workflow, see “Portfolio Object Workflow” on page 4-21 and for more detailed information on the theoretical basis for mean-variance optimization, see “Portfolio Optimization Theory” on page 4-3.

Syntax

```
p = Portfolio
p = Portfolio(Name,Value)

p = Portfolio(p,Name,Value)
```

Description

`p = Portfolio` constructs an empty Portfolio object for mean-variance portfolio optimization and analysis. You can then add elements to the Portfolio object using the supported add and set functions. For more information, see “Creating the Portfolio Object” on page 4-28..

`p = Portfolio(Name,Value)` constructs a Portfolio object for mean-variance portfolio optimization and analysis with additional options specified by one or more `Name,Value` arguments.

`p = Portfolio(p,Name,Value)` constructs a Portfolio object for mean-variance portfolio optimization and analysis using a previously constructed Portfolio object `p` with additional options specified by one or more `Name,Value` arguments.

Examples

Create an Empty Portfolio Object

You can create a Portfolio object, `p`, with no input arguments and display it using `disp`.

```
p = Portfolio;
disp(p);
```

```
  Portfolio with properties:
```

```
          BuyCost: []
         SellCost: []
      RiskFreeRate: []
         AssetMean: []
        AssetCovar: []
     TrackingError: []
      TrackingPort: []
          Turnover: []
       BuyTurnover: []
      SellTurnover: []
              Name: []
         NumAssets: []
         AssetList: []
          InitPort: []
       AInequality: []
       bInequality: []
         AEquality: []
         bEquality: []
        LowerBound: []
        UpperBound: []
       LowerBudget: []
       UpperBudget: []
       GroupMatrix: []
        LowerGroup: []
        UpperGroup: []
            GroupA: []
            GroupB: []
       LowerRatio: []
       UpperRatio: []
```

This approach provides a way to set up a portfolio optimization problem with the `Portfolio` function. You can then use the associated set functions to set and modify collections of properties in the `Portfolio` object.

Create a Portfolio Object Using a Single-Step Setup

You can use the `Portfolio` function directly set up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C`.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

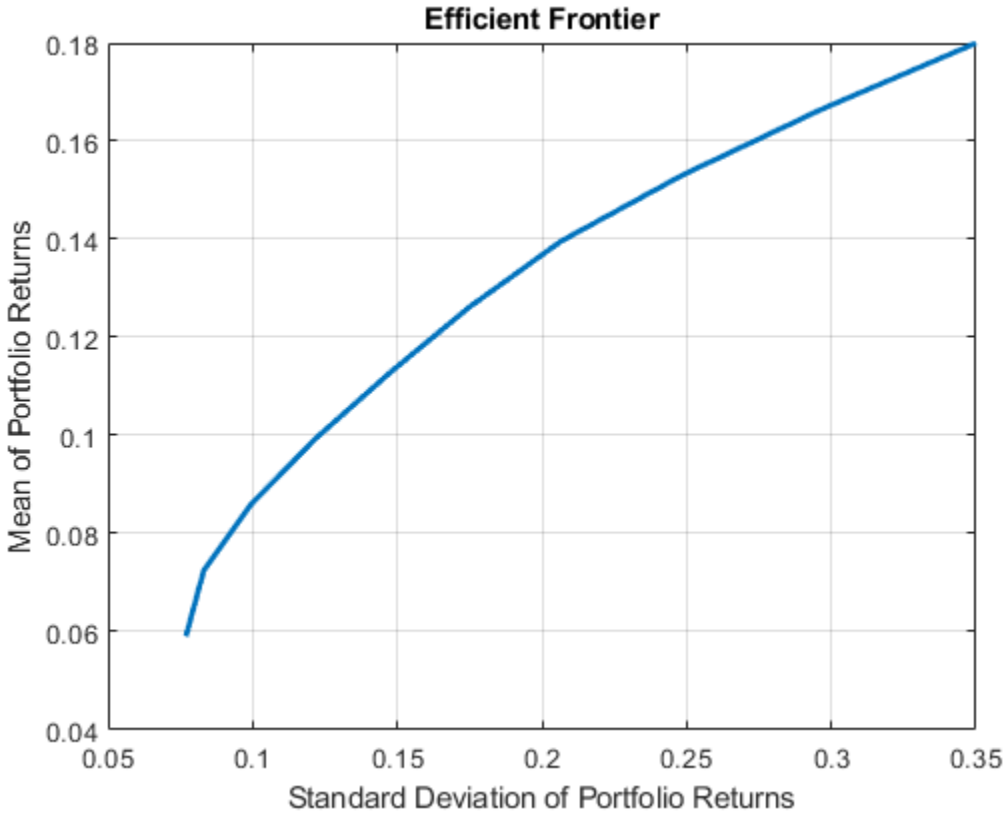
```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio('assetmean', m, 'assetcovar', C, ...
'lowerbudget', 1, 'upperbudget', 1, 'lowerbound', 0)

p =
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [4x1 double]
       AssetCovar: [4x4 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 4
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
```

Note that the `LowerBound` property value undergoes scalar expansion since `AssetMean` and `AssetCovar` provide the dimensions of the problem.

Create a Portfolio Object Using a Sequence of Steps

Using a sequence of steps is an alternative way to accomplish the same task of setting up a "standard" portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C` (which also illustrates that argument names are not case sensitive).

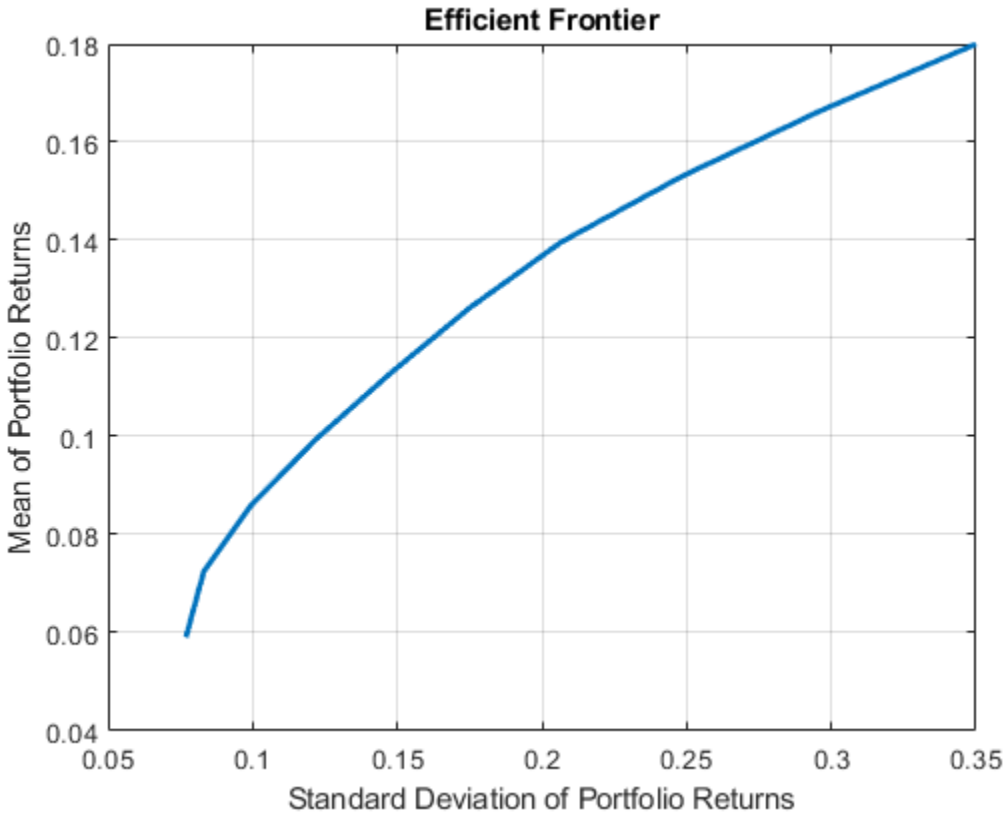
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = Portfolio(p, 'assetmean', m, 'assetcovar', C);
p = Portfolio(p, 'lowerbudget', 1, 'upperbudget', 1);
p = Portfolio(p, 'lowerbound', 0);

plotFrontier(p);
```



This way works because the calls to the `Portfolio` function are in this particular order. In this case, the call to initialize `AssetMean` and `AssetCovar` provides the dimensions for the problem. If you were to do this step last, you would have to explicitly dimension the `LowerBound` property as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p = Portfolio(p, 'LowerBound', zeros(size(m)));
p = Portfolio(p, 'LowerBudget', 1, 'UpperBudget', 1);
```

```
p = Portfolio(p, 'AssetMean', m, 'AssetCovar', C);

plotFrontier(p);
```



If you did not specify the size of `LowerBound` but, instead, input a scalar argument, the `Portfolio` function assumes that you are defining a single-asset problem and produces an error at the call to set asset moments with four assets.

Create a Portfolio Object Using Shortcuts for Property Names

You can create a Portfolio object, `p` with the `Portfolio` function using shortcuts for property names.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio('mean', m, 'covar', C, 'budget', 1, 'lb', 0)
```

```
p =
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [4x1 double]
       AssetCovar: [4x4 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 4
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
```

```
       UpperRatio: []
```

Direct Setting of Portfolio Object Properties

Although not recommended, you can set properties directly, however no error-checking is done on your inputs.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio;
p.NumAssets = numel(m);
p.AssetMean = m;
p.AssetCovar = C;
p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));
disp(p)
```

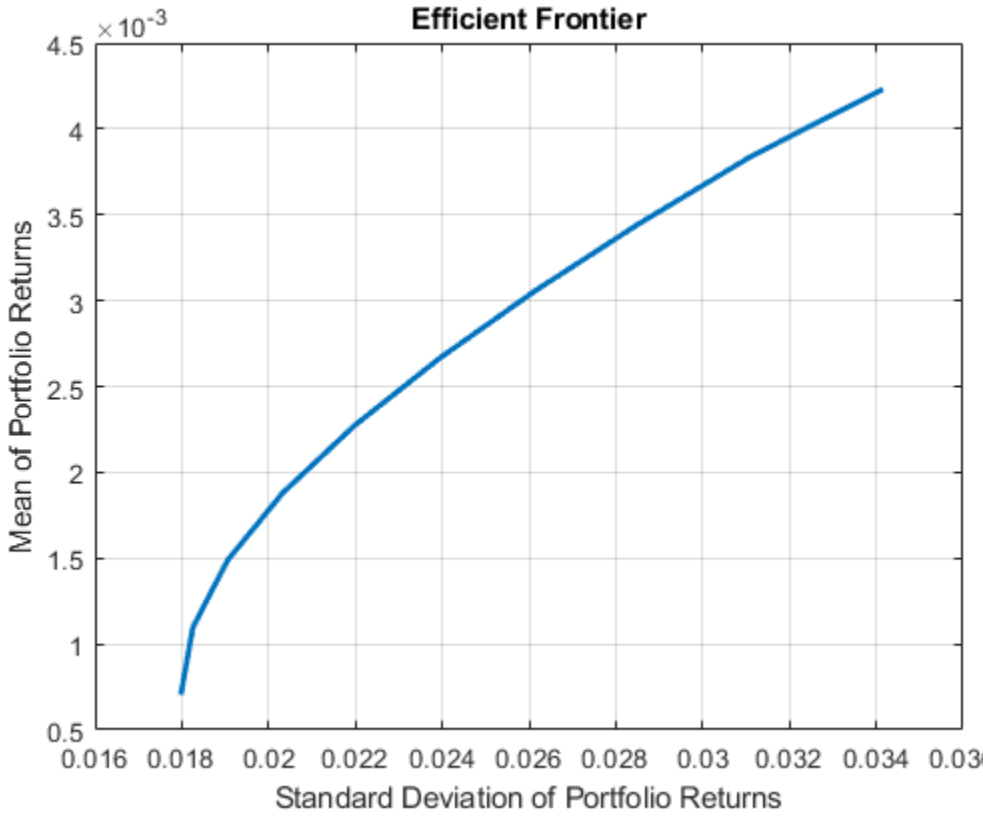
```
  Portfolio with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
       AssetMean: [4x1 double]
      AssetCovar: [4x4 double]
   TrackingError: []
    TrackingPort: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
```

```
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
```

Create a `Portfolio` Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = Portfolio('AssetList',Assets(1:12));
p = estimateAssetMoments(p, Data(:,1:12),'missingdata',true);
p = setDefaultConstraints(p);
plotFrontier(p);
```




```
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1        Port2        Port3       Port4       Port5
    AAPL     0.017926     0.058247     0.097816    0.12955     0
```

```
AMZN            0           0           0           0    0
CSCO            0           0           0           0    0
DELL    0.0041906           0           0           0    0
EBAY            0           0           0           0    0
GOOG      0.16144     0.35678     0.55228     0.75116    1
HPQ      0.052566    0.032302    0.011186           0    0
IBM       0.46422     0.36045     0.25577     0.11928    0
INTC            0           0           0           0    0
MSFT      0.29966     0.19222    0.082949           0    0
ORCL            0           0           0           0    0
YHOO            0           0           0           0    0
```

- "Creating the Portfolio Object" on page 4-28
- "Setting and Getting Properties" on page 4-24
- "Working with Portfolio Constraints Using Defaults" on page 4-67
- "Common Operations on the Portfolio Object" on page 4-36
- "Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-48
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-129
- "Validate the Portfolio Problem for Portfolio Object" on page 4-104
- "Asset Allocation Case Study" on page 4-175
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`p` — Previously constructed Portfolio object

object

Previously constructed `Portfolio` object, specified using the `Portfolio` function

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: p = Portfolio('AssetList',Assets(1:12));

AEquality — Linear equality constraint matrix
[] (default) | matrix

Linear equality constraint matrix, specified as a matrix.

Data Types: double

AInequality — Linear inequality constraint matrix
[] (default) | matrix

Linear inequality constraint matrix, specified as a matrix.

Data Types: double

AssetCovar — Covariance of asset returns
[] (default) | square matrix

Covariance of asset returns, specified as a square matrix.

Data Types: double

AssetList — cell array of character vectors
[] (default) | square matrix

Names or symbols of assets in the universe, specified as a cell array of character vectors.

Data Types: cell

AssetMean — Mean of asset returns
[] (default) | vector

Mean of asset returns, specified as a vector.

Data Types: double

bEquality — Linear equality constraint vector
[] (default) | vector

Linear equality constraint vector, specified as a vector.

Data Types: `double`

`bInequality` — Linear inequality constraint
`[]` (default) | vector

Linear inequality constraint vector, specified as a vector.

Data Types: `double`

`BuyCost` — Proportional purchase costs
`[]` (default) | vector

Proportional purchase costs, specified as a vector.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases
`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar.

Data Types: `double`

`GroupA` — Group A weights to be bounded by weights in group B
`[]` (default) | matrix

Group A weights to be bounded by weights in group B, specified as a matrix.

Data Types: `double`

`GroupB` — Group B weights
`[]` (default) | matrix

Group B weights, specified as a matrix.

Data Types: `double`

`GroupMatrix` — Group membership matrix
`[]` (default) | matrix

Group membership matrix, specified as a matrix.

Data Types: `double`

`InitPort` — Initial portfolio
`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`LowerBound` — Lower-bound constraint
`[]` (default) | vector

Lower-bound constraint, specified as a vector.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint
`[]` (default) | scalar

Lower-bound budget constraint, specified as a scalar.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint
`[]` (default) | vector

Lower-bound group constraint, specified as a vector.

Data Types: `double`

`LowerRatio` — Minimum ratio of allocations between Groups A and B
`[]` (default) | vector

Minimum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object
`[]` (default) | character vector

Name for instance of the Portfolio object, specified as a character vector.

Data Types: `char`

`NumAssets` — Number of assets in the universe
`[]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

`RiskFreeRate` — Risk-free rate
`[]` (default) | scalar

Risk-free rate, specified as a scalar.

Data Types: `double`

`SellCost` — Proportional sales costs
`[]` (default) | vector

Proportional sales costs, specified as a vector.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales
`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar.

Data Types: `double`

`TrackingError` — Upper bound for tracking error constraint
`[]` (default) | scalar

Upper bound for tracking error constraint, specified as a scalar.

Data Types: `double`

`TrackingPort` — Tracking portfolio for tracking error constraint
`[]` (default) | vector

Tracking portfolio for tracking error constraint, specified as a vector.

Data Types: `double`

`Turnover` — Turnover constraint
`[]` (default) | scalar

Turnover constraint, specified as a scalar.

Data Types: `double`

`UpperBound` — Upper-bound constraint
`[]` (default) | vector

Upper-bound constraint, specified as a vector.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint
`[]` (default) | scalar

Upper-bound budget constraint, specified as a scalar.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint
`[]` (default) | vector

Upper-bound group constraint, specified as a vector.

Data Types: `double`

`UpperRatio` — Maximum ratio of allocations between Groups A and B
`[]` (default) | vector

Maximum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

Output Arguments

`p` — Updated Portfolio object
object for portfolio

Updated portfolio object, returned as a `Portfolio`, For more information on using the `Portfolio` object, see `Portfolio`.

Definitions

Mean-Variance Portfolio Optimization

For more information on the theory and definition of mean-variance optimization supported by portfolio optimization tools in Financial Toolbox, see “Portfolio Optimization Theory” on page 4-3.

Portfolio Problem Sufficiency

A mean-variance portfolio optimization is completely specified with the Portfolio object if two condition are met.

The following are the two conditions that must be met:

- The moments of asset returns must be specified such that the property `AssetMean` contains a valid finite mean vector of asset returns and the property `AssetMean` contains a valid symmetric positive-semidefinite matrix for the covariance of asset returns.

  The first condition is satisfied by setting the properties associated with the moments of asset returns.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded.

  The second condition is satisfied by an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed, and several functions, such as `estimateBounds`, provide ways to ensure that your problem is properly formulated.

Although the general sufficiency conditions for mean-variance portfolio optimization go beyond these two conditions, the Portfolio object implemented in Financial Toolbox implicitly handles all these additional conditions. For more information on the Markowitz model for mean-variance portfolio optimization, see “Portfolio Optimization” on page A-7.

Shortcuts for Property Names

The `Portfolio` function has shorter argument names that replace longer argument names associated with specific properties of the Portfolio object.

For example, rather than enter `'assetcovar'`, the `Portfolio` function accepts the case-insensitive name `'covar'` to set the `AssetCovar` property in a Portfolio object. Every shorter argument name corresponds with a single property in the `Portfolio` function. The one exception is the alternative argument name `'budget'`, which signifies both the `LowerBudget` and `UpperBudget` properties. When `'budget'` is used, then the `LowerBudget` and `UpperBudget` properties are set to the same value to form an equality budget constraint.

Shortcuts for Property Names

Shortcut Argument Name	Equivalent Argument / Property Name
`ae`	`AEquality`
`ai`	`AInequality`
`covar`	`AssetCovar`
`assetnames` or `assets`	`AssetList`
`mean`	`AssetMean`
`be`	`bEquality`
`bi`	`bInequality`
`group`	`GroupMatrix`
`lb`	`LowerBound`
`n` or `num`	`NumAssets`
`rfr`	`RiskFreeRate`
`ub`	`UpperBound`
`budget`	`UpperBudget` and `LowerBudget`

References

[1] For a complete list of references for the Portfolio object, see “Portfolio Optimization” on page A-7.

See Also

`estimateFrontier` | `plotFrontier`

Topics

"Creating the Portfolio Object" on page 4-28
"Setting and Getting Properties" on page 4-24
"Working with Portfolio Constraints Using Defaults" on page 4-67
"Common Operations on the Portfolio Object" on page 4-36
"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-48
"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-109
"Estimate Efficient Frontiers for Portfolio Object" on page 4-129
"Validate the Portfolio Problem for Portfolio Object" on page 4-104
"Asset Allocation Case Study" on page 4-175
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3
"Portfolio Object Workflow" on page 4-21

Introduced in R2011a

Portfolio Properties

Manage Portfolio object for mean-variance portfolio optimization and analysis

Description

The main workflow for portfolio optimization is to create an instance of a `Portfolio` object that completely specifies a portfolio optimization problem and to operate on the `Portfolio` object using the supported object functions to obtain and analyze efficient portfolios.

The Portfolio object and its associated functions are an interface for mean-variance portfolio optimization. So, almost everything you do with the Portfolio object can be done using the associated functions. The basic workflow is:

1. Design your portfolio problem.
2. Use the `Portfolio` function to create the Portfolio object or use the various set functions to set up your portfolio problem.
3. Use estimate functions to solve your portfolio problem.

In addition, functions are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a Portfolio object, you can save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of mean-variance portfolio optimization, means that you have either data or moments for asset returns and a collection of constraints on your portfolios, use the `Portfolio` function to set the properties for the Portfolio object. The `Portfolio` function lets you create an object from scratch or update an existing object. Since the Portfolio object is a value object, it is easy to create a basic object, then use functions to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see “Creating the Portfolio Object” on page 4-28.

For more information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-21.

Properties

Setting Up the Object

`AssetList` — Names or symbols of assets in universe

`[]` (default) | cell array of character vectors

Names or symbols of assets in the universe, specified as a cell array of character vectors.

Data Types: `cell`

`InitPort` — Initial portfolio

`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object

`[]` (default) | character vector

Name for instance of the Portfolio object, specified as a character vector.

Data Types: `char`

`NumAssets` — Number of assets in the universe

`[]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

Portfolio Object Constraints

`AEquality` — Linear equality constraint matrix

`[]` (default) | matrix

Linear equality constraint matrix, specified as a matrix.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix

`[]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix.

Data Types: `double`

`bEquality` — Linear equality constraint vector
`[]` (default) | vector

Linear equality constraint vector, specified as a vector.

Data Types: `double`

`bInequality` — Linear inequality constraint
`[]` (default) | vector

Linear inequality constraint vector, specified as a vector.

Data Types: `double`

`GroupA` — Group A weights to be bounded by weights in group B
`[]` (default) | matrix

Group A weights to be bounded by weights in group B, specified as a matrix.

Data Types: `double`

`GroupB` — Group B weights
`[]` (default) | matrix

Group B weights, specified as a matrix.

Data Types: `double`

`GroupMatrix` — Group membership matrix
`[]` (default) | matrix

Group membership matrix, specified as a matrix.

Data Types: `double`

`LowerBound` — Lower-bound constraint
`[]` (default) | vector

Lower-bound constraint, specified as a vector.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint
`[]` (default) | scalar

Lower-bound budget constraint, specified as a scalar.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint
`[]` (default) | vector

Lower-bound group constraint, specified as a vector.

Data Types: `double`

`LowerRatio` — Minimum ratio of allocations between Groups A and B
`[]` (default) | vector

Minimum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

`TrackingError` — Upper bound for tracking error constraint
`[]` (default) | scalar

Upper bound for tracking error constraint, specified as a scalar.

Data Types: `double`

`TrackingPort` — Tracking portfolio for tracking error constraint
`[]` (default) | vector

Tracking portfolio for tracking error constraint, specified as a vector.

Data Types: `double`

`UpperBound` — Upper-bound constraint
`[]` (default) | vector

Upper-bound constraint, specified as a vector.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint
`[]` (default) | scalar

Upper-bound budget constraint, specified as a scalar.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint
`[]` (default) | vector

Upper-bound group constraint, specified as a vector.

Data Types: `double`

`UpperRatio` — Maximum ratio of allocations between Groups A and B
`[]` (default) | vector

Maximum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

Portfolio Object Modeling

`AssetCovar` — Covariance of asset returns
`[]` (default) | square matrix

Covariance of asset returns, specified as a square matrix.

Data Types: `double`

`AssetMean` — Mean of asset returns
`[]` (default) | vector

Mean of asset returns, specified as a vector.

Data Types: `double`

`BuyCost` — Proportional purchase costs
`[]` (default) | vector

Proportional purchase costs, specified as a vector.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases
`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar.

Data Types: `double`

`RiskFreeRate` — Risk-free rate

`[]` (default) | scalar

Risk-free rate, specified as a scalar.

Data Types: `double`

`SellCost` — Proportional sales costs

`[]` (default) | vector

Proportional sales costs, specified as a vector.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales

`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar.

Data Types: `double`

`Turnover` — Turnover constraint

`[]` (default) | scalar

Turnover constraint, specified as a scalar.

Data Types: `double`

See Also

`Portfolio`

Topics

"Creating the Portfolio Object" on page 4-28
"Setting and Getting Properties" on page 4-24
"Working with Portfolio Constraints Using Defaults" on page 4-67
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 5-3
"Portfolio Object Workflow" on page 4-21

PortfolioCVaR

PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis

Description

The PortfolioCVaR object implements conditional value-at-risk (CVaR) portfolio optimization. PortfolioCVaR objects support functions that are specific to conditional value-at-risk (CVaR) portfolio optimization.

The main workflow for CVaR portfolio optimization is to create an instance of a `PortfolioCVaR` object that completely specifies a portfolio optimization problem and to operate on the `PortfolioCVaR` object using supported functions to obtain and analyze efficient portfolios. A CVaR optimization problem is completely specified with the following four elements:

- A universe of assets with scenarios of asset total returns for a period of interest, where scenarios comprise a collection of samples from the underlying probability distribution for asset total returns. This collection must be sufficiently large for asymptotic convergence of sample statistics. Asset return moments and related statistics are derived exclusively from the scenarios.
- A portfolio set that specifies the set of portfolio choices in terms of a collection of constraints.
- A model for portfolio return and risk proxies, which, for CVaR optimization, is either the gross or net mean of portfolio returns and the conditional value-at-risk of portfolio returns.
- A probability level that specifies the probability that a loss is less than or equal to the value-at-risk. Typical values are 0.9 and 0.95, which indicate 10% and 5% loss probabilities.

After these four elements have been specified in an unambiguous way, it is possible to solve and analyze CVaR portfolio optimization problems.

The simplest CVaR portfolio optimization problem has:

- Scenarios of asset total returns

- A requirement that all portfolios have nonnegative weights that sum to 1 (the summation constraint is known as a budget constraint)
- Built-in models for portfolio return and risk proxies that use scenarios of asset total returns
- A probability level of 0.95

Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified by:

```
p = PortfolioCVaR('Scenarios', AssetScenarios, 'LowerBound', 0, 'Budget', 1, ...
'ProbabilityLevel', 0.95);
```

or equivalently by:

```
p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);
```

To confirm that this is a valid portfolio optimization problem, the following function determines whether the set of PortfolioCVaR choices is bounded (a necessary condition for portfolio optimization).

```
[lb, ub, isbounded] = estimateBounds(p);
```

Given the problem specified in the PortfolioCVaR object `p`, the efficient frontier for this problem can be displayed with:

```
plotFrontier(p);
```

and efficient portfolios can be obtained with:

```
pwgt = estimateFrontier(p);
```

For more information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow” on page 5-20 and for more detailed information on the theoretical basis for conditional value-at-risk optimization, see “Portfolio Optimization Theory” on page 5-3.

Creation

To create a `PortfolioCVaR` object, use the `PortfolioCVaR` function. For more details on working with a PortfolioCVaR object, see:

Properties

PortfolioCVaR	Manage Portfolio object for conditional value-at-risk portfolio optimization and analysis

Object Functions

setAssetList	Set up list of identifiers for assets
setInitPort	Set up initial or current portfolio
setDefaultConstraints	Set up portfolio constraints with nonnegative weights that sum to 1
estimateAssetMoments	Estimate mean and covariance of asset returns from data
setCosts	Set up proportional transaction costs
addEquality	Add linear equality constraints for portfolio weights to existing constraints
addGroupRatio	Add group ratio constraints for portfolio weights to existing group ratio constraints
addGroups	Add group constraints for portfolio weights to existing group constraints
addInequality	Add linear inequality constraints for portfolio weights to existing constraints
getBounds	Obtain bounds for portfolio weights from portfolio object
getBudget	Obtain budget constraint bounds from portfolio object

getCosts	Obtain buy and sell transaction costs from portfolio object
getEquality	Obtain equality constraint arrays from portfolio object
getGroupRatio	Obtain group ratio constraint arrays from portfolio object
getGroups	Obtain group constraint arrays from portfolio object
getInequality	Obtain inequality constraint arrays from portfolio object
getOneWayTurnover	Obtain one-way turnover constraints from portfolio object
setGroups	Set up group constraints for portfolio weights
setInequality	Set up linear inequality constraints for portfolio weights
setBounds	Set up bounds for portfolio weights
setBudget	Set up budget constraints
setCosts	Set up proportional transaction costs
setDefaultConstraints	Set up portfolio constraints with nonnegative weights that sum to 1
setEquality	Set up linear equality constraints for portfolio weights
setGroupRatio	Set up group ratio constraints for portfolio weights
setInitPort	Set up initial or current portfolio
setOneWayTurnover	Set up one-way portfolio turnover constraints
setTurnover	Set up maximum portfolio turnover constraint
checkFeasibility	Check feasibility of input portfolios against portfolio object
estimateBounds	Estimate global lower and upper bounds for set of portfolios
estimateFrontier	Estimate specified number of optimal portfolios on the efficient frontier
estimateFrontierByReturn	Estimate optimal portfolios with targeted portfolio returns
estimateFrontierByRisk	Estimate optimal portfolios with targeted portfolio risks
estimateFrontierLimits	Estimate optimal portfolios at endpoints of efficient frontier
plotFrontier	Plot efficient frontier

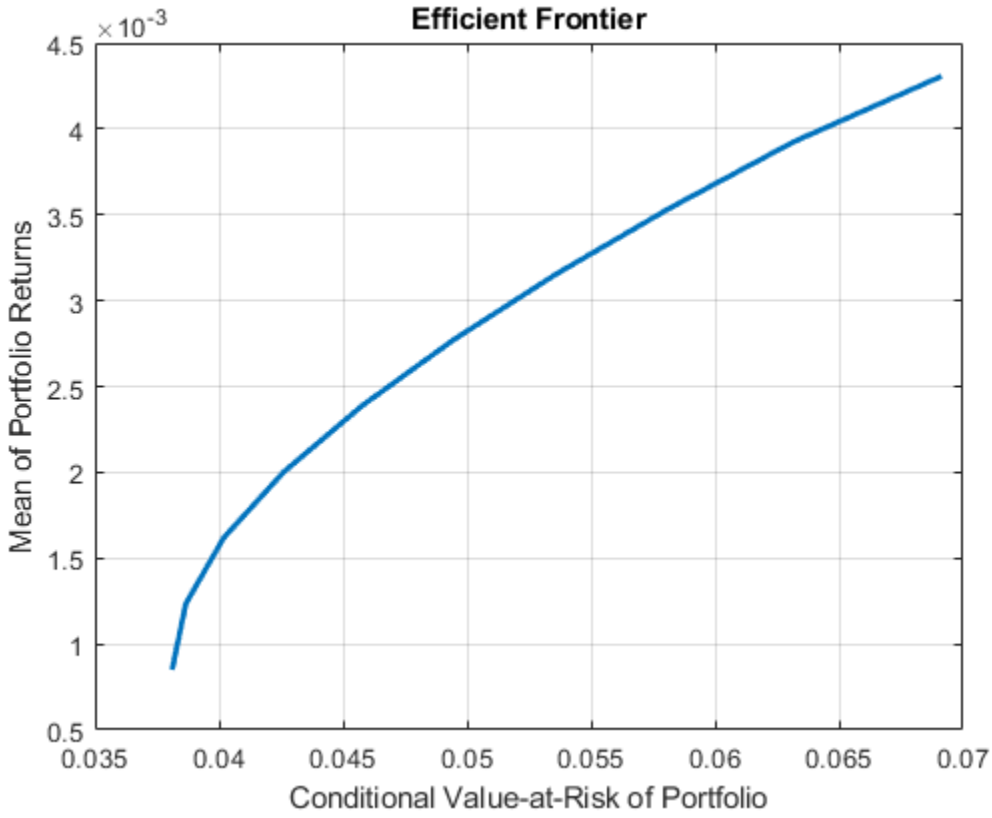
estimatePortReturn	Estimate mean of portfolio returns
estimatePortRisk	Estimate portfolio risk according to risk proxy associated with corresponding object
setSolver	Choose main solver and specify associated solver options for portfolio optimization
setProbabilityLevel	Set probability level for VaR and CVaR calculations
setScenarios	Set asset returns scenarios by direct matrix
getScenarios	Obtain scenarios from portfolio object
simulateNormalScenariosByData	Simulate multivariate normal asset return scenarios from data
simulateNormalScenariosByMoments	Simulate multivariate normal asset return scenarios from mean and covariance of asset returns
estimateScenarioMoments	Estimate mean and covariance of asset return scenarios
estimatePortVaR	Estimate value-at-risk for PortfolioCVaR object
estimatePortStd	

Examples

Construct a PortfolioCVaR Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = PortfolioCVaR('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```




```
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1          Port2          Port3          Port4          Port5
    AAPL      0.010984       0.073246       0.11933        0.13068    1.5092e-14
```

```
AMZN             0             0    3.4661e-23             0    2.8997e-14
CSCO    5.8775e-39             0             0    2.5754e-33    4.1869e-14
DELL      0.022454             0    8.4578e-23    3.0815e-33    3.9048e-14
EBAY             0             0    8.4579e-23    1.2326e-32    1.3394e-15
GOOG       0.20335       0.38055       0.56242       0.75932             1
HPQ       0.041724     0.0099223    4.0953e-23             0    3.8894e-14
IBM        0.44482       0.36453       0.26282          0.11    3.7902e-14
INTC     2.351e-38             0    1.8237e-22             0    3.8264e-14
MSFT       0.27667       0.17175      0.055435             0    4.0873e-14
ORCL             0             0    7.0447e-23             0    3.7811e-14
YHOO             0    1.1755e-38    1.7605e-23    2.6507e-33     3.535e-14
```

- "Creating the PortfolioCVaR Object" on page 5-27
- "Common Operations on the PortfolioCVaR Object" on page 5-36
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Postprocessing Results to Set Up Tradable Portfolios" on page 5-130

References

[1] For a complete list of references for the PortfolioCVaR object, see "Portfolio Optimization" on page A-7.

See Also

`Portfolio` | `PortfolioMAD`

Topics

"Creating the PortfolioCVaR Object" on page 5-27
"Common Operations on the PortfolioCVaR Object" on page 5-36
"Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119

"Postprocessing Results to Set Up Tradable Portfolios" on page 5-130
"Portfolio Optimization Theory" on page 5-3
"PortfolioCVaR Object Workflow" on page 5-20

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

PortfolioCVaR

Create PortfolioCVaR object for conditional value-at-risk portfolio optimization

Use the `PortfolioCVaR` function to create a `PortfolioCVaR` object for conditional value-at-risk portfolio optimization. For more information, see `PortfolioCVaR`

You can use the `PortfolioCVaR` function in several ways. To set up a portfolio optimization problem in a PortfolioCVaR object, the simplest syntax is:

```
p = PortfolioCVaR;
```

This syntax creates a PortfolioCVaR object, `p`, such that all object properties are empty.

The `PortfolioCVaR` function also accepts collections of argument name-value pair arguments for properties and their values. The `PortfolioCVaR` function accepts inputs for properties with the general syntax:

```
p = PortfolioCVaR('property1', value1, 'property2', value2, ... );
```

If a PortfolioCVaR object already exists, the syntax permits the first (and only the first argument) of the `PortfolioCVaR` function to be an existing object with subsequent argument name-value pair arguments for properties to be added or modified. For example, given an existing PortfolioCVaR object in `p`, the general syntax is:

```
p = PortfolioCVaR(p, 'property1', value1, 'property2', value2, ... );
```

Input argument names are not case sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 18-1337). The `PortfolioCVaR` function tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process to formulate a problem. In addition, a PortfolioCVaR object is a value object so that, given portfolio `p`, the following code creates two objects, `p` and `q`, that are distinct:

```
q = PortfolioCVaR(p, ...)
```

After creating a `PortfolioCVaR` object, you can use the associated object functions to set portfolio constraints, analyze the efficient frontier, and validate the portfolio model.

For details on this workflow, see “PortfolioCVaR Object Workflow” on page 5-20 and for more detailed information on the theoretical basis for conditional value-at-risk portfolio optimization, see “Portfolio Optimization Theory” on page 5-3.

Syntax

```
p = PortfolioCVaR
p = PortfolioCVaR(Name,Value)

p = PortfolioCVaR(p,Name,Value)
```

Description

`p = PortfolioCVaR` constructs an empty PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis. You can then add elements to the PortfolioCVaR object using the supported add and set functions. For more information, see “Creating the PortfolioCVaR Object” on page 5-27..

`p = PortfolioCVaR(Name,Value)` constructs a PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis with additional options specified by one or more `Name,Value` arguments.

`p = PortfolioCVaR(p,Name,Value)` constructs a PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis using a previously constructed PortfolioCVaR object `p` with additional options specified by one or more `Name,Value` arguments.

Examples

Create an Empty PortfolioCVaR Object

You can create a PortfolioCVaR object, `p`, with no input arguments and display it using `disp`.

```
p = PortfolioCVaR;
disp(p);
```

```
PortfolioCVaR with properties:

           BuyCost: []
          SellCost: []
      RiskFreeRate: []
  ProbabilityLevel: []
          Turnover: []
       BuyTurnover: []
      SellTurnover: []
      NumScenarios: []
              Name: []
         NumAssets: []
         AssetList: []
          InitPort: []
       AInequality: []
       bInequality: []
         AEquality: []
         bEquality: []
        LowerBound: []
        UpperBound: []
       LowerBudget: []
       UpperBudget: []
       GroupMatrix: []
        LowerGroup: []
        UpperGroup: []
            GroupA: []
            GroupB: []
        LowerRatio: []
        UpperRatio: []
```

This approach provides a way to set up a portfolio optimization problem with the `PortfolioCVaR` function. You can then use the associated set functions to set and modify collections of properties in the `PortfolioCVaR` object.

Create a PortfolioCVaR Object Using a Single-Step Setup

You can use the `PortfolioCVaR` function directly set up a “standard” portfolio optimization problem. Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

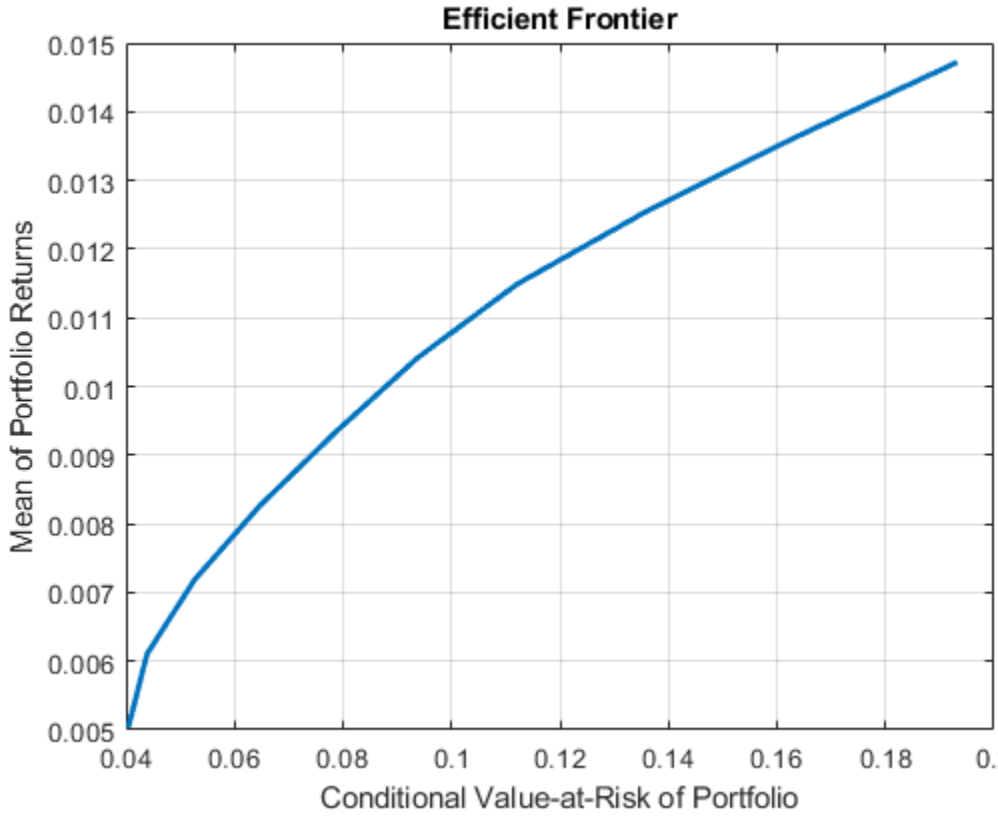
```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('Scenarios', AssetScenarios, ...
'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1, ...
'ProbabilityLevel', 0.95)

p =
  PortfolioCVaR with properties:

            BuyCost: []
           SellCost: []
       RiskFreeRate: []
    ProbabilityLevel: 0.9500
           Turnover: []
        BuyTurnover: []
       SellTurnover: []
       NumScenarios: 20000
               Name: []
          NumAssets: 4
          AssetList: []
           InitPort: []
        AInequality: []
        bInequality: []
          AEquality: []
          bEquality: []
         LowerBound: [4x1 double]
         UpperBound: []
        LowerBudget: 1
        UpperBudget: 1
        GroupMatrix: []
         LowerGroup: []
         UpperGroup: []
             GroupA: []
             GroupB: []
         LowerRatio: []
         UpperRatio: []
```

Note that the `LowerBound` property value undergoes scalar expansion since `AssetScenarios` provides the dimensions of the problem.

Create a PortfolioCVaR Object Using a Sequence of Steps

Using a sequence of steps is an alternative way to accomplish the same task of setting up a "standard" CVaR portfolio optimization problem, given `AssetScenarios` variable is:

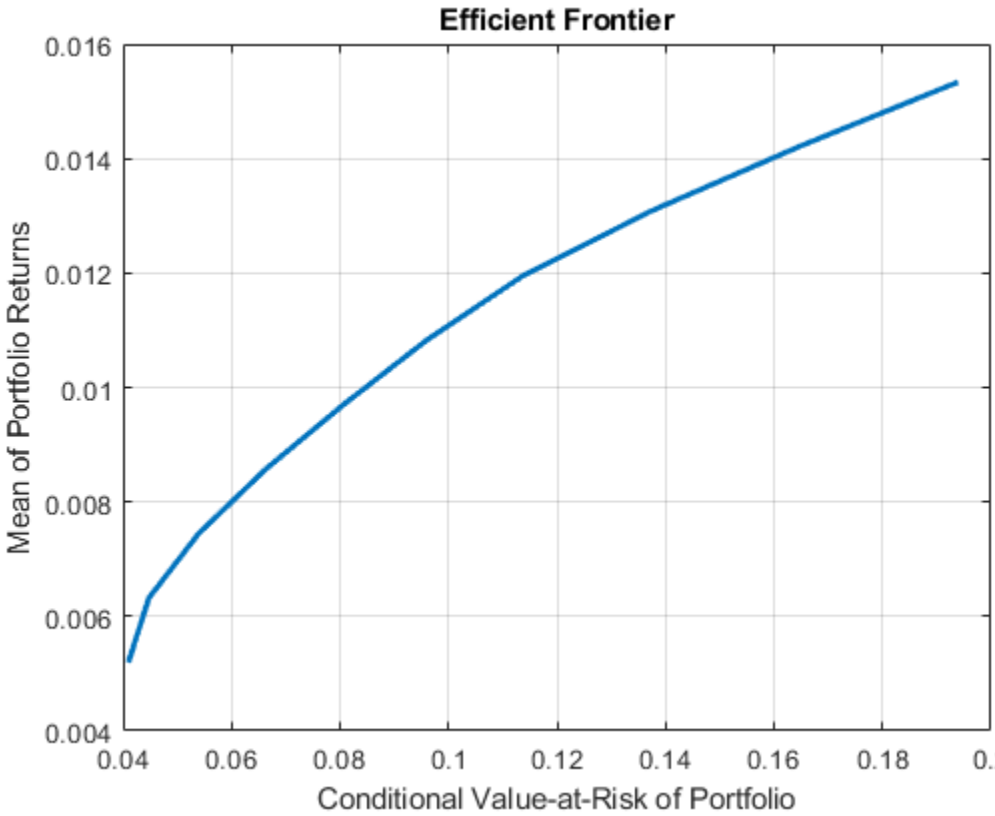
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = PortfolioCVaR(p, 'LowerBound', 0);
p = PortfolioCVaR(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```



This way works because the calls to the `PortfolioCVaR` function are in this particular order. In this case, the call to initialize `AssetScenarios` provides the dimensions for the problem. If you were to do this step last, you would have to explicitly dimension the `LowerBound` property as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;
```

```
AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = PortfolioCVaR(p, 'LowerBound', zeros(size(m)));
p = PortfolioCVaR(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setProbabilityLevel(p, 0.95);
p = setScenarios(p, AssetScenarios);

plotFrontier(p);
```



If you did not specify the size of `LowerBound` but, instead, input a scalar argument, the `PortfolioCVaR` function assumes that you are defining a single-asset problem and produces an error at the call to set asset scenarios with four assets.

Create a PortfolioCVaR Object Using Shortcuts for Property Names

You can create a PortfolioCVaR object, `p` with the `PortfolioCVaR` function using shortcuts for property names.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('scenario', AssetScenarios, 'lb', 0, 'budget', 1, 'plevel', 0.95)
```

```
p =
  PortfolioCVaR with properties:

            BuyCost: []
           SellCost: []
       RiskFreeRate: []
    ProbabilityLevel: 0.9500
           Turnover: []
        BuyTurnover: []
       SellTurnover: []
       NumScenarios: 20000
               Name: []
          NumAssets: 4
          AssetList: []
           InitPort: []
        AInequality: []
        bInequality: []
          AEquality: []
          bEquality: []
         LowerBound: [4x1 double]
         UpperBound: []
        LowerBudget: 1
        UpperBudget: 1
        GroupMatrix: []
         LowerGroup: []
```

```
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
```

Direct Setting of PortfolioCVaR Object Properties

Although not recommended, you can set properties directly, however no error-checking is done on your inputs.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;

p = setScenarios(p, AssetScenarios);
p.ProbabilityLevel = 0.95;

p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));
disp(p)
```

```
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: 0.9500
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: 20000
```

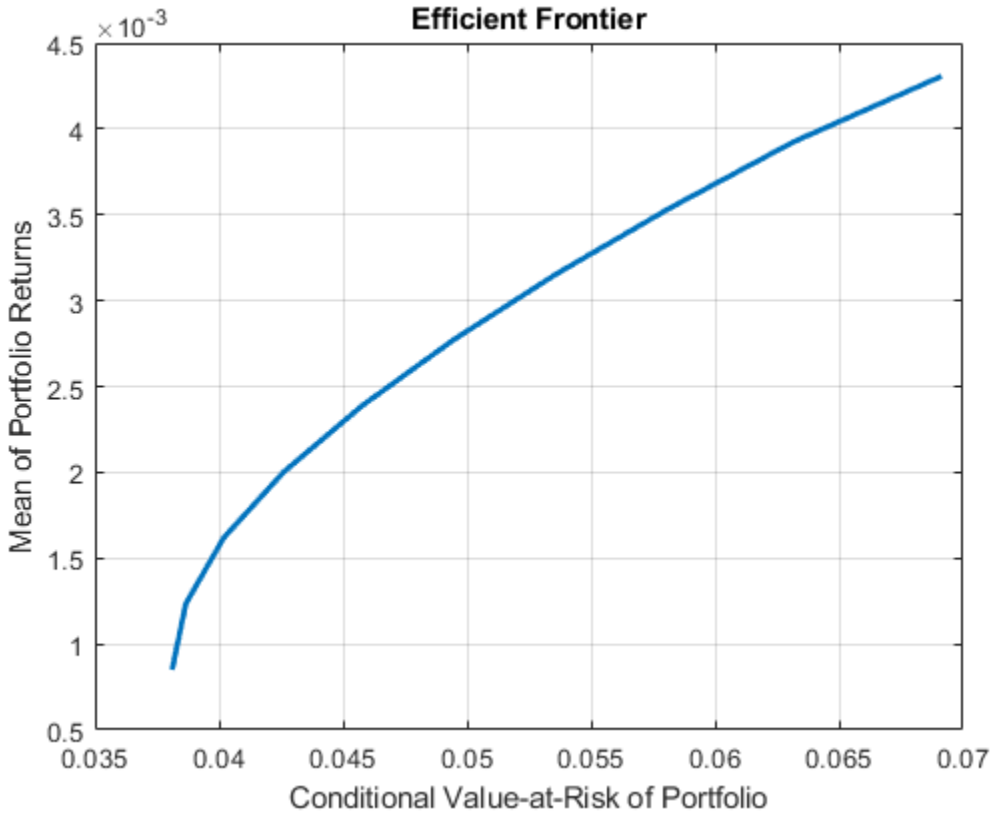
```
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Scenarios cannot be assigned directly to a PortfolioCVaR object. Scenarios must always be set through either the `PortfolioCVaR` function, the `setScenarios` function, or any of the scenario simulation functions.

Construct a PortfolioCVaR Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = PortfolioCVaR('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

plotFrontier(p);
```




```
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1          Port2          Port3          Port4          Port5
    AAPL      0.010984       0.073246       0.11933        0.13068    1.5092e-14
```

```
AMZN             0             0    3.4661e-23             0    2.8997e-14
CSCO    5.8775e-39             0             0    2.5754e-33    4.1869e-14
DELL      0.022454             0    8.4578e-23    3.0815e-33    3.9048e-14
EBAY             0             0    8.4579e-23    1.2326e-32    1.3394e-15
GOOG       0.20335       0.38055       0.56242       0.75932             1
HPQ       0.041724     0.0099223    4.0953e-23             0    3.8894e-14
IBM        0.44482       0.36453       0.26282          0.11    3.7902e-14
INTC     2.351e-38             0    1.8237e-22             0    3.8264e-14
MSFT       0.27667       0.17175      0.055435             0    4.0873e-14
ORCL             0             0    7.0447e-23             0    3.7811e-14
YHOO             0    1.1755e-38    1.7605e-23    2.6507e-33     3.535e-14
```

- "Creating the PortfolioCVaR Object" on page 5-27
- "Common Operations on the PortfolioCVaR Object" on page 5-36
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-101
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-119
- "Postprocessing Results to Set Up Tradable Portfolios" on page 5-130

Input Arguments

`p` — Previously constructed PortfolioCVaR object

object

Previously constructed `PortfolioCVaR` object, specified using the `PortfolioCVaR` function

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = PortfolioCVaR('AssetList',Assets(1:12));`

`AEquality` — Linear equality constraint matrix
`[]` (default) | matrix

Linear equality constraint matrix, specified as a matrix.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix
`[]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix.

Data Types: `double`

`AssetList` — cell array of character vectors
`[]` (default) | square matrix

Names or symbols of assets in the universe, specified as a cell array of character vectors.

Data Types: `cell`

`bEquality` — Linear equality constraint vector
`[]` (default) | vector

Linear equality constraint vector, specified as a vector.

Data Types: `double`

`bInequality` — Linear inequality constraint
`[]` (default) | vector

Linear inequality constraint vector, specified as a vector.

Data Types: `double`

`BuyCost` — Proportional purchase costs
`[]` (default) | vector

Proportional purchase costs, specified as a vector.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases
`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar.

Data Types: `double`

`GroupA` — Group A weights to be bounded by weights in group B
`[]` (default) | matrix

Group A weights to be bounded by weights in group B, specified as a matrix.

Data Types: `double`

`GroupB` — Group B weights
`[]` (default) | matrix

Group B weights, specified as a matrix.

Data Types: `double`

`GroupMatrix` — Group membership matrix
`[]` (default) | matrix

Group membership matrix, specified as a matrix.

Data Types: `double`

`InitPort` — Initial portfolio
`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`LowerBound` — Lower-bound constraint
`[]` (default) | vector

Lower-bound constraint, specified as a vector.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint
`[]` (default) | scalar

Lower-bound budget constraint, specified as a scalar.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint
`[]` (default) | vector

Lower-bound group constraint, specified as a vector.

Data Types: `double`

`LowerRatio` — Minimum ratio of allocations between Groups A and B
`[]` (default) | vector

Minimum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object
`[]` (default) | character vector

Name for instance of the Portfolio object, specified as a character vector.

Data Types: `char`

`NumAssets` — Number of assets in the universe
`[]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

`NumScenarios` — Number of scenarios
`[]` (default) | integer scalar

Number of scenarios, specified as an integer scalar.

Data Types: `double`

`ProbabilityLevel` — Value-at-risk probability level which is 1 − (loss probability)
`[]` (default) | scalar

Value-at-risk probability level which is 1 − (loss probability), specified as a scalar.

Data Types: `double`

`RiskFreeRate` — Risk-free rate
`[]` (default) | scalar

Risk-free rate, specified as a scalar.

Data Types: `double`

`SellCost` — Proportional sales costs

`[]` (default) | vector

Proportional sales costs, specified as a vector.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales

`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar.

Data Types: `double`

`Turnover` — Turnover constraint

`[]` (default) | scalar

Turnover constraint, specified as a scalar.

Data Types: `double`

`UpperBound` — Upper-bound constraint

`[]` (default) | vector

Upper-bound constraint, specified as a vector.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint

`[]` (default) | scalar

Upper-bound budget constraint, specified as a scalar.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint

`[]` (default) | vector

Upper-bound group constraint, specified as a vector.

Data Types: `double`

`UpperRatio` — Maximum ratio of allocations between Groups A and B
`[]` (default) | vector

Maximum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

Output Arguments

`p` — Updated PortfolioCVaR object
object for CVaR portfolio

Updated CVaR portfolio object, returned as a `PortfolioCVaR` object, For more information on using the `PortfolioCVaR` object, see `PortfolioCVaR`.

Definitions

Conditional Value-at-Risk Portfolio Optimization

For more information on the theory and definition of conditional value-at-risk optimization supported by portfolio optimization tools in Financial Toolbox, see “Portfolio Optimization Theory” on page 5-3.

PortfolioCVaR Problem Sufficiency

A CVaR portfolio optimization problem is completely specified with the PortfolioCVaR object if three conditions are met.

The following are the three conditions that must be met:

- You must specify a collection of asset returns or prices known as scenarios such that all scenarios are finite asset returns or prices. These scenarios are meant to be samples from the underlying probability distribution of asset returns. This condition can be satisfied by the `setScenarios` function or with several canned scenario simulation functions.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded. You can satisfy this condition using an extensive collection of

properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be imposed and several tools, such as the `estimateBounds` function, provide ways to ensure that your problem is properly formulated.

- You must specify a probability level to locate the level of tail loss above which the conditional value-at-risk is to be minimized. This condition can be satisfied by the `setProbabilityLevel` function.

  Although the general sufficient conditions for CVaR portfolio optimization go beyond the first three conditions, the PortfolioCVaR object handles all these additional conditions.

Shortcuts for Property Names

The `PortfolioCVaR` function has shorter argument names that replace longer argument names associated with specific properties of the PortfolioCVaR object.

For example, rather than enter `'ProbabilityLevel'`, the `PortfolioCVaR` function accepts the case-insensitive name `'plevel'` to set the `ProbabilityLevel` property in a PortfolioCVaR object. Every shorter argument name corresponds with a single property in the `PortfolioCVaR` function. The one exception is the alternative argument name `'budget'`, which signifies both the `LowerBudget` and `UpperBudget` properties. When `'budget'` is used, then the `LowerBudget` and `UpperBudget` properties are set to the same value to form an equality budget constraint.

Shortcuts for Property Names

Shortcut Argument Name	Equivalent Argument / Property Name
`ae`	`AEquality`
`ai`	`AInequality`
`assetnames` or `assets`	`AssetList`
`be`	`bEquality`
`bi`	`bInequality`
`budget`	`UpperBudget` and `LowerBudget`
`group`	`GroupMatrix`
`lb`	`LowerBound`
`n` or `num`	`NumAssets`
`level`, `problevel`, or `plevel`	`ProbabilityLevel`
`rfr`	`RiskFreeRate`
`scenario` or `assetscenarios`	`Scenarios`
`ub`	`UpperBound`

References

[1] For a complete list of references for the PortfolioCVaR object, see “Portfolio Optimization” on page A-7.

See Also

`estimateFrontier` | `plotFrontier` | `setScenarios`

Topics

“Creating the PortfolioCVaR Object” on page 5-27
“Common Operations on the PortfolioCVaR Object” on page 5-36
“Working with CVaR Portfolio Constraints Using Defaults” on page 5-63
“Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
“Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-101
“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-119

"Postprocessing Results to Set Up Tradable Portfolios" on page 5-130
"Portfolio Optimization Theory" on page 5-3
"PortfolioCVaR Object Workflow" on page 5-20

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

PortfolioCVaR Properties

Manage Portfolio object for conditional value-at-risk portfolio optimization and analysis

Description

The main workflow for CVaR portfolio optimization is to create an instance of a `PortfolioCVaR` object that completely specifies a conditional value-at-risk optimization problem and to operate on the `PortfolioCVaR` object using the supported object functions to obtain and analyze efficient portfolios.

The PortfolioCVaR object and its associated functions are an interface for conditional value-at-risk optimization. So, almost everything you do with the PortfolioCVaR object can be done using the associated functions. The basic workflow is:

1. Design your portfolio problem.
2. Use the `PortfolioCVaR` function to create the PortfolioCVaR object or use the various set functions to set up your portfolio problem.
3. Use estimate functions to solve your portfolio problem.

In addition, functions are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a PortfolioCVaR object, you can save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of CVaR portfolio optimization, means that you have either scenarios, data, or moments for asset returns, a probability level, and a collection of constraints on your portfolios, use the `PortfolioCVaR` function to set the properties for the PortfolioCVaR object.

The `PortfolioCVaR` function lets you create an object from scratch or update an existing object. Since the PortfolioCVaR object is a value object, it is easy to create a basic object, then use methods to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see “Creating the PortfolioCVaR Object” on page 5-27

For more information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow” on page 5-20.

Properties

Setting Up the Object

`AssetList` — Names or symbols of assets in universe

`[]` (default) | cell array of character vectors

Names or symbols of assets in the universe, specified as a cell array of character vectors.

Data Types: `cell`

`InitPort` — Initial portfolio

`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object

`[]` (default) | character vector

Name for instance of the Portfolio object, specified as a character vector.

Data Types: `char`

`NumAssets` — Number of assets in the universe

`[]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

Portfolio Object Constraints

`AEquality` — Linear equality constraint matrix

`[]` (default) | matrix

Linear equality constraint matrix, specified as a matrix.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix

`[]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix.

Data Types: `double`

`bEquality` — Linear equality constraint vector
`[]` (default) | vector

Linear equality constraint vector, specified as a vector.

Data Types: `double`

`bInequality` — Linear inequality constraint
`[]` (default) | vector

Linear inequality constraint vector, specified as a vector.

Data Types: `double`

`GroupA` — Group A weights to be bounded by weights in group B
`[]` (default) | matrix

Group A weights to be bounded by weights in group B, specified as a matrix.

Data Types: `double`

`GroupB` — Group B weights
`[]` (default) | matrix

Group B weights, specified as a matrix.

Data Types: `double`

`GroupMatrix` — Group membership matrix
`[]` (default) | matrix

Group membership matrix, specified as a matrix.

Data Types: `double`

`LowerBound` — Lower-bound constraint
`[]` (default) | vector

Lower-bound constraint, specified as a vector.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint
`[]` (default) | scalar

Lower-bound budget constraint, specified as a scalar.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint
`[]` (default) | vector

Lower-bound group constraint, specified as a vector.

Data Types: `double`

`LowerRatio` — Minimum ratio of allocations between Groups A and B
`[]` (default) | vector

Minimum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

`UpperBound` — Upper-bound constraint
`[]` (default) | vector

Upper-bound constraint, specified as a vector.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint
`[]` (default) | scalar

Upper-bound budget constraint, specified as a scalar.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint
`[]` (default) | vector

Upper-bound group constraint, specified as a vector.

Data Types: `double`

`UpperRatio` — Maximum ratio of allocations between Groups A and B
`[]` (default) | vector

Maximum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

Portfolio Object Modeling

`BuyCost` — Proportional purchase costs
`[]` (default) | vector

Proportional purchase costs, specified as a vector.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases
`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar.

Data Types: `double`

`RiskFreeRate` — Risk-free rate
`[]` (default) | scalar

Risk-free rate, specified as a scalar.

Data Types: `double`

`ProbabilityLevel` — Value-at-risk probability level which is 1 − (loss probability)
`[]` (default) | scalar

Value-at-risk probability level which is 1 − (loss probability), specified as a scalar.

Data Types: `double`

`NumScenarios` — Number of scenarios
`[]` (default) | integer scalar

Number of scenarios, specified as an integer scalar.

Data Types: `double`

`SellCost` — Proportional sales costs
`[]` (default) | vector

Proportional sales costs, specified as a vector.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales

`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar.

Data Types: `double`

`Turnover` — Turnover constraint

`[]` (default) | scalar

Turnover constraint, specified as a scalar.

Data Types: `double`

See Also

`PortfolioCVaR`

Topics

"Creating the PortfolioCVaR Object" on page 5-27
"Setting and Getting Properties" on page 5-23
"Working with CVaR Portfolio Constraints Using Defaults" on page 5-63
"Portfolio Optimization Theory" on page 5-3
"PortfolioCVaR Object Workflow" on page 5-20

External Websites

CVaR Portfolio Optimization (5 min 33 sec)

PortfolioMAD

PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis

Description

The PortfolioMAD object implements mean-absolute deviation portfolio optimization, where MAD stands for “mean-absolute deviation.” PortfolioMAD objects support functions that are specific to MAD portfolio optimization.

The main workflow for MAD portfolio optimization is to create an instance of a PortfolioMAD object that completely specifies a portfolio optimization problem and to operate on the PortfolioMAD object to obtain and analyze efficient portfolios. A MAD optimization problem is completely specified with these three elements:

- A universe of assets with scenarios of asset total returns for a period of interest, where scenarios comprise a collection of samples from the underlying probability distribution for asset total returns. This collection must be sufficiently large for asymptotic convergence of sample statistics. Asset return moments and related statistics are derived exclusively from the scenarios.
- A portfolio set that specifies the set of portfolio choices in terms of a collection of constraints.
- A model for portfolio return and risk proxies, which, for MAD optimization, is either the gross or net mean of portfolio returns and the mean-absolute deviation of portfolio returns.

After these three elements have been specified unambiguously, it is possible to solve and analyze MAD portfolio optimization problems.

The simplest MAD portfolio optimization problem has:

- Scenarios of asset total returns
- A requirement that all portfolios have nonnegative weights that sum to 1 (the summation constraint is known as a budget constraint)
- Built-in models for portfolio return and risk proxies that use scenarios of asset total returns

Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified by:

```
p = PortfolioMAD('Scenarios', AssetScenarios, 'LowerBound', 0, 'Budget', 1);
```

or equivalently by:

```
p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
```

To confirm that this is a valid portfolio optimization problem, the following function determines whether the set of PortfolioMAD choices is bounded (a necessary condition for portfolio optimization).

```
[lb, ub, isbounded] = estimateBounds(p);
```

Given the problem specified in the PortfolioMAD object `p`, the efficient frontier for this problem can be displayed with:

```
plotFrontier(p);
```

and efficient portfolios can be obtained with:

```
pwgt = estimateFrontier(p);
```

For more information on the workflow when using PortfolioMAD objects, see“PortfolioMAD Object Workflow” on page 6-19 and for more detailed information on the theoretical basis for mean-absolute deviation optimization, see “Portfolio Optimization Theory” on page 6-3.

Creation

To create a `PortfolioMAD` object, use the `PortfolioMAD` function. For more details on working with a PortfolioMAD object, see:

- “PortfolioMAD Object Properties and Functions” on page 6-21
- “Working with PortfolioMAD Objects” on page 6-21
- “Setting and Getting Properties” on page 6-22
- “Displaying PortfolioMAD Objects” on page 6-23
- “Saving and Loading PortfolioMAD Objects” on page 6-23

- "Estimating Efficient Portfolios and Frontiers" on page 6-23
- "Arrays of PortfolioMAD Objects" on page 6-23
- "Subclassing PortfolioMAD Objects" on page 6-24
- "Conventions for Representation of Data" on page 6-24

Properties

PortfolioMAD	Manage PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis

Object Functions

setAssetList	Set up list of identifiers for assets
setInitPort	Set up initial or current portfolio
setDefaultConstraints	Set up portfolio constraints with nonnegative weights that sum to 1
estimateAssetMoments	Estimate mean and covariance of asset returns from data
setCosts	Set up proportional transaction costs
addEquality	Add linear equality constraints for portfolio weights to existing constraints
addGroupRatio	Add group ratio constraints for portfolio weights to existing group ratio constraints
addGroups	Add group constraints for portfolio weights to existing group constraints
addInequality	Add linear inequality constraints for portfolio weights to existing constraints
getBounds	Obtain bounds for portfolio weights from portfolio object
getBudget	Obtain budget constraint bounds from portfolio object
getCosts	Obtain buy and sell transaction costs from portfolio object
getEquality	Obtain equality constraint arrays from portfolio object
getGroupRatio	Obtain group ratio constraint arrays from portfolio object

getGroups	Obtain group constraint arrays from portfolio object
getInequality	Obtain inequality constraint arrays from portfolio object
getOneWayTurnover	Obtain one-way turnover constraints from portfolio object
setGroups	Set up group constraints for portfolio weights
setInequality	Set up linear inequality constraints for portfolio weights
setBounds	Set up bounds for portfolio weights
setBudget	Set up budget constraints
setCosts	Set up proportional transaction costs
setDefaultConstraints	Set up portfolio constraints with nonnegative weights that sum to 1
setEquality	Set up linear equality constraints for portfolio weights
setGroupRatio	Set up group ratio constraints for portfolio weights
setInitPort	Set up initial or current portfolio
setOneWayTurnover	Set up one-way portfolio turnover constraints
setTurnover	Set up maximum portfolio turnover constraint
checkFeasibility	Check feasibility of input portfolios against portfolio object
estimateBounds	Estimate global lower and upper bounds for set of portfolios
estimateFrontier	Estimate specified number of optimal portfolios on the efficient frontier
estimateFrontierByReturn	Estimate optimal portfolios with targeted portfolio returns
estimateFrontierByRisk	Estimate optimal portfolios with targeted portfolio risks
estimateFrontierLimits	Estimate optimal portfolios at endpoints of efficient frontier
plotFrontier	Plot efficient frontier
estimatePortReturn	Estimate mean of portfolio returns
estimatePortRisk	Estimate portfolio risk according to risk proxy associated with corresponding object
setSolver	Choose main solver and specify associated solver options for portfolio optimization

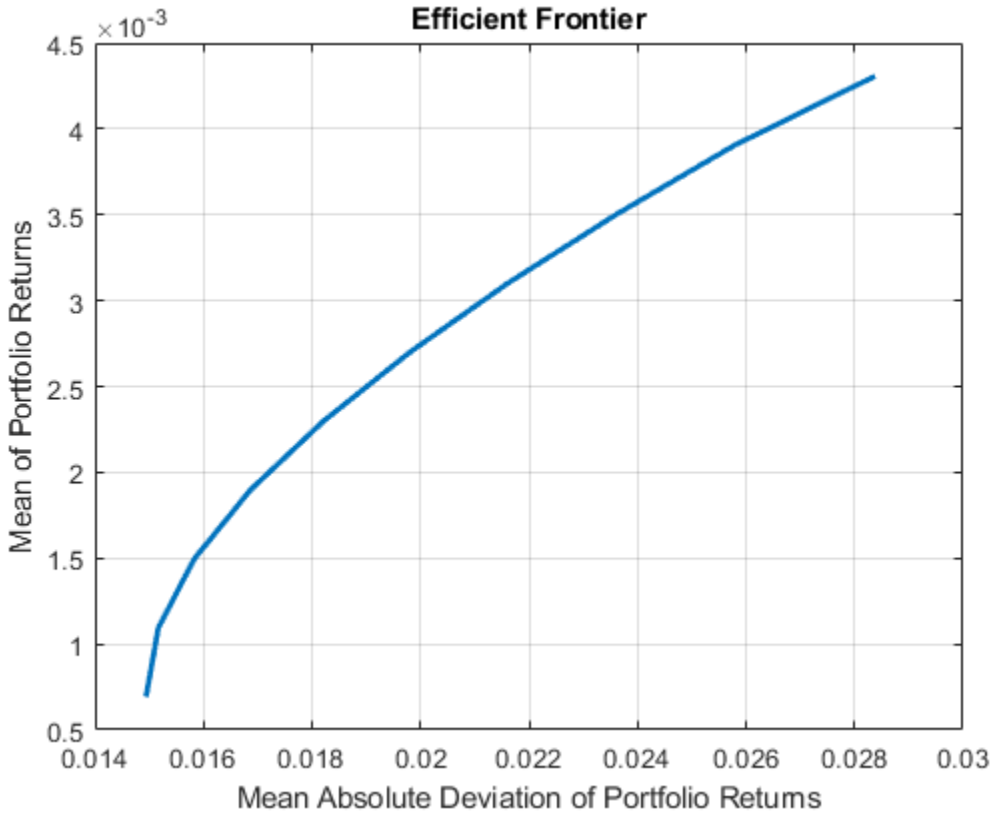
setProbabilityLevel	Set probability level for VaR and CVaR calculations
setScenarios	Set asset returns scenarios by direct matrix
getScenarios	Obtain scenarios from portfolio object
simulateNormalScenariosByData	Simulate multivariate normal asset return scenarios from data
simulateNormalScenariosByMoments	Simulate multivariate normal asset return scenarios from mean and covariance of asset returns
estimateScenarioMoments	Estimate mean and covariance of asset return scenarios
estimatePortStd	Estimate standard deviation of portfolio returns

Examples

Create a `PortfolioMAD` Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = PortfolioMAD('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
p = setDefaultConstraints(p);

plotFrontier(p);
```




```
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1          Port2          Port3          Port4          Port5
    AAPL     0.030236       0.075387       0.11278        0.13456    1.5092e-14
```

```
AMZN    1.6541e-21    1.2618e-22    8.6501e-23     4.6635e-18    2.8997e-14
CSCO    1.5007e-22    7.7933e-23    3.6225e-23     1.1556e-33    4.1869e-14
DELL     0.0089659             0             0     2.5995e-17    3.9048e-14
EBAY    2.0446e-22             0    6.6122e-24     6.4047e-17    1.3394e-15
GOOG       0.16117       0.35201       0.54486        0.74888             1
HPQ       0.056551      0.024037             0    -3.6588e-33    3.8894e-14
IBM        0.45905       0.37891       0.29383        0.11656    3.7902e-14
INTC    -4.702e-38    6.0531e-22             0     1.7947e-17    3.8264e-14
MSFT       0.28403       0.16966      0.048527     8.3535e-18    4.0873e-14
ORCL    5.3466e-21             0    2.5731e-22     3.7312e-18    3.7811e-14
YHOO             0    3.5531e-23             0     1.4482e-33     3.535e-14
```

- “Creating the PortfolioMAD Object” on page 6-26
- “Common Operations on the PortfolioMAD Object” on page 6-34
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-61
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
- “Validate the MAD Portfolio Problem” on page 6-91
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
- “Postprocessing Results to Set Up Tradable Portfolios” on page 6-122

References

[1] For a complete list of references for the PortfolioMAD object, see “Portfolio Optimization” on page A-7.

See Also

`Portfolio` | `PortfolioCVaR`

Topics

“Creating the PortfolioMAD Object” on page 6-26
“Common Operations on the PortfolioMAD Object” on page 6-34
“Working with MAD Portfolio Constraints Using Defaults” on page 6-61
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
“Validate the MAD Portfolio Problem” on page 6-91

"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
"Postprocessing Results to Set Up Tradable Portfolios" on page 6-122
"Portfolio Optimization Theory" on page 6-3
"PortfolioMAD Object Workflow" on page 6-19

Introduced in R2013b

PortfolioMAD

Create PortfolioMAD object for mean-absolute deviation portfolio optimization

Use the `PortfolioMAD` function to create a `PortfolioMAD` object for mean-absolute deviation portfolio optimization. For more information, see `PortfolioMAD`.

You can use the `PortfolioMAD` function in several ways. To set up a portfolio optimization problem in a PortfolioMAD object, the simplest syntax is:

```
p = PortfolioMAD;
```

This syntax creates a PortfolioMAD object, `p`, such that all object properties are empty.

The `PortfolioMAD` function also accepts collections of argument name-value pair arguments for properties and their values. The `PortfolioMAD` function accepts inputs for properties with the general syntax:

```
p = PortfolioMAD('property1', value1, 'property2', value2, ... );
```

If a PortfolioMAD object exists, the syntax permits the first (and only the first argument) of the `PortfolioMAD` function to be an existing object with subsequent argument name-value pair arguments for properties to be added or modified. For example, given an existing PortfolioMAD object in `p`, the general syntax is:

```
p = PortfolioMAD(p, 'property1', value1, 'property2', value2, ... );
```

Input argument names are not case-sensitive, but must be completely specified. In addition, several properties can be specified with alternative argument names (see “Shortcuts for Property Names” on page 18-1371). The `PortfolioMAD` function tries to detect problem dimensions from the inputs and, once set, subsequent inputs can undergo various scalar or matrix expansion operations that simplify the overall process to formulate a problem. In addition, a PortfolioMAD object is a value object so that, given portfolio `p`, the following code creates two objects, `p` and `q`, that are distinct:

```
q = PortfolioMAD(p, ...)
```

After creating a `PortfolioMAD` object, you can use the associated object functions to set portfolio constraints, analyze the efficient frontier, and validate the portfolio model.

For details on this workflow, see"PortfolioMAD Object Workflow" on page 6-19 and for more detailed information on the theoretical basis for conditional value-at-risk portfolio optimization, see "Portfolio Optimization Theory" on page 6-3.

Syntax

```
p = PortfolioMAD
p = PortfolioMAD(Name,Value)

p = PortfolioMAD(p,Name,Value)
```

Description

`p = PortfolioMAD` constructs an empty PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis. You can then add elements to the PortfolioMAD object using the supported add and set functions. For more information, see "Creating the PortfolioMAD Object" on page 6-26.

`p = PortfolioMAD(Name,Value)` constructs a PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis with additional options specified by one or more `Name,Value` arguments.

`p = PortfolioMAD(p,Name,Value)` constructs a PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis using a previously constructed PortfolioMAD object `p` with additional options specified by one or more `Name,Value` arguments.

Examples

Create an Empty PortfolioMAD Object

You can create a PortfolioMAD object, `p`, with no input arguments and display it using `disp`.

```
p = PortfolioMAD;
disp(p);
```

```
PortfolioMAD with properties:

       BuyCost: []
      SellCost: []
  RiskFreeRate: []
      Turnover: []
   BuyTurnover: []
  SellTurnover: []
  NumScenarios: []
          Name: []
     NumAssets: []
     AssetList: []
      InitPort: []
   AInequality: []
   bInequality: []
     AEquality: []
     bEquality: []
    LowerBound: []
    UpperBound: []
   LowerBudget: []
   UpperBudget: []
   GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
        GroupA: []
        GroupB: []
    LowerRatio: []
    UpperRatio: []
```

This approach provides a way to set up a portfolio optimization problem with the `PortfolioMAD` function. You can then use the associated set functions to set and modify collections of properties in the `PortfolioMAD` object.

Create a PortfolioMAD Object Using a Single-Step Setup

You can use the `PortfolioMAD` function directly set up a “standard” portfolio optimization problem. Given scenarios of asset returns in the variable `AssetScenarios`, this problem is completely specified as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
```

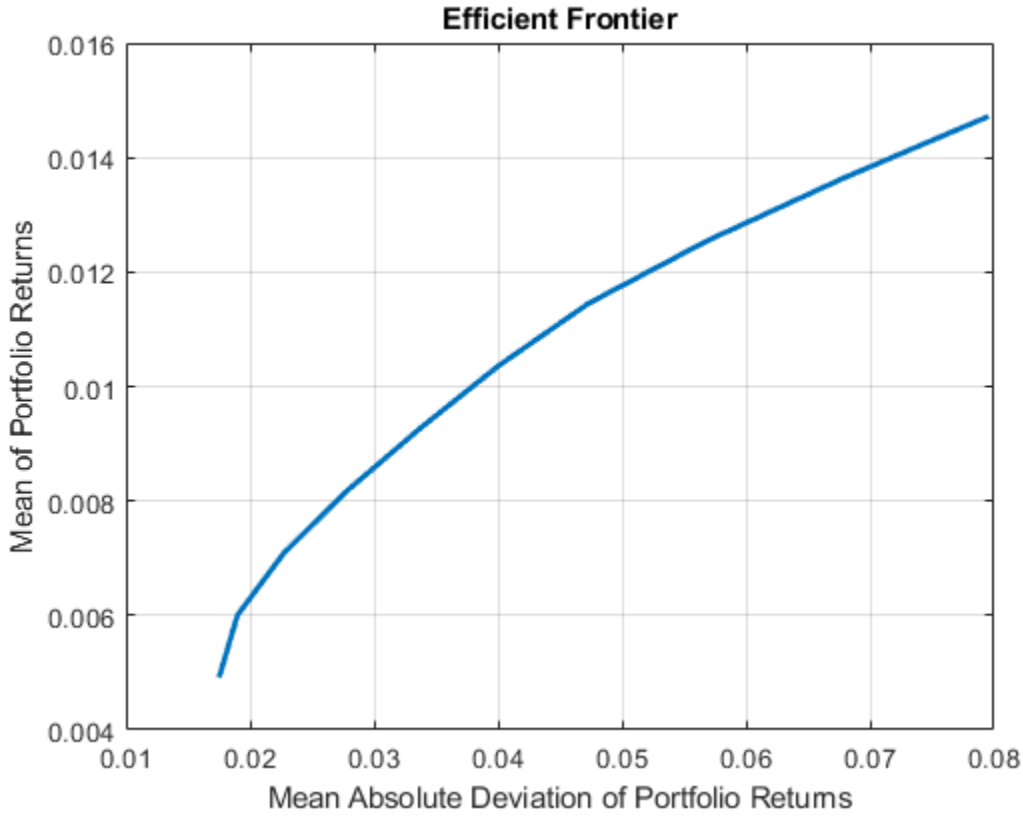
```
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('Scenarios', AssetScenarios, ...
'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1)

p =
  PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 20000
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Note that the `LowerBound` property value undergoes scalar expansion since `AssetScenarios` provides the dimensions of the problem.

Create a PortfolioMAD Object Using a Sequence of Steps

Using a sequence of steps is an alternative way to accomplish the same task of setting up a "standard" MAD portfolio optimization problem, given `AssetScenarios` variable is:

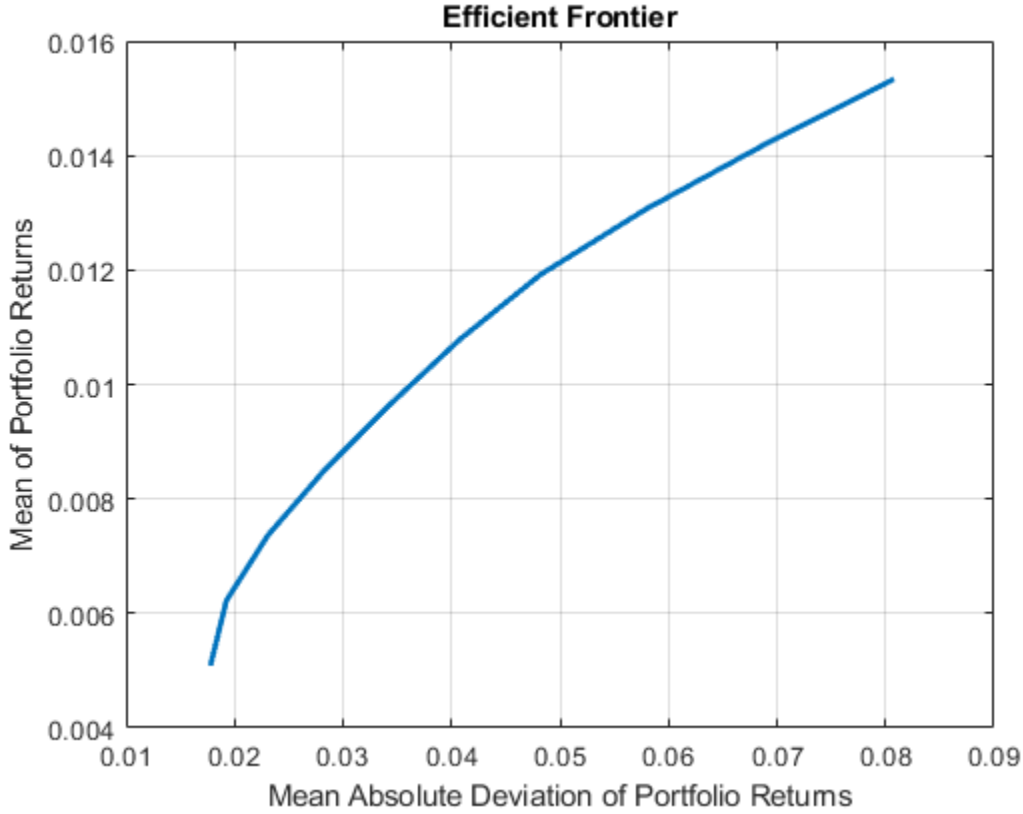
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = PortfolioMAD(p, 'LowerBound', 0);
p = PortfolioMAD(p, 'LowerBudget', 1, 'UpperBudget', 1);

plotFrontier(p);
```



This way works because the calls to the `PortfolioMAD` function are in this particular order. In this case, the call to initialize `AssetScenarios` provides the dimensions for the problem. If you were to do this step last, you would have to explicitly dimension the `LowerBound` property as follows:

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;
```

```
AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = PortfolioMAD(p, 'LowerBound', zeros(size(m)));
p = PortfolioMAD(p, 'LowerBudget', 1, 'UpperBudget', 1);
p = setScenarios(p, AssetScenarios);

plotFrontier(p);
```



If you did not specify the size of `LowerBound` but, instead, input a scalar argument, the `PortfolioMAD` function assumes that you are defining a single-asset problem and produces an error at the call to set asset scenarios with four assets.

Create a PortfolioMAD Object Using Shortcuts for Property Names

You can create a PortfolioMAD object, `p` with the `PortfolioMAD` function using shortcuts for property names.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
0.00408 0.0289 0.0204 0.0119;
0.00192 0.0204 0.0576 0.0336;
0 0.0119 0.0336 0.1225 ];

m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('scenario', AssetScenarios, 'lb', 0, 'budget', 1)
```

```
p =
  PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 20000
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
```

```
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Direct Setting of PortfolioMAD Object Properties

Although not recommended, you can set properties directly, however no error-checking is done on your inputs.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;

p = setScenarios(p, AssetScenarios);

p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));
disp(p);

  PortfolioMAD with properties:

        BuyCost: []
       SellCost: []
    RiskFreeRate: []
       Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 20000
            Name: []
       NumAssets: 4
       AssetList: []
```

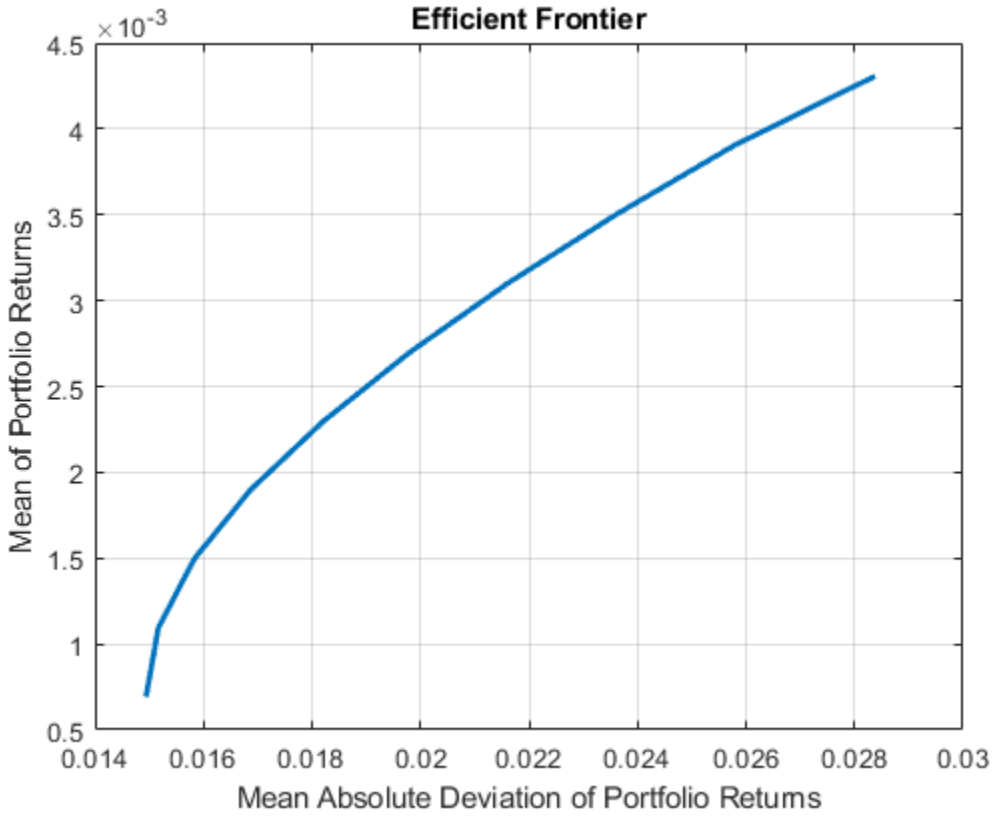
```
      InitPort: []
   AInequality: []
   bInequality: []
     AEquality: []
     bEquality: []
    LowerBound: [4x1 double]
    UpperBound: []
   LowerBudget: 1
   UpperBudget: 1
   GroupMatrix: []
    LowerGroup: []
    UpperGroup: []
        GroupA: []
        GroupB: []
    LowerRatio: []
    UpperRatio: []
```

Scenarios cannot be assigned directly to a PortfolioMAD object. Scenarios must always be set through either the `PortfolioMAD` function, the `setScenarios` function, or any of the scenario simulation functions.

Create a `PortfolioMAD` Object and Determine Efficient Portfolios

Create efficient portfolios:

```
load CAPMuniverse

p = PortfolioMAD('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
p = setDefaultConstraints(p);

plotFrontier(p);
```




```
pwgt = estimateFrontier(p, 5);

pnames = cell(1,5);
for i = 1:5
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1          Port2          Port3          Port4          Port5
    AAPL     0.030236       0.075387       0.11278        0.13456    1.5092e-14
```

```
AMZN    1.6541e-21    1.2618e-22    8.6501e-23     4.6635e-18    2.8997e-14
CSCO    1.5007e-22    7.7933e-23    3.6225e-23     1.1556e-33    4.1869e-14
DELL     0.0089659             0             0     2.5995e-17    3.9048e-14
EBAY    2.0446e-22             0    6.6122e-24     6.4047e-17    1.3394e-15
GOOG       0.16117       0.35201       0.54486        0.74888             1
HPQ       0.056551      0.024037             0    -3.6588e-33    3.8894e-14
IBM        0.45905       0.37891       0.29383        0.11656    3.7902e-14
INTC    -4.702e-38    6.0531e-22             0     1.7947e-17    3.8264e-14
MSFT       0.28403       0.16966      0.048527     8.3535e-18    4.0873e-14
ORCL    5.3466e-21             0    2.5731e-22     3.7312e-18    3.7811e-14
YHOO             0    3.5531e-23             0     1.4482e-33     3.535e-14
```

- "Creating the PortfolioMAD Object" on page 6-26
- "Common Operations on the PortfolioMAD Object" on page 6-34
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-61
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42
- "Validate the MAD Portfolio Problem" on page 6-91
- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-96
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-111
- "Postprocessing Results to Set Up Tradable Portfolios" on page 6-122

Input Arguments

`p` — Previously constructed PortfolioMAD object

object

Previously constructed `PortfolioMAD` object, specified using the `PortfolioMAD` function

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = PortfolioMAD('AssetList',Assets(1:12));`

`AEquality` — Linear equality constraint matrix
`[]` (default) | matrix

Linear equality constraint matrix, specified as a matrix.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix
`[]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix.

Data Types: `double`

`AssetList` — Names or symbols of assets in universe
`[]` (default) | cell array of character vectors

Names or symbols of assets in the universe, specified as a cell array of character vectors.

Data Types: `cell`

`bEquality` — Linear equality constraint vector
`[]` (default) | vector

Linear equality constraint vector, specified as a vector.

Data Types: `double`

`bInequality` — Linear inequality constraint
`[]` (default) | vector

Linear inequality constraint vector, specified as a vector.

Data Types: `double`

`BuyCost` — Proportional purchase costs
`[]` (default) | vector

Proportional purchase costs, specified as a vector.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases
`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar.

Data Types: `double`

`GroupA` — Group A weights to be bounded by weights in group B
`[]` (default) | matrix

Group A weights to be bounded by weights in group B, specified as a matrix.

Data Types: `double`

`GroupB` — Group B weights
`[]` (default) | matrix

Group B weights, specified as a matrix.

Data Types: `double`

`GroupMatrix` — Group membership matrix
`[]` (default) | matrix

Group membership matrix, specified as a matrix.

Data Types: `double`

`InitPort` — Initial portfolio
`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`LowerBound` — Lower-bound constraint
`[]` (default) | vector

Lower-bound constraint, specified as a vector.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint
`[]` (default) | scalar

Lower-bound budget constraint, specified as a scalar.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint
`[]` (default) | vector

Lower-bound group constraint, specified as a vector.

Data Types: `double`

`LowerRatio` — Minimum ratio of allocations between Groups A and B
`[]` (default) | vector

Minimum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object
`[]` (default) | character vector

Name for instance of the Portfolio object, specified as a character vector.

Data Types: `char`

`NumAssets` — Number of assets in the universe
`[]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

`NumScenarios` — Number of scenarios
`[]` (default) | integer scalar

Number of scenarios, specified as an integer scalar.

Data Types: `double`

`RiskFreeRate` — Risk-free rate
`[]` (default) | scalar

Risk-free rate, specified as a scalar.

Data Types: `double`

`SellCost` — Proportional sales costs
`[]` (default) | vector

Proportional sales costs, specified as a vector.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales
`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar.

Data Types: `double`

`Turnover` — Turnover constraint
`[]` (default) | scalar

Turnover constraint, specified as a scalar.

Data Types: `double`

`UpperBound` — Upper-bound constraint
`[]` (default) | vector

Upper-bound constraint, specified as a vector.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint
`[]` (default) | scalar

Upper-bound budget constraint, specified as a scalar.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint
`[]` (default) | vector

Upper-bound group constraint, specified as a vector.

Data Types: `double`

`UpperRatio` — Maximum ratio of allocations between Groups A and B
`[]` (default) | vector

Maximum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

Output Arguments

`p` — Updated PortfolioMAD object
object for PortfolioMAD

Updated MAD portfolio object, returned as a `PortfolioMAD`, For more information on using the `PortfolioMAD` object, see `Portfolio`.

Definitions

Mean-Absolute Deviation Portfolio Optimization

For more information on the theory and definition of mean-absolute deviation (MAD) optimization supported by portfolio optimization tools in Financial Toolbox, see “Portfolio Optimization Theory” on page 5-3.

PortfolioMAD Problem Sufficiency

A MAD portfolio optimization problem is completely specified with the PortfolioMAD object if three conditions are met.

The following are the three conditions that must be met:

- You must specify a collection of asset returns or prices known as scenarios such that all scenarios are finite asset returns or prices. These scenarios are meant to be samples from the underlying probability distribution of asset returns. This condition can be satisfied by the `setScenarios` function or with several canned scenario simulation functions.
- The set of feasible portfolios must be a nonempty compact set, where a compact set is closed and bounded. You can satisfy this condition using an extensive collection of properties that define different types of constraints to form a set of feasible portfolios. Since such sets must be bounded, either explicit or implicit constraints can be

imposed and several tools, such as the `estimateBounds` function, provide ways to ensure that your problem is properly formulated.

Although the general sufficient conditions for MAD portfolio optimization go beyond these conditions, the PortfolioMAD object handles all these additional conditions.

Shortcuts for Property Names

The `PortfolioMAD` function has shorter argument names that replace longer argument names associated with specific properties of the PortfolioMAD object.

For example, rather than enter `'AInequality'`, the `PortfolioMAD` function. accepts the case-insensitive name `'ai'` to set the `AInequality` property in a PortfolioMAD object. Every shorter argument name corresponds with a single property in the `PortfolioMAD` function. The one exception is the alternative argument name `'budget'`, which signifies both the `LowerBudget` and `UpperBudget` properties. When `'budget'` is used, then the `LowerBudget` and `UpperBudget` properties are set to the same value to form an equality budget constraint.

Shortcuts for Property Names

Shortcut Argument Name	Equivalent Argument / Property Name
`ae`	`AEquality`
`ai`	`AInequality`
`assetnames` or `assets`	`AssetList`
`be`	`bEquality`
`bi`	`bInequality`
`budget`	`UpperBudget` and `LowerBudget`
`group`	`GroupMatrix`
`lb`	`LowerBound`
`n` or `num`	`NumAssets`
`rfr`	`RiskFreeRate`
`scenario` or `assetscenarios`	`Scenarios`
`ub`	`UpperBound`

References

[1] For a complete list of references for the PortfolioMAD object, see “Portfolio Optimization” on page A-7.

See Also

`estimateFrontier` | `plotFrontier` | `setScenarios`

Topics

“Creating the PortfolioMAD Object” on page 6-26
“Common Operations on the PortfolioMAD Object” on page 6-34
“Working with MAD Portfolio Constraints Using Defaults” on page 6-61
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42
“Validate the MAD Portfolio Problem” on page 6-91
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-96
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-111
“Postprocessing Results to Set Up Tradable Portfolios” on page 6-122
“Portfolio Optimization Theory” on page 6-3
“PortfolioMAD Object Workflow” on page 6-19

Introduced in R2013b

PortfolioMAD Properties

Manage PortfolioMAD object for mean-absolute deviation portfolio optimization and analysis

Description

The main workflow for MAD portfolio optimization is to create an instance of a `PortfolioMAD` object that completely specifies a mean-absolute deviation optimization problem and to operate on the `PortfolioMAD` object using the supported object functions to obtain and analyze efficient portfolios.

The PortfolioMAD object and its associated functions are an interface for mean-absolute deviation optimization. So, almost everything you do with the PortfolioMAD object can be done using the associated functions. The basic workflow is:

1. Design your portfolio problem.
2. Use the `PortfolioMAD` function to create the PortfolioMAD object or use the various set functions to set up your portfolio problem.
3. Use estimate functions to solve your portfolio problem.

In addition, methods are available to help you view intermediate results and to diagnose your computations. Since MATLAB features are part of a PortfolioMAD object, you can save and load objects from your workspace and create and manipulate arrays of objects. After settling on a problem, which, in the case of MAD portfolio optimization, means that you have either scenarios, data, or moments for asset returns, and a collection of constraints on your portfolios, use the `PortfolioMAD` function to set the properties for the PortfolioMAD object.

The `PortfolioMAD` function lets you create an object from scratch or update an existing object. Since the PortfolioMAD object is a value object, it is easy to create a basic object, then use methods to build upon the basic object to create new versions of the basic object. This is useful to compare a basic problem with alternatives derived from the basic problem. For details, see “Creating the PortfolioMAD Object” on page 6-26.

For more information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-19 and for more detailed information on the

theoretical basis for mean-absolute deviation optimization, see “Portfolio Optimization Theory” on page 6-3.

Properties

Setting Up the Object

`AssetList` — cell array of character vectors
`[]` (default) | square matrix

Names or symbols of assets in the universe, specified as a cell array of character vectors

Data Types: `cell`

`InitPort` — Initial portfolio
`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object
`[]` (default) | character vector

Name for instance of the Portfolio object, specified as a character vector.

Data Types: `char`

`NumAssets` — Number of assets in the universe
`[]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

Portfolio Object Constraints

`AEquality` — Linear equality constraint matrix
`[]` (default) | matrix

Linear equality constraint matrix, specified as a matrix.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix

`[]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix.

Data Types: `double`

`bEquality` — Linear equality constraint vector

`[]` (default) | vector

Linear equality constraint vector, specified as a vector.

Data Types: `double`

`bInequality` — Linear inequality constraint

`[]` (default) | vector

Linear inequality constraint vector, specified as a vector.

Data Types: `double`

`GroupA` — Group A weights to be bounded by weights in group B

`[]` (default) | matrix

Group A weights to be bounded by weights in group B, specified as a matrix.

Data Types: `double`

`GroupB` — Group B weights

`[]` (default) | matrix

Group B weights, specified as a matrix.

Data Types: `double`

`GroupMatrix` — Group membership matrix

`[]` (default) | matrix

Group membership matrix, specified as a matrix.

Data Types: `double`

`LowerBound` — Lower-bound constraint

`[]` (default) | vector

Lower-bound constraint, specified as a vector.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint
`[]` (default) | scalar

Lower-bound budget constraint, specified as a scalar.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint
`[]` (default) | vector

Lower-bound group constraint, specified as a vector.

Data Types: `double`

`LowerRatio` — Minimum ratio of allocations between Groups A and B
`[]` (default) | vector

Minimum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

`UpperBound` — Upper-bound constraint
`[]` (default) | vector

Upper-bound constraint, specified as a vector.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint
`[]` (default) | scalar

Upper-bound budget constraint, specified as a scalar.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint
`[]` (default) | vector

Upper-bound group constraint, specified as a vector.

Data Types: `double`

`UpperRatio` — Maximum ratio of allocations between Groups A and B

`[]` (default) | vector

Maximum ratio of allocations between `GroupA` and `GroupB`, specified as a vector.

Data Types: `double`

Portfolio Object Modeling

`BuyCost` — Proportional purchase costs

`[]` (default) | vector

Proportional purchase costs, specified as a vector.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases

`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar.

Data Types: `double`

`RiskFreeRate` — Risk-free rate

`[]` (default) | scalar

Risk-free rate, specified as a scalar.

Data Types: `double`

`ProbabilityLevel` — Value-at-risk probability level which is 1 − (loss probability)

`[]` (default) | scalar

Value-at-risk probability level which is 1 − (loss probability), specified as a scalar.

Data Types: `double`

`NumScenarios` — Number of scenarios

`[]` (default) | integer scalar

Number of scenarios, specified as an integer scalar.

Data Types: `double`

`SellCost` — Proportional sales costs

`[]` (default) | vector

Proportional sales costs, specified as a vector.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales

`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar.

Data Types: `double`

`Turnover` — Turnover constraint

`[]` (default) | scalar

Turnover constraint, specified as a scalar.

Data Types: `double`

See Also

`PortfolioMAD`

Topics

“Creating the PortfolioMAD Object” on page 6-26
“Common Operations on the PortfolioMAD Object” on page 6-34
“Working with MAD Portfolio Constraints Using Defaults” on page 6-61
“Portfolio Optimization Theory” on page 6-3
“PortfolioCVaR Object Workflow” on page 5-20

portopt

Portfolios on constrained efficient frontier

Note `portopt` has been partially removed and will no longer accept `ConSet` or `varargin` arguments. Use `Portfolio` instead to solve portfolio problems that are more than a long-only fully-invested portfolio. For information on the workflow when using Portfolio objects, see "Portfolio Object Workflow". For more information on migrating `portopt` code to `Portfolio`, see "portopt Migration to Portfolio Object".

Syntax

```
[PortRisk,PortReturn,PortWts] = portopt(ExpReturn,ExpCovariance)

[PortRisk,PortReturn,PortWts] = portopt(ExpReturn,ExpCovariance,NumPorts)

[PortRisk,PortReturn,PortWts] = portopt(ExpReturn,ExpCovariance,NumPorts,PortReturn)
```

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.

Description

`[PortRisk,PortReturn,PortWts] = portopt(ExpReturn,ExpCovariance)` sets up the most basic portfolio problem with weights greater than or equal to `0` that must sum to `1`. All that is necessary to solve this problem is the mean and covariance of asset returns. The problem returns 10 equally-spaced points on the efficient frontier by return.

`[PortRisk,PortReturn,PortWts] = portopt(ExpReturn,ExpCovariance,NumPorts)` sets up the basic portfolio problem but lets you specify how many equally-spaced points on the efficient frontier that you want in `NumPorts`. If you specify `1`, it returns the minimum-risk portfolio.

`[PortRisk,PortReturn,PortWts] = portopt(ExpReturn,ExpCovariance,NumPorts,PortReturn)` sets up the basic portfolio problem but lets you specify target returns on the efficient frontier in the vector `PortReturn`. This functionality requires that if you set `PortReturn`, `NumPorts` should be empty.

Note `portopt` generates a warning if have returns outside the range and returns the portfolios at the endpoints of the efficient frontier.

The outputs for `portopt` are:

`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.

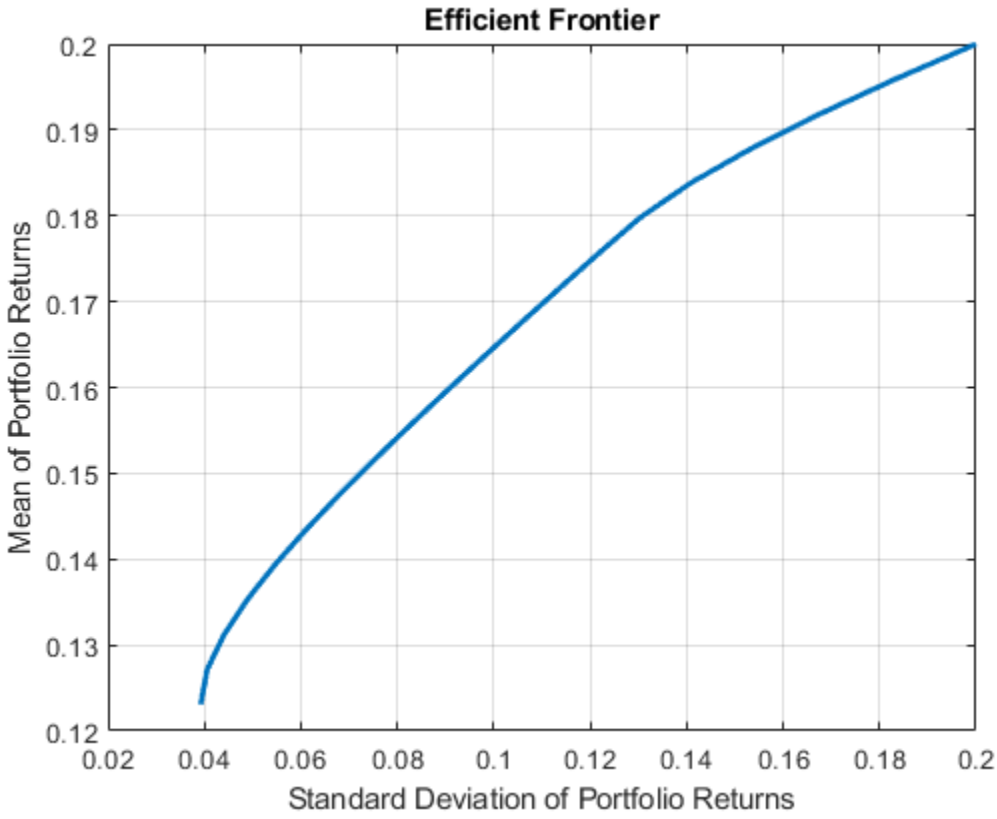
If `portopt` is invoked without output arguments, it writes to the current figure window.

Examples

Plot the Risk-Return Efficient Frontier

Use `portopt` to connect 20 portfolios along the efficient 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)
```



- “Portfolio Construction Examples” on page 3-7
- “Plotting an Efficient Frontier Using portopt” on page 10-27
- “Portfolio Selection and Risk Aversion” on page 3-9
- “Active Returns and Tracking Error Efficient Frontier” on page 3-43
- “portopt Migration to Portfolio Object” on page 3-14

See Also

`ewstats` | `frontier` | `portcons` | `portstats`

Topics

“Portfolio Construction Examples” on page 3-7
“Plotting an Efficient Frontier Using portopt” on page 10-27
“Portfolio Selection and Risk Aversion” on page 3-9
“Active Returns and Tracking Error Efficient Frontier” on page 3-43
“portopt Migration to Portfolio Object” on page 3-14
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

portrand

Randomized portfolio risks, returns, and weights

Syntax

```
[PortRisk,PortReturn,PortWts] = portrand(Asset,Return,Points,Method)

portrand(Asset,Return,Points,Method)
```

Arguments

`Asset`	Matrix of time series data. Each row is an observation and each column represents a single security.
`Return`	(Optional) Row vector where each column represents the rate of return for the corresponding security in `Asset`. By default, `Return` is computed by taking the average value of each column of `Asset`.
`Points`	(Optional) Scalar that specifies how many random points should be generated. Default = `1000`.
`Method`	(Optional) A character vector that specifies how to generate random portfolios from the set of portfolios with two possible methods: • `'uniform'` – Uniformly distributed portfolio weights (default method). The `'uniform'` method generates portfolio weights that are uniformly distributed on the set of portfolio weights. • `'geometric'` – Concentrated portfolio weights around the geometric center of the set of portfolios. The `'geometric'` method generates portfolio weights that are concentrated around the geometric center of the set of portfolio weights. **Note** The `'uniform'` and `'geometric'` methods generate weights that are distributed symmetrically around the geometric center of the set of weights.

Description

`[PortRisk,PortReturn,PortWts] = portrand(Asset,Return,Points,Method)` returns the risks, rates of return, and weights of random portfolio configurations.

`PortRisk`	`Points`-by-`1` vector of standard deviations.
`PortReturn`	`Points`-by-`1` vector of expected rates of return.
`PortWts`	`Points` by number of securities matrix of asset weights. Each row of `PortWts` is a different portfolio configuration.

`portrand(Asset, Return, Points, Method)` plots the points representing each portfolio configuration. It does not return any data to the MATLAB workspace.

Note Portfolios are selected at random from a set of portfolios such that portfolio weights are nonnegative and sum to 1. The sample mean and covariance of asset returns are used to compute portfolio returns for each random portfolio.

References

Bodie, Kane, and Marcus. *Investments.* Chapter 7.

See Also

`portror` | `portvar`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

portror

Portfolio expected rate of return

Syntax

```
R = portror(Return,Weight)
```

Arguments

`Return`	`1`-by-`N` matrix of rates of return. Each column of `Return` represents the rate of return for a single security
`Weight`	`M`-by-`N` matrix of weights. Each row of `Weight` represents a different weighting combination of the assets in the portfolio.

Description

`R = portror(Return,Weight)` returns a `1`-by-`M` vector for the expected rate of return.

Examples

Portfolio Expected Rate of Return

This example shows a portfolio that is made up of two assets ABC and XYZ having expected rates of return of 10% and 14%, respectively. If 40% percent of the portfolio's funds are allocated to asset ABC and the remaining funds are allocated to asset XYZ, the portfolio's expected rate of return is:

```
r = portror([.1 .14],[.4 .6])
```

```
r = 0.1240
```

- “Portfolio Construction Examples” on page 3-7

References

Bodie, Kane, and Marcus. *Investments.* Chapter 7.

See Also

`portrand` | `portvar`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

portsim

Monte Carlo simulation of correlated asset returns

Syntax

```
RetSeries = portsim(ExpReturn,ExpCovariance,NumObs,RetIntervals,NumSim,Method)
```

Arguments

`ExpReturn`	1 by number of assets (`NASSETS`) vector specifying the expected (mean) return of each asset.
`ExpCovariance`	`NASSETS`-by-`NASSETS` matrix of asset return covariances. `ExpCovariance` must be symmetric and positive semidefinite (no negative eigenvalues). The standard deviations of the returns are `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) Character vector 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

- 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 then

converted to asset prices, all terminal prices for a given asset are in close agreement. Although all realizations are drawn independently, they produce similar terminal asset prices. Set *Method* to `'Expected'` to avoid this behavior.

- The returns from the portfolios in `PortWts` are given by `PortReturn = PortWts * RetSeries(:,:,1)'`, where `PortWts` is a matrix in which each row contains the asset allocations of a portfolio. Each row of `PortReturn` corresponds to one of the portfolios identified in `PortWts`, and each column corresponds to one of the observations taken from the first realization (the first plane) in `RetSeries`. See `portopt` and `portstats` for portfolio specification and optimization.

Examples

Distinction Between Simulation Methods

This example shows the distinction between the `Exact` and `Expected` methods of simulation.

Consider a portfolio of five assets with the following expected returns, standard deviations, and correlation matrix based on daily asset returns (where `ExpReturn` and `Sigmas` are divided by 100 to convert percentages to returns).

```
ExpReturn     = [0.0246  0.0189  0.0273  0.0141  0.0311]/100;
Sigmas        = [0.9509  1.4259  1.5227  1.1062  1.0877]/100;
Correlations  = [1.0000  0.4403  0.4735  0.4334  0.6855
                 0.4403  1.0000  0.7597  0.7809  0.4343
                 0.4735  0.7597  1.0000  0.6978  0.4926
                 0.4334  0.7809  0.6978  1.0000  0.4289
                 0.6855  0.4343  0.4926  0.4289  1.0000];
```

Convert the correlations and standard deviations to a covariance matrix.

```
ExpCovariance = corr2cov(Sigmas, Correlations)
```

```
ExpCovariance =

   1.0e-03 *

    0.0904    0.0597    0.0686    0.0456    0.0709
```

```
    0.0597    0.2033    0.1649    0.1232    0.0674
    0.0686    0.1649    0.2319    0.1175    0.0816
    0.0456    0.1232    0.1175    0.1224    0.0516
    0.0709    0.0674    0.0816    0.0516    0.1183
```

Assume that there are 252 trading days in a calendar year, and simulate two sample paths (realizations) of daily returns over a two-year period. Since `ExpReturn` and `ExpCovariance` are expressed daily, set `RetIntervals = 1`.

```
StartPrice    = 100;
NumObs        = 504;  % two calendar years of daily returns
NumSim        = 2;
RetIntervals  = 1;    % one trading day
NumAssets     = 5;
```

To illustrate the distinction between methods, simulate two paths by each method, starting with the same random number state.

```
rng('default');
RetExact = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Exact');
rng(0);
RetExpected = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected');
```

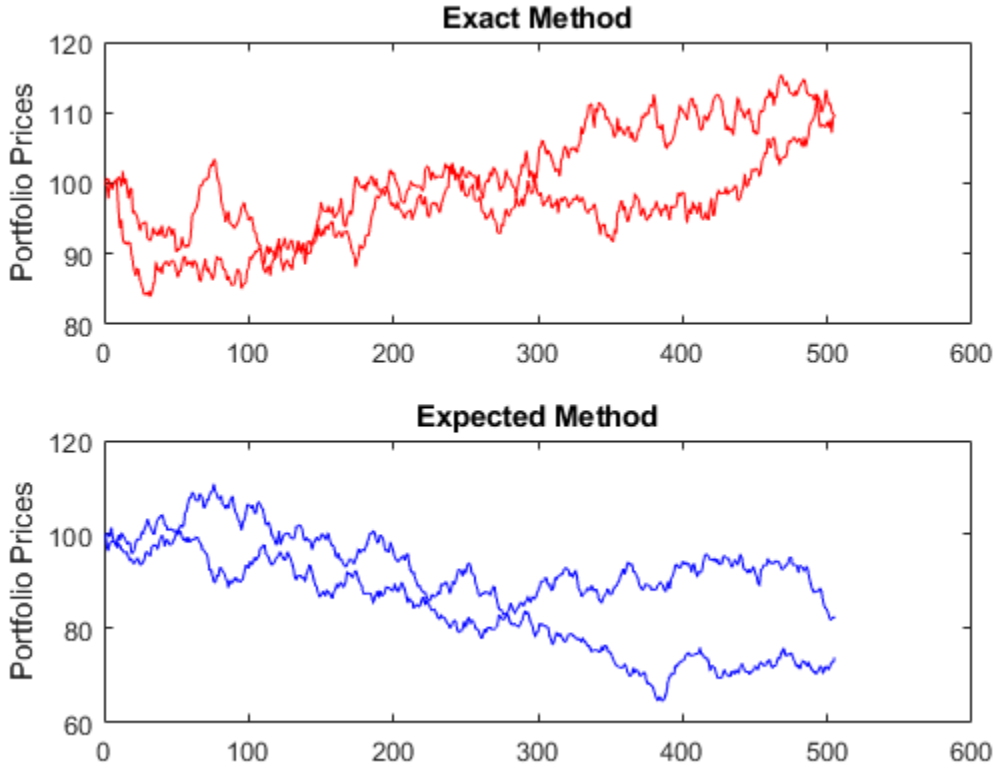
Compare the mean and covariance of `RetExact` with the inputs (`ExpReturn` and `ExpCovariance`), you will observe that they are almost identical.

At this point, `RetExact` and `RetExpected` are both 504-by-5-by-2 arrays. Now assume an equally weighted portfolio formed from the five assets and create arrays of portfolio returns in which each column represents the portfolio return of the corresponding sample path of the simulated returns of the five assets. The portfolio arrays `PortRetExact` and `PortRetExpected` are 504-by-2 matrices.

```
Weights         = ones(NumAssets, 1)/NumAssets;
PortRetExact    = zeros(NumObs, NumSim);
PortRetExpected = zeros(NumObs, NumSim);

for i = 1:NumSim
    PortRetExact(:,i)    = RetExact(:,:,i) * Weights;
    PortRetExpected(:,i) = RetExpected(:,:,i) * Weights;
end
```

Finally, convert the simulated portfolio returns to prices and plot the data. In particular, note that since the `Exact` method matches expected return and covariance, the terminal portfolio prices are virtually identical for each sample path. This is not true for the `Expected` simulation method. Although this example examines portfolios, the same methods apply to individual assets as well. Thus, `Exact` simulation is most appropriate when unique paths are required to reach the same terminal prices.

```
PortExact   = ret2tick(PortRetExact, ...
repmat(StartPrice,1,NumSim));
PortExpected = ret2tick(PortRetExpected, ...
repmat(StartPrice,1,NumSim));
subplot(2,1,1), plot(PortExact, '-r')
ylabel('Portfolio Prices')
title('Exact Method')
subplot(2,1,2), plot(PortExpected, '-b')
ylabel('Portfolio Prices')
title('Expected Method')
```



Interaction Between `ExpReturn`, `ExpCovariance`, and `RetIntervals`

This example shows the interplay among `ExpReturn`, `ExpCovariance`, and `RetIntervals`. Recall that `portsim` simulates correlated asset returns over an interval of length *dt*, given by the equation

$$\frac{dS}{S} = \mu dt + \sigma dz = \mu dt + \sigma\varepsilon\sqrt{dt},$$

where S is the asset price, μ is the expected rate of return, σ is the volatility of the asset price, and ε represents a random drawing from a standardized normal distribution.

The time increment dt is determined by the optional input `RetIntervals`, either as an explicit input argument or as a unit time increment by default. Regardless, the periodicity of `ExpReturn`, `ExpCovariance`, and `RetIntervals` must be consistent. For example, if `ExpReturn` and `ExpCovariance` are annualized, then `RetIntervals` must be in years. This point is often misunderstood.

To illustrate the interplay among `ExpReturn`, `ExpCovariance`, and `RetIntervals`, consider a portfolio of five assets with the following expected returns, standard deviations, and correlation matrix based on daily asset returns.

```
ExpReturn     = [0.0246  0.0189  0.0273  0.0141  0.0311]/100;

Sigmas        = [0.9509  1.4259  1.5227  1.1062  1.0877]/100;

Correlations  = [1.0000  0.4403  0.4735  0.4334  0.6855
                 0.4403  1.0000  0.7597  0.7809  0.4343
                 0.4735  0.7597  1.0000  0.6978  0.4926
                 0.4334  0.7809  0.6978  1.0000  0.4289
                 0.6855  0.4343  0.4926  0.4289  1.0000];
```

Convert the correlations and standard deviations to a covariance matrix of daily returns.

```
ExpCovariance = corr2cov(Sigmas, Correlations);
```

Assume 252 trading days per calendar year, and simulate a single sample path of daily returns over a four-year period. Since the `ExpReturn` and `ExpCovariance` inputs are expressed daily, set `RetIntervals = 1`.

```
StartPrice    = 100;
NumObs        = 1008;   % four calendar years of daily returns
RetIntervals  = 1;      % one trading day
NumAssets     = length(ExpReturn);
randn('state',0);
RetSeries1 = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, 1, 'Expected');
```

Now annualize the daily data, thereby changing the periodicity of the data, by multiplying `ExpReturn` and `ExpCovariance` by 252 and dividing `RetIntervals` by 252 (`RetIntervals` = 1/252 of a year). Resetting the random number generator to its initial state, you can reproduce the results.

```
rng('default');
RetSeries2 = portsim(ExpReturn*252, ExpCovariance*252, ...
NumObs, RetIntervals/252, 1, 'Expected');
```

Assume an equally weighted portfolio and compute portfolio returns associated with each simulated return series.

```
Weights  = ones(NumAssets, 1)/NumAssets;

PortRet1 = RetSeries2 * Weights;
PortRet2 = RetSeries2 * Weights;
```

Comparison of the data reveals that `PortRet1` and `PortRet2` are identical.

Univariate Geometric Brownian Motion

This example shows how to 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.

Assume that 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, `ExpCovariance` is entered as a variance rather than the more familiar standard deviation (volatility).

Set the random number generator state, and simulate 10,000 trials (realizations) of stock returns over a full calendar year of 252 trading days.

```
rng('default');
RetSeries = squeeze(portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected'));
```

The `squeeze` function reformats the output array of simulated returns from a `252`-by-`1`-by-`10000` array to more convenient `252`-by-`10000` array. (Recall that `portsim` is fundamentally a multivariate simulation engine).

In accordance with Hull's equations 12.4 and 12.5 on page 236

$$E(S_T) = S_0 e^{\mu T}$$

$$var(S_T) = S_0{}^2 e^{2\mu T} \left(e^{\sigma^2 T} - 1\right)$$

convert the simulated return series to a price series and compute the sample mean and the variance of the terminal stock prices.

```
StockPrices = ret2tick(RetSeries, repmat(StartPrice, 1, NumSim));

SampMean = mean(StockPrices(end,:))
SampVar = var(StockPrices(end,:))
```

```
SampMean =

   24.4489


SampVar =

  101.4243
```

Compare these values with the values you obtain by using Hull's equations.

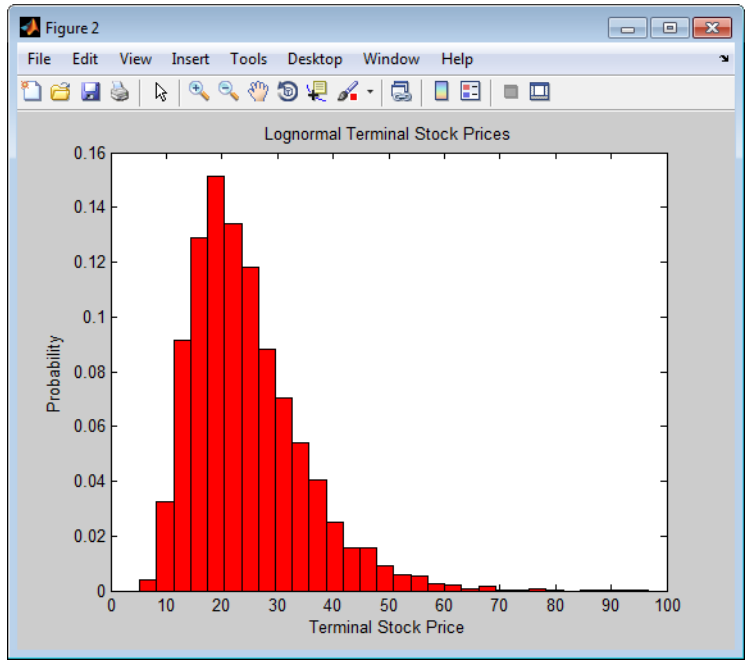
```
ExpValue = StartPrice*exp(ExpReturn)
ExpVar = ...
StartPrice*StartPrice*exp(2*ExpReturn)*(exp((ExpCovariance)) - 1)
```

```
ExpValue =

   24.4281


ExpVar =

  103.5391
```

These results are very close to the results shown in Hull's example 12.2.

Display the sample density function of the terminal stock price after one calendar year. From the sample density function, the lognormal distribution of terminal stock prices is apparent.

```
[count, BinCenter] = hist(StockPrices(end,:), 30);
figure
bar(BinCenter, count/sum(count), 1, 'r')
xlabel('Terminal Stock Price')
ylabel('Probability')
title('Lognormal Terminal Stock Prices')
```



- “Portfolio Construction Examples” on page 3-7

References

Hull, John, C. *Options, Futures, and Other Derivatives.* 5th Edition. Upper Saddle River, New Jersey: Prentice-Hall, 2003, ISBN 0-13-009056-5.

See Also

`ewstats` | `portopt` | `portstats` | `randn` | `ret2tick` | `squeeze`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

portstats

Portfolio expected return and risk

Syntax

```
[PortRisk,PortReturn] = portstats(ExpReturn,ExpCovariance,PortWts)
```

Arguments

`ExpReturn`	`1`-by-number of assets (`NASSETS`) vector specifying the expected (mean) return of each asset.
`ExpCovariance`	`NASSETS`-by-`NASSETS` matrix specifying the covariance of the asset returns.
`PortWts`	(Optional) Number of portfolios (`NPORTS`) by `NASSETS` matrix of weights allocated to each asset. Each row represents a different weighting combination. Default = `1/NASSETS` (equally weighted).

Description

`[PortRisk,PortReturn] = portstats(ExpReturn,ExpCovariance,PortWts)` computes the expected rate of return and risk for a portfolio of assets.

`PortRisk` is an `NPORTS`-by-`1` vector of the standard deviation of each portfolio.

`PortReturn` is an `NPORTS`-by-`1` vector of the expected return of each portfolio.

Examples

Computes the Expected Rate of Return and Risk for a Portfolio of Assets

This example shows how to calculate the expected rate of return and risk for a portfolio of assets.

```
ExpReturn = [0.1 0.2 0.15];

ExpCovariance = [0.0100   -0.0061    0.0042
                -0.0061    0.0400   -0.0252
                 0.0042   -0.0252    0.0225 ];

PortWts=[0.4 0.2 0.4; 0.2 0.4 0.2];

[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance,...
PortWts)
```

```
PortRisk =

    0.0560
    0.0550


PortReturn =

    0.1400
    0.1300
```

- "Portfolio Construction Examples" on page 3-7
- "Portfolio Selection and Risk Aversion" on page 3-9
- "Active Returns and Tracking Error Efficient Frontier" on page 3-43

See Also

`ewstats` | `portalloc`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Selection and Risk Aversion" on page 3-9
"Active Returns and Tracking Error Efficient Frontier" on page 3-43
"Analyzing Portfolios" on page 3-2

“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

portvar

Variance for portfolio of assets

Syntax

```
V = portvar(Asset,Weight)
```

Arguments

`Asset`	`M`-by-`N` matrix of `M` asset returns for `N` securities.
`Weight`	`R`-by-`N` matrix of `R` portfolio weights for `N` securities. Each row of `Weight` constitutes a portfolio of securities in `Asset`.

Description

`V = portvar(Asset,Weight)` returns the portfolio variance as an `R`-by-`1`vector (assuming `Weight` is a matrix of size `R`-by-`N`) with each row representing a variance calculation for each row of `Weight`.

`V = portvar(Asset)` assigns each security an equal weight when calculating the portfolio variance.

References

Bodie, Kane, and Marcus. *Investments.* Chapter 7.

See Also

`portrand` | `portror`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

portvrisk

Portfolio value at risk (VaR)

Syntax

```
ValueAtRisk = portvrisk(PortReturn,PortRisk)
ValueAtRisk = portvrisk(___,RiskThreshold,PortValue)
```

Description

`ValueAtRisk = portvrisk(PortReturn,PortRisk)` returns the maximum potential loss in the value of a portfolio over one period of time (that is, monthly, quarterly, yearly, and so on) given the loss probability level.

`ValueAtRisk = portvrisk(___,RiskThreshold,PortValue)` adds optional arguments for `RiskThreshold` and `PortValue`.

Examples

Compute the Maximum Potential Loss in the Value of a Portfolio Over One Period of Time

This example shows how to return the maximum potential loss in the value of a portfolio over one period of time, where `ValueAtRisk` is computed on a per-unit basis.

```
PortReturn = 0.29/100;
PortRisk = 3.08/100;
RiskThreshold = [0.01;0.05;0.10];
PortValue = 1;
ValueAtRisk = portvrisk(PortReturn,PortRisk,...
RiskThreshold,PortValue)
```

```
ValueAtRisk =

    0.0688
```

```
    0.0478
    0.0366
```

Compute the Maximum Potential Loss in the Value of a Portfolio Over One Period of Time Using Actual Values

This example shows how to return the maximum potential loss in the value of a portfolio over one period of time, where `ValueAtRisk` is computed with actual values.

```
PortReturn = [0.29/100;0.30/100];
PortRisk = [3.08/100;3.15/100];
RiskThreshold = 0.10;
PortValue = [1000000000;500000000];
ValueAtRisk = portvrisk(PortReturn,PortRisk,...
RiskThreshold,PortValue)
```

```
ValueAtRisk =

   1.0e+07 *

    3.6572
    1.8684
```

- “Portfolio Construction Examples” on page 3-7

Input Arguments

`PortReturn` — Expected return of each portfolio over period

numeric

Expected return of each portfolio over the period, specified as a `NPORTS`-by-`1` vector or scalar.

Data Types: `double`

`PortRisk` — Standard deviation of each portfolio over period

numeric

Standard deviation of each portfolio over period, specified as a `NPORTS`-by-`1` vector or scalar.

Data Types: `double`

`RiskThreshold` — Loss probability
`0.05` (5%) (default) | decimal

(Optional) Loss probability, specified as a `NPORTS`-by-`1` vector or scalar.

Data Types: `double`

`PortValue` — Total value of asset portfolio
`1` (default) | numeric

(Optional) Total value of asset portfolio, specified as a `NPORTS`-by-`1` vector or scalar.

Note If `PortReturn` and `PortRisk` are in dollar units, then `PortValue` should be `1`. If `PortReturn` and `PortRisk` are on a percentage basis, then `PortValue` should be the total value of the portfolio.

Data Types: `double`

Output Arguments

`ValueAtRisk` — Estimated maximum loss in portfolio
vector

Estimated maximum loss in the portfolio, returned as an `NPORTS`-by-`1` vector. `ValueAtRisk` is predicted with a confidence probability of `1` − `RiskThreshold`. `portvrisk` calculates `ValueAtRisk` using a normal distribution.

Note If `PortValue` is not given, `ValueAtRisk` is presented on a per-unit basis. A value of `0` indicates no losses.

See Also

`Portfolio` | `portopt`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

posvolidx

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

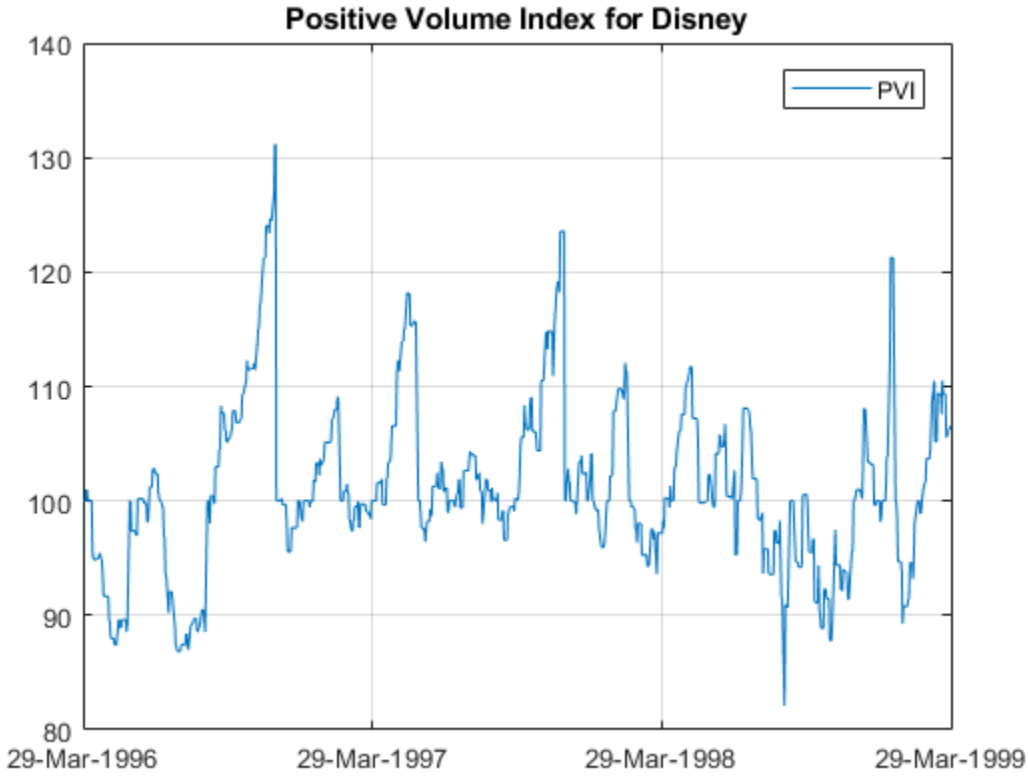
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Positive Volume Index

This example shows how to compute the positive volume index for Disney stock and plot the results.

```
load disney.mat
dis_PosVol = posvolidx(dis);
plot(dis_PosVol)
title('Positive Volume Index for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 236–238.

See Also

`negvolidx` | `onbalvol`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

power

Financial time series power

Syntax

```
newfts = tsobj .^ array

newfts = array .^tsobj

newfts = tsobj_1 .^ tsobj_2
```

Arguments

`tsobj`	Financial time series object.
`array`	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`	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`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

prbyzero

Price bonds in portfolio by set of zero curves

Syntax

```
BondPrices = prbyzero(Bonds,Settle,ZeroRates,ZeroDates)
BondPrices = prbyzero(___,Compounding)
```

Description

`BondPrices = prbyzero(Bonds,Settle,ZeroRates,ZeroDates)` computes the bond prices in a portfolio using a set of zero curves.

`BondPrices = prbyzero(___,Compounding)` adds an optional argument for `Compounding`.

Examples

Compute the Bond Prices in a Portfolio Using a Set of Zero Curves

This example uses the function `zbtprice` to compute a zero curve given a portfolio of coupon bonds and their prices. It then reverses the process, using the zero curve as input to the function `prbyzero` to compute the prices.

```
Bonds = [datenum('6/1/1998') 0.0475 100 2 0 0;
         datenum('7/1/2000') 0.06 100 2 0 0;
         datenum('7/1/2000') 0.09375 100 6 1 0;
         datenum('6/30/2001') 0.05125 100 1 3 1;
         datenum('4/15/2002') 0.07125 100 4 1 0;
         datenum('1/15/2000') 0.065 100 2 0 0;
         datenum('9/1/1999') 0.08 100 3 3 0;
         datenum('4/30/2001') 0.05875 100 2 0 0;
         datenum('11/15/1999') 0.07125 100 2 0 0;
         datenum('6/30/2000') 0.07 100 2 3 1;
```

```
          datenum('7/1/2001') 0.0525 100 2 3 0;
          datenum('4/30/2002') 0.07 100 2 0 0];

Prices = [ 99.375;
           99.875;
          105.75 ;
           96.875;
          103.625;
          101.125;
          103.125;
           99.375;
          101.0  ;
          101.25 ;
           96.375;
          102.75 ];

Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve, on an actual/365 basis.

```
OutputCompounding = 2;
```

Execute the function `zbtprice` which returns the zero curve at the maturity dates.

```
[ZeroRates, ZeroDates] = zbtprice(Bonds, Prices, Settle,...
OutputCompounding)

ZeroRates =

    0.0616
    0.0609
    0.0658
    0.0590
    0.0647
    0.0655
    0.0606
    0.0601
    0.0642
    0.0621


ZeroDates =

      729907
      730364
```

```
      730439
      730500
      730667
      730668
      730971
      731032
      731033
      731321
```

Execute the function `prbyzero`.

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

```
BondPrices =

   99.3750
   98.7980
  106.8270
   96.8750
  103.6249
  101.1250
  103.1250
   99.3637
  101.0000
  101.2500
```

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.

Compute the Bond Prices in a Portfolio Using a Set of Zero Curves and datetime Inputs

This example uses the function `zbtprice` to compute a zero curve given a portfolio of coupon bonds and their prices. It then reverses the process, using the zero curve as input to the function `prbyzero` with `datetime` inputs 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');
OutputCompounding = 2;

[ZeroRates, ZeroDates] = zbtprice(Bonds, Prices, Settle, OutputCompounding);

dates = datetime(Bonds(:,1),'ConvertFrom','datenum','Locale','en_US');
data = Bonds(:,2:end);
t=[table(dates) array2table(data)];
BondPrices = prbyzero(t, datetime(Settle,'ConvertFrom','datenum','Locale','en_US'),...
ZeroRates, datetime(ZeroDates,'ConvertFrom','datenum','Locale','en_US'))

BondPrices =

   99.3750
   98.7980
  106.8270
   96.8750
  103.6249
  101.1250
  103.1250
   99.3637
```

```
101.0000
101.2500
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`Bonds` — Coupon bond information to compute prices

table | matrix

Coupon bond information to compute prices, specified as a 6-column table or a `NumBonds`-by-`6` matrix of bond information where the table columns or matrix columns contains:

- `Maturity` (Required) Maturity date of the bond, as a serial date number. Use `datenum` to convert date character vectors to serial date numbers. If the input `Bonds` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- `CouponRate` (Required) 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 bond. A vector of integers.
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)

  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)
  - 11 = 30/360E (ICMA)
  - 12 = actual/365 (ISDA)
  - 13 = BUS/252
  - For more information, see **basis** on page Glossary-0 .
- `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

:

Note

- If `Bonds` is a table, the table columns have the same meaning as when a matrix is used, but the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- If `Bonds` is a matrix, it is a `NUMBONDS`-by-`6` matrix of bonds where each row describes one bond. The first two columns are required; the remaining columns are optional but must be added in order. All rows in `Bonds` must have the same number of columns. The columns are `Maturity`, `CouponRate`, `Face`, `Period`, `Basis`, and `EndMonthRule`.

.

Data Types: `double` | `table`

`Settle` — Settlement date
serial date number | date character vector | datetime

Settlement date, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`ZeroRates` — Observed zero rates
decimal fractions

Observed zero rates, specified as `NUMDATES`-by-`NUMCURVES` matrix of decimal fractions. Each column represents a rate curve. Each row represents an observation date.

Data Types: `double` | `datetime` | `char`

`ZeroDates` — Observed dates for `ZeroRates`
serial date number | date character vector | datetime

Observed dates for `ZeroRates`, specified as a `NUMDATES`-by-`1` column using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`Compounding` — Compounding frequency of input `ZeroRates` vwhen annualized
`2` (default) | numeric values: `1`, `2`, `3`, `4`, `6`, `12`,

(Optional) Compounding frequency of input `ZeroRates` when annualized, specified using the allowed values:

- `1` — Annual compounding
- `2` — Semiannual compounding (default)
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Data Types: `double`

Output Arguments

`BondPrices` — Clean bond prices
numeric

Clean bond prices, returned as a `NUMBONDS`-by-`NUMCURVES` matrix. Each column is derived from the corresponding zero curve in `ZeroRates`.

In addition, you can use theFinancial Instruments Toolbox method `getZeroRates` for an `IRDataCurve` object with a `Dates` property to create a vector of dates and data acceptable for `prbyzero`. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” (Financial Instruments Toolbox).

See Also

`datetime` | `tr2bonds` | `zbtprice`

Topics

“Term Structure of Interest Rates” on page 2-45
“Fixed-Income Terminology” on page 2-25

Introduced before R2006a

prcroc

Price rate of change

Syntax

```
proc = prcroc(closep,nTimes)

procts = prcroc(tsobj,nTimes)

procts = prcroc(tsobj,nTimes,'ParameterName',ParameterValue, ...)
```

Arguments

`closep`	Closing price
`nTimes`	(Optional) Time difference. Default = `12`.
`tsobj`	Financial time series object

Description

`proc = prcroc(closep,nTimes)` calculates the price rate of change `proc` from the closing price `closep`. If `nTimes` time is specified, the price rate of change is calculated between the current closing price and the closing price `nTimes` ago.

`procts = prcroc(tsobj,nTimes)` calculates the price rate of change `procts` from the financial time series object `tsobj`. `tsobj` must contain a data series named `Close`. The output `procts` is a financial time series object with similar dates as `tsobj` and a data series named `PriceROC`. If `nTimes` is specified, the price rate of change is calculated between the current closing price and the closing price `nTimes` ago.

`procts = prcroc(tsobj,nTimes,'ParameterName',ParameterValue, ...)` specifies the name for the required data series when it is different from the default name. The valid parameter name is

- `CloseName`: closing price series name

The parameter value is a character vector that represents the valid parameter name.

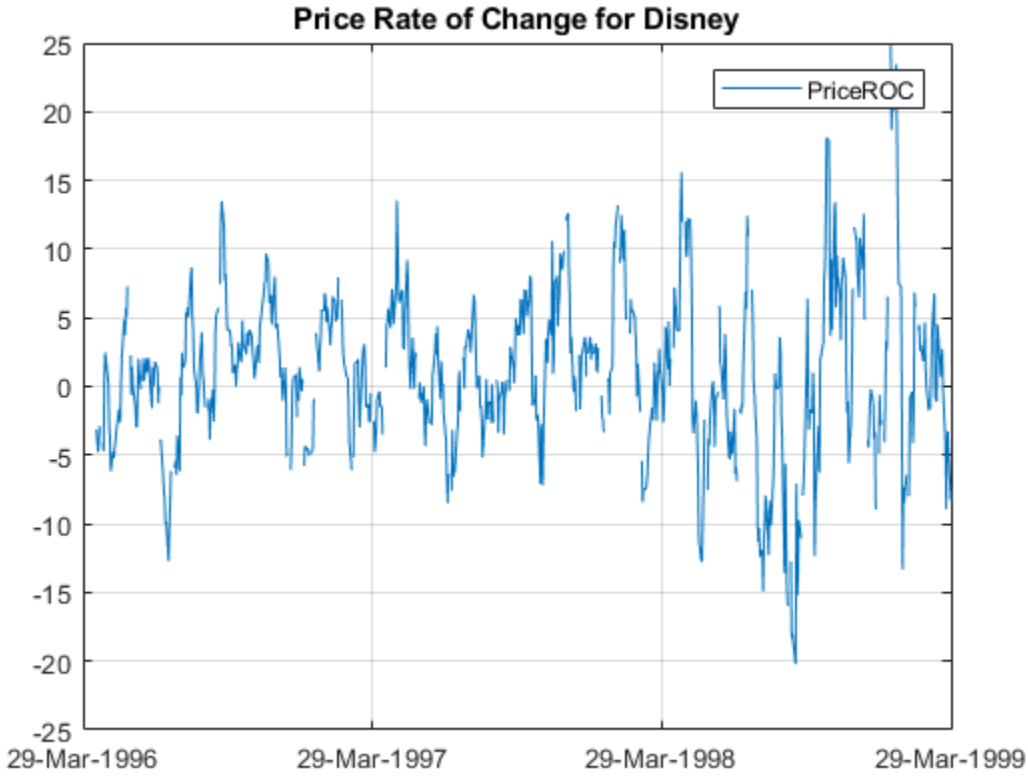
Note, to compute a quantity over `n` periods, you must specify `n+1` for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Compute the Price Rate of Change

This example shows how to compute the price rate of change for Disney stock and plot the results.

```
load disney.mat
dis_PriceRoc = prcroc(dis);
plot(dis_PriceRoc)
title('Price Rate of Change for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 243–245.

See Also

`volroc`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

prdisc

Price of discounted security

Syntax

```
Price = prdisc(Settle,Maturity,FaceDiscount)
Price = prdisc(___,Basis)
```

Description

`Price = prdisc(Settle,Maturity,FaceDiscount)` returns the price of a security whose yield is quoted as a bank discount rate (for example, U. S. Treasury bills).

`Price = prdisc(___,Basis)` adds an optional argument for `Basis`.

Examples

Calculate the Price of a Security Whose Yield is Quoted as a Bank Discount Rate

This example shows how to return the price of a security whose yield is quoted as a bank discount rate (for example, U. S. Treasury bills).

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Discount = 0.087;
Basis = 2;

Price = prdisc(Settle, Maturity, Face, Discount, Basis)

Price = 96.2783
```

Calculate the Price of a Security Whose Yield is Quoted as a Bank Discount Rate Using datetime Inputs

This example shows how to use `datetime` inputs to return the price of a security whose yield is quoted as a bank discount rate (for example, U. S. Treasury bills).

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Discount = 0.087;
Basis = 2;

Price = prdisc(datetime(Settle,'Locale','en_US'),datetime(Maturity,'Locale','en_US'), F

Price = 96.2783
```

- “Pricing Functions” on page 2-35

Input Arguments

`Settle` — Settlement date

serial date number | date character vector | datetime

Settlement date, specified as serial date numbers, date character vectors, or datetime arrays.

Settle must be earlier than `Maturity`.

Data Types: `double` | `datetime` | `char`

`Maturity` — Maturity date

serial date number | date character vector | datetime

Maturity date, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`Face` — Redemption (par, face) value

numeric

Redemption (par, face) value, specified as a numeric value.

Data Types: `double`

`Discount` — Bank discount rate of the security

decimal fraction

Bank discount rate of the security, specified as a decimal fraction value.

Data Types: `double`

`Basis` — Day-count basis of instrument

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

(Optional) Day-count basis of the instrument, specified as a numeric value. Allowed values are:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`Price` — Price of discounted security
numeric

Price of discounted security, returned as a numeric value.

References

[1] Mayle. *"Standard Securities Calculation Methods."* Volumes I-II, 3rd edition. Formula 2.

See Also

`acrudisc` | `bndprice` | `datetime` | `discrate` | `prmat` | `ylddisc`

Topics

"Pricing Functions" on page 2-35
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

priceandvol

Price and volume chart

Syntax

```
priceandvol(X)
```

Arguments

X	X can be a M-by-6 matrix or table. If X is M-by-6 matrix , the columns are date, open, high, low, close, and volume. If X is a table, the first column of the table is the date, and can be either serial date numbers, date character vectors, or datetime arrays. The other columns represent the same data as in the matrix version of the input.

Description

`priceandvol(X)` plots the asset data displaying the open, high, low, and closing prices on one axis and the volume on a second axis.

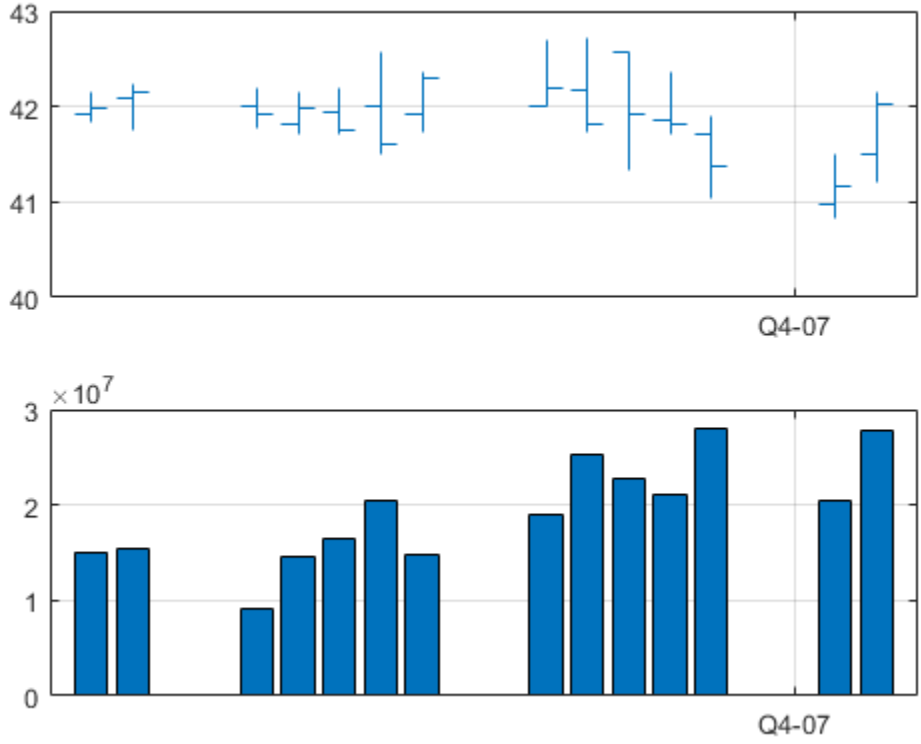
Examples

Create a Price Volume Chart

This example shows how to create a price volume chart, given asset X as an M-by-6 matrix for date, open, high, low, close, and volume.

```
X = [...

733299.00         41.93         42.15         41.83         41.99   15045445.00;...
733300.00         42.09         42.24         41.76         42.14   15346658.00;...
```

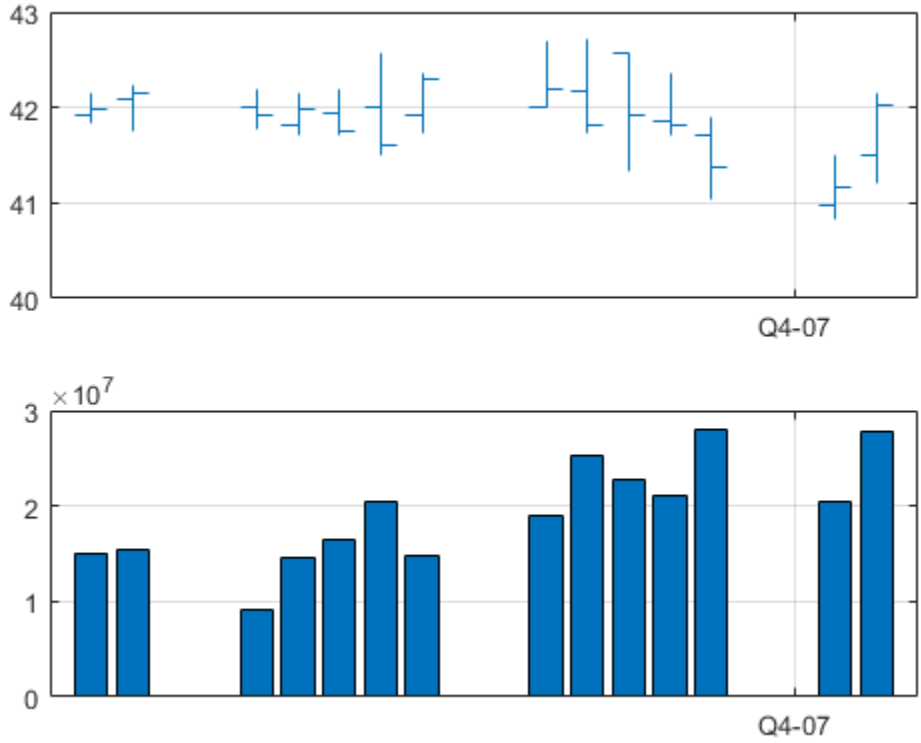
```
733303.00        42.00        42.20        41.78        41.93     9034397.00;...
733304.00        41.82        42.16        41.70        41.98    14486275.00;...
733305.00        41.94        42.19        41.70        41.75    16389872.00;...
733306.00        42.00        42.57        41.50        41.61    20475208.00;...
733307.00        41.93        42.35        41.74        42.29    14833200.00;...
733310.00        42.01        42.70        42.01        42.19    18945176.00;...
733311.00        42.18        42.72        41.73        41.82    25188101.00;...
733312.00        42.57        42.57        41.33        41.93    22689878.00;...
733313.00        41.86        42.35        41.71        41.81    21084723.00;...
733314.00        41.70        41.90        41.04        41.37    27963619.00;...
733317.00        40.98        41.49        40.82        41.17    20385033.00;...
733318.00        41.50        42.15        41.21        42.02    27783775.00];

priceandvol(X)
```



Create a Price Volume Chart Using datetime Input

This example shows how to use `datetime` input to create a price volume chart, given asset `X` as an `M`-by-6 matrix for date, open, high, low, close, and volume.

```
X = [...
733299.00         41.93         42.15         41.83         41.99   15045445.00;...
733300.00         42.09         42.24         41.76         42.14   15346658.00;...
733303.00         42.00         42.20         41.78         41.93    9034397.00;...
733304.00         41.82         42.16         41.70         41.98   14486275.00;...
733305.00         41.94         42.19         41.70         41.75   16389872.00;...
733306.00         42.00         42.57         41.50         41.61   20475208.00;...
733307.00         41.93         42.35         41.74         42.29   14833200.00;...
733310.00         42.01         42.70         42.01         42.19   18945176.00;...
733311.00         42.18         42.72         41.73         41.82   25188101.00;...
733312.00         42.57         42.57         41.33         41.93   22689878.00;...
733313.00         41.86         42.35         41.71         41.81   21084723.00;...
733314.00         41.70         41.90         41.04         41.37   27963619.00;...
733317.00         40.98         41.49         40.82         41.17   20385033.00;...
733318.00         41.50         42.15         41.21         42.02   27783775.00];

dates = datetime(X(:,1),'ConvertFrom','datenum','Locale','en_US');
data = X(:,2:end);
t=[table(dates) array2table(data)];
priceandvol(t);
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `bolling` | `datetime` | `highlow` | `kagi` | `linebreak` | `movavg` | `pointfig` | `renko` | `volarea`

Topics

“Charting Financial Data” on page 2-14

Introduced in R2008a

prmat

Price with interest at maturity

Syntax

```
[Price,AccruInterest] = prmat(Settle,Maturity,Issue,Face,CouponRate
Yield)
[Price,AccruInterest] = prmat(___,Basis)
```

Description

`[Price,AccruInterest] = prmat(Settle,Maturity,Issue,Face,CouponRate Yield)` 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`.

`[Price,AccruInterest] = prmat(___,Basis)` adds an optional argument for `Basis`.

Examples

Find the Yield of a Security Paying Interest at Maturity

This example shows how to find the yield of a security paying interest at maturity for the following.

```
Settle = '02/07/2000';
Maturity = '04/13/2000';
Issue = '10/11/1999';
Face = 100;
Price = 99.98;
CouponRate = 0.0608;
Basis = 1;
```

```
Yield = yldmat(Settle, Maturity, Issue, Face, Price,...
CouponRate, Basis)

Yield = 0.0607
```

Find the Yield of a Security Paying Interest at Maturity Using datetime Inputs

This example shows how to use `datetime` inputs find the yield of a security paying interest at maturity for the following:

```
Settle = '7-Feb-2000';
Maturity = '13-Apr-2000';
Issue = '11-Oct-1999';
Face = 100;
Price = 99.98;
CouponRate = 0.0608;
Basis = 1;

Settle = datetime(Settle,'Locale','en_US');
Maturity = datetime(Maturity,'Locale','en_US');
Issue = datetime(Issue,'Locale','en_US');

Yield = yldmat(Settle, Maturity, Issue, Face, Price,...
CouponRate, Basis)

Yield = 0.0607
```

- “Yield Functions” on page 2-35

Input Arguments

`Settle` — Settlement date of security

serial date number | date character vector | datetime

Settlement date of the security, specified as serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of security
serial date number | date character vector | datetime

Maturity date of the security, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Issue` — Issue date
serial date number | date character vector | datetime

Issue date of the security, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value
numeric

Redemption value (par value), specified as a numeric value.

Data Types: `double`

`CouponRate` — Coupon rate
decimal fraction

Coupon rate, specified as a decimal fraction value.

Data Types: `double`

`Yield` — Annual yield
decimal fraction

Annual yield, specified as a decimal fraction value.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis of the security, specified using the following values:

- 0 = actual/actual

- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`Price` — Security price

numeric

Security price, returned as a numeric value.

`AccruInterest` — Accured interest

numeric

Accured interest for security, returned as a numeric value.

References

[1] Mayle, J. *Standard Securities Calculation Methods*. Volumes I-II, 3rd edition. Formula 3.

See Also

`acrudisc` | `bndprice` | `bndyield` | `datetime` | `prmat` | `ylddisc` | `yldtbill`

Topics

"Yield Functions" on page 2-35
"Yield Conventions" on page 2-34

Introduced before R2006a

prtbill

Price of Treasury bill

Syntax

```
Price = prtbill(Settle,Maturity,Face,Discount)
```

Description

`Price = prtbill(Settle,Maturity,Face,Discount)` returns the price for a Treasury bill.

Examples

Calculate the Price for a Treasury Bill

This example shows how to return the price for a Treasury bill, where the settlement date of a Treasury bill is February 10, 2002, the maturity date is August 6, 2002, the discount rate is 3.77%, and the par value is $1000.

```
Price = prtbill('2/10/2002', '8/6/2002', 1000, 0.0377)
```

```
Price = 981.4642
```

Calculate the Price for a Treasury Bill Using datetime Inputs

This example shows how to use `datetime` inputs to return the price for a Treasury bill, where the settlement date of a Treasury bill is February 10, 2002, the maturity date is August 6, 2002, the discount rate is 3.77%, and the par value is $1000.

```
Price = prtbill(datetime('10-Feb-2002','Locale','en_US'), datetime('6-Aug-2002','Locale
```

```
Price = 981.4642
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`Settle` — Settlement date for Treasury bill

serial date number | date character vector | datetime

Settlement date for the Treasury bill, specified as serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date for Treasury bill

serial date number | date character vector | datetime

Maturity date for the Treasury bill, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value of Treasury bill

numeric

Redemption value (par value) of the Treasury bill, specified as a numeric value.

Data Types: `double`

`Discount` — Discount rate of Treasury bill

decimal fraction

Discount rate of the Treasury bill, specified as a decimal fraction value.

Data Types: `double`

Output Arguments

`Price` — Treasury bill price

numeric

Treasury bill price, returned as a numeric value.

References

[1] Bodie, Kane, and Marcus. *Investments*. McGraw-Hill Education, 2013.

See Also

`beytbill` | `datetime` | `yldtbill`

Topics

"Computing Treasury Bill Price and Yield" on page 2-41
"Treasury Bills Defined" on page 2-40

Introduced before R2006a

pvfix

Present value with fixed periodic payments

Syntax

```
PresentVal = pvfix(Rate,NumPeriods,Payment,ExtraPayment,Due)
```

Arguments

`rate`	Periodic interest rate, as a decimal fraction.
`NumPeriods`	Number of periods.
`Payment`	Periodic payment.
`ExtraPayment`	(Optional) Payment received other than `Payment` in the last period. Default = `0`.
`Due`	(Optional) When payments are due or made: `0` = end of period (default), or `1` = beginning of period.

Description

`PresentVal = pvfix(Rate,NumPeriods,Payment,ExtraPayment,Due)` returns the present value of a series of equal payments.

Examples

Calculate the Present Value of a Series of Equal Payments

This example shows how to return the present value of a series of equal payments, where $200 is paid monthly into a savings account earning 6%. The payments are made at the end of the month for five years.

```
PresentVal = pvfix(0.06/12, 5*12, 200, 0, 0)

PresentVal = 1.0345e+04
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

fvfix | fvvar | payper | pvvar

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

pvtrend

Price and Volume Trend (PVT)

Syntax

```
pvt = pvtrend(closep,tvolume)

pvt = pvtrend([closep tvolume])

pvtts = pvtrend(tsobj)

pvtts = pvtrend(tsobj,'ParameterName',ParameterValue, ...)
```

Arguments

`closep`	Closing price.
`tvolume`	Volume traded.
`tsobj`	Financial time series object.
`'ParameterName'`	Valid parameter names are: • `CloseName` — closing prices series name • `VolumeName` — volume traded series name
`ParameterValue`	Parameter values are the character vectors that represent the valid parameter names.

Description

`pvt = pvtrend(closep,tvolume)` calculates the Price and Volume Trend (PVT) from the stock closing price (`closep`) data and the volume traded (`tvolume`) data.

`pvt = pvtrend([closep tvolume])` accepts a two-column matrix in which the first column contains the closing prices (`closep`) and the second contains the volume traded (`tvolume`).

`pvtts = pvtrend(tsobj)` calculates the PVT from the stock data contained in the financial time series object `tsobj`. The object `tsobj` must contain the closing price series `Close` and the volume traded series `Volume`. The output `pvtts` is a financial time series object with dates similar to `tsobj` and a data series named `PVT`.

`pvtts = pvtrend(tsobj,'ParameterName',ParameterValue, ...)` accepts parameter name/ parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Parameter values are the character vectors that represent the valid parameter names.

Examples

Calculate the Price and Volume Trend (PVT)

This example shows how to calculate the PVT for Disney stock and plot the results.

```
load disney.mat
dis_PVTrend = pvtrend(dis);
plot(dis_PVTrend)
title('Price and Volume Trend for Disney')
```

- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 239–240.

See Also

`onbalvol` | `volroc`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

pvvar

Present value of varying cash flow

Syntax

```
PresentVal = pvvar(CashFlow,Rate,CFDates)
```

Arguments

`CashFlow`	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number). If `CashFlow` is a matrix, each column is treated as a separate cash-flow stream.
`Rate`	Periodic interest rate. Enter as a decimal fraction. If `CashFlow` is a matrix, a scalar `Rate` is allowed when the same rate applies to all cash-flow streams in `CashFlow`. When multiple cash-flow streams require different discount rates, `Rate` must be a vector whose length equals the number of columns in `CashFlow`.
`CFDates`	(Optional) A vector of serial date numbers, date character vectors, or datetime arrays on which the cash flows occur. Specify `CFDates` when there are irregular (nonperiodic) cash flows. The default assumes that `CashFlow` contains regular (periodic) cash flows. If `CashFlow` is a matrix, and all cash-flow streams share the same dates, `CFDates` can be a vector whose length matches the number of rows in `CashFlow`. When different cash-flow streams have different payment dates, specify `CFDates` as a matrix the same size as `CashFlow`.

Description

`PresentVal = pvvar(CashFlow,Rate,CFDates)` returns the net present value of a varying cash flow. Present value is calculated at the time the first cash flow occurs.

Examples

This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 8%.

Year 1	$2000
Year 2	$1500
Year 3	$3000
Year 4	$3800
Year 5	$5000

To calculate the net present value of this regular cash flow

```
PresentVal = pvvar([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
PresentVal =

            1715.39
```

An investment of $10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

Cash Flow	Dates
($10000)	January 12, 1987
$2500	February 14, 1988
$2000	March 3, 1988
$3000	June 14, 1988
$4000	December 1, 1988

To calculate the net present value of this irregular cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];

CFDates = ['01/12/1987'
           '02/14/1988'
           '03/03/1988'
           '06/14/1988'
           '12/01/1988'];
```

```
PresentVal = pvvar(CashFlow, 0.09, CFDates)
```

returns

```
PresentVal =

             142.16
```

The net present value of the same investment under different discount rates of 7%, 9%, and 11% is obtained in a single call:

```
PresentVal = pvvar(repmat(CashFlow,3,1)', [.07 .09 .11], CFDates)

pv =

  419.0136  142.1648 -122.1275
```

See Also

datetime | fvfix | fvvar | irr | payuni | pvfix

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

pyld2zero

Zero curve given par yield curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the new optional name-value pair inputs: `InputCompounding`, `InputBasis`, `OutputCompounding`, and `OutputBasis`.

Syntax

```
[ZeroRates,CurveDates] = pyld2zero(ParRates,CurveDates,Settle)
[ZeroRates,CurveDates] = pyld2zero(___,Name,Value)
```

Description

`[ZeroRates,CurveDates] = pyld2zero(ParRates,CurveDates,Settle)` returns a zero curve given a par yield curve and its maturity dates. If either input for `CurveDates` or `Settle` is a datetime array, `CurveDates` is returned as a datetime array. Otherwise, `CurveDates` is returned as a serial date number.

`[ZeroRates,CurveDates] = pyld2zero(___,Name,Value)` adds optional name-value pair arguments

Examples

Compute Zero Curve Given Par Yield Curve

Define the settlement date, maturity, and zero rates.

```
Settle = datenum('01-Feb-2013');
CurveDates = datemnth(Settle,12*[1 2 3 5 7 10 20 30]');
ZeroRates = [.11 0.30 0.64 1.44 2.07 2.61 3.29 3.55]'/100;
```

```
InputCompounding = 2;
InputBasis = 1;
OutputCompounding = 2;
OutputBasis = 1;
```

Compute par yield curve from zero rates.

```
ParRates = zero2pyld(ZeroRates, CurveDates, Settle,'InputCompounding',2,...
'InputBasis',1,'OutputCompounding',2,'OutputBasis',1)
```

```
ParRates =

    0.0011
    0.0030
    0.0064
    0.0142
    0.0201
    0.0251
    0.0309
    0.0330
```

Compute zero curve from the par yield curve.

```
ZeroRates = pyld2zero(ParRates, CurveDates, Settle,'InputCompounding',2,...
'InputBasis',1,'OutputCompounding',2,'OutputBasis',1)
```

```
ZeroRates =

    0.0011
    0.0030
    0.0064
    0.0144
    0.0207
    0.0261
    0.0329
    0.0355
```

Compute Zero Curve Given Par Yield Curve Using datetime Inputs

Use `datetime` inputs to compute the zero curve given the par yield curve.

```
Settle = datenum('01-Feb-2013');

CurveDates = [datenum('01-Feb-2014')
    datenum('01-Feb-2015')
    datenum('01-Feb-2016')
    datenum('01-Feb-2018')
    datenum('01-Feb-2020')
    datenum('01-Feb-2023')
    datenum('01-Feb-2033')
    datenum('01-Feb-2043')];

OriginalParRates = [0.11 0.30 0.64 1.42 2.02 2.51 3.10 3.31]'/100;

InputCompounding = 1;
InputBasis = 0;
OutputCompounding = 1;
OutputBasis = 0;

Settle = datetime(Settle, 'ConvertFrom', 'datenum','Locale','en_US');
CurveDates = datetime(CurveDates, 'ConvertFrom', 'datenum','Locale','en_US');
[ZeroRates Dates] = pyld2zero(OriginalParRates, CurveDates, Settle, ...
'OutputCompounding', OutputCompounding, 'OutputBasis', OutputBasis, ...
'InputCompounding', InputCompounding, 'InputBasis', InputBasis)

ZeroRates =

    0.0011
    0.0030
    0.0064
    0.0144
    0.0207
    0.0261
    0.0329
    0.0356


Dates = 8x1 datetime array
   01-Feb-2014 00:00:00
   01-Feb-2015 00:00:00
   01-Feb-2016 00:00:00
   01-Feb-2018 00:00:00
   01-Feb-2020 00:00:00
   01-Feb-2023 00:00:00
   01-Feb-2033 00:00:00
```

```
    01-Feb-2043 00:00:00
```

Demonstrate a Roundtrip From `pyld2zero` to `zero2pyld`

Given the following a par yield curve and its maturity dates, return the `ZeroRates`.

```
Settle = datenum('01-Feb-2013');

CurveDates = [datenum('01-Feb-2014')
    datenum('01-Feb-2015')
    datenum('01-Feb-2016')
    datenum('01-Feb-2018')
    datenum('01-Feb-2020')
    datenum('01-Feb-2023')
    datenum('01-Feb-2033')
    datenum('01-Feb-2043')];

OriginalParRates = [0.11 0.30 0.64 1.42 2.02 2.51 3.10 3.31]'/100;

InputCompounding = 1;
InputBasis = 0;
OutputCompounding = 1;
OutputBasis = 0;

ZeroRates = pyld2zero(OriginalParRates, CurveDates, Settle, ...
'OutputCompounding', OutputCompounding, 'OutputBasis', OutputBasis, ...
'InputCompounding', InputCompounding, 'InputBasis', InputBasis)

ZeroRates =

    0.0011
    0.0030
    0.0064
    0.0144
    0.0207
    0.0261
    0.0329
    0.0356
```

With the `ZeroRates`, use the `zero2pyld` function to return the `ParRatesOut` and determine the roundtrip error.

```
ParRatesOut = zero2pyld(ZeroRates, CurveDates, Settle, ...
'OutputCompounding', OutputCompounding, 'OutputBasis', OutputBasis, ...
'InputCompounding', InputCompounding, 'InputBasis', InputBasis)
```

```
ParRatesOut =

    0.0011
    0.0030
    0.0064
    0.0142
    0.0202
    0.0251
    0.0310
    0.0331
```

```
max(abs(OriginalParRates - ParRatesOut)) % Roundtrip error
```

```
ans = 1.2750e-16
```

- “Term Structure of Interest Rates” on page 2-45
- “Sensitivity of Bond Prices to Interest Rates” on page 10-3
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-11
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-23

Input Arguments

`ParRates` — Annualized par yields

decimal fraction

Annualized par yields (coupon rates), specified as a `NUMBONDS`-by-`1` vector using decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by `CurveDates`.

Data Types: `double`

`CurveDates` — Maturity dates
serial date number | date character vector | datetime

Maturity dates which correspond to the input `ParRates`, specified as a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`Settle` — Common settlement date for `ZeroRates`
serial date number | date character vector | datetime

Common settlement date for input `ParRates`, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
[ZeroRates,CurveDates] =
pyld2zero(ParRates,CurveDates,Settle,'OutputCompounding',
3,'OutputBasis',5,'InputCompounding',4,'InputBasis',5)
```

`OutputCompounding` — Compounding frequency of output `ZeroRates`
`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of output `ZeroRates`, specified using the allowed values:

- `0` — Simple interest (no compounding)
- `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

Note

- If `OutputCompounding` is set to `0` (simple), `-1` (continuous), or `365` (daily), the `InputCompounding` must also be specified using a valid value.
- If `OutputCompounding` is not specified, then `OutputCompounding` is assigned the value specified for `InputCompounding`.
- If either `OutputCompounding` or `InputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`OutputBasis` — Day-count basis of output `ZeroRates`

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of output `ZeroRates`, specified using allowed values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)

- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `OutputBasis` is not specified, then `OutputBasis` is assigned the value specified for `InputBasis`. If either `InputBasis` or `OutputBasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

`InputCompounding` — Compounding frequency of input `ParRates`

`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of input `ParRates`, specified using allowed values:

- `1` — Annual compounding
- `2` — Semiannual compounding (default)
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note

- If `OutputCompounding` is `1`, `2`, `3`, `4`, `6`, or `12` and `InputCompounding` is not specified, the value of `OutputCompounding` is used.
- If `OutputCompounding` is `0` (simple), `-1` (continuous), or `365` (daily), a valid `InputCompounding` value must also be specified.
- If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`InputBasis` — Day-count basis of input `ParRates`

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of the input `ParRates`, specified using allowed values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `InputBasis` is not specified, then `InputBasis` is assigned the value specified for `OutputBasis`. If either `InputBasis` or `Outputbasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

Output Arguments

`ZeroRates` — Zero rates

numeric

Zero rates, returned as a `NUMBONDS`-by-`1` numeric vector. In aggregate, the rates in `ZeroRates` constitute a zero curve for the investment horizon represented by `CurveDates`. `ZeroRates` are ordered by ascending maturity.

`CurveDates` — Maturity dates that correspond to `ZeroRates`

serial date number | date character vector | datetime

Maturity dates that correspond to the `ZeroRates`, returned as a `NUMBONDS`-by-`1` vector of maturity dates that correspond to each par rate contained in `ZeroRates`. `CurveDates` are ordered by ascending maturity.

If either input for `CurveDates` or `Settle` is a datetime array, `CurveDates` is returned as a datetime array. Otherwise, `CurveDates` are returned as a serial date numbers.

See Also

`datetime` | `datetime` | `disc2zero` | `fwd2zero` | `getForwardRates` | `zbtprice` | `zbtyield` | `zero2disc` | `zero2fwd` | `zero2pyld`

Topics

“Term Structure of Interest Rates” on page 2-45
“Sensitivity of Bond Prices to Interest Rates” on page 10-3
“Bond Prices and Yield Curve Parallel Shifts” on page 10-11
“Bond Prices and Yield Curve Nonparallel Shifts” on page 10-16
“Term Structure Analysis and Interest-Rate Swaps” on page 10-23
“Fixed-Income Terminology” on page 2-25

Introduced before R2006a

quarter

Returns the quarter of given date

Syntax

```
q = quarter(date)
q = quarter( ___ ,month1)
q = quarter(date,month1,dateformat)
```

Description

`q = quarter(date)` returns the quarter of the given `date`, assuming the standard calendar (starting on January 1st).

`q = quarter( ___ ,month1)` returns the quarter of the date for a calendar which starts on the month specified by `month1`. `month1` must be an integer between `1`–`12` representing Jan-Dec respectively.

`q = quarter(date,month1,dateformat)` returns the quarter of the date for a calendar which starts on the month specified by `month1`. `month1` must be an integer between `1`–`12` representing January to December respectively. The `dateformat` input is a character vector to specify the format of your date character vector in case it is not normally recognized by the `datenum` function.

Examples

Determine the Quarter for a Given Date

`quarter` returns the quarter of the given `date`, assuming the standard calendar that starts on Jan 1st.

```
quarter('7-Apr-2013')
```

```
ans = 2
```

Determine the Quarter for a Given Date for an Off-Cycle Calendar

`quarter` returns the quarter of the date for a calendar which starts on the month specified by `month1`.

```
quarter('7-Apr-2013',5)
```

```
ans = 4
```

If the financial calendar starts in May (where `month1 = 5`), April would be the last month and fall in the last quarter.

Determine the Quarter for a Given Date When the Date is Not Recognized by datenum

When using `quarter`, the optional input argument for `dateformat` is a character vector which lets you specify the format of your date character vector in case it isn't normally recognized by the `datenum` function.

```
quarter('07-04-2013',1)
```

```
ans = 3
```

This gives the quarter `3` because by default `datenum` interprets the date as July 7th, 2013.

If you really meant April 7th, 2013, you can use `dateformat` to specify the intended format.

```
quarter('07-04-2013',1,'dd-mm-yyyy')
```

```
ans = 2
```

Input Arguments

`date` — Date in a quarter

serial date number | date character vector

Date in a quarter, specified as serial date numbers or date character vectors.

Data Types: `double` | `char`

`month1` — First month in a calendar
integer with value between `1`–`12`

First month in a calendar, specified as an integer with a value between `1`–`12`, representing January to December respectively. Use `month1` when the standard calendar that starts on January 1st does not apply.

Data Types: `double`

`dateformat` — Format of date
character vector

Format of date, specified as a character vector. `dateformat` input is a character vector to specify the format of your date character vector in case it is not normally recognized by the `datenum` function.

Data Types: `char`

Output Arguments

`q` — Quarter for given date
integer

Quarter for given date, returned as an integer between `1`–`4`.

See Also

`datenum`

Introduced in R2016a

rdivide

Financial time series division

Syntax

```
newfts = tsobj_1 ./ tsobj_2

newfts = tsobj ./ array

newfts = array ./ tsobj
```

Arguments

`tsobj_1, tsobj_2`	Pair of financial time series objects.
`array`	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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

renko

Renko chart

Syntax

```
renko(X)
renko(X,threshold)
```

Arguments

`X`	`X` can be a `M`-by-`2` matrix or a table. If `X` is a `M`-by-`2` matrix, the first column contains date numbers and the second column is the asset price. If `X` is a `M`-by-`2` table, where each column has the same interpretation. However, in the table form, the first column may be serial date numbers, date character vectors, or datetime arrays.
`threshold`	(Optional) Specifies a threshold value for asset price. By default, `threshold` is set to `1`.

Description

`renko(X)` plots asset price with respect to dates.

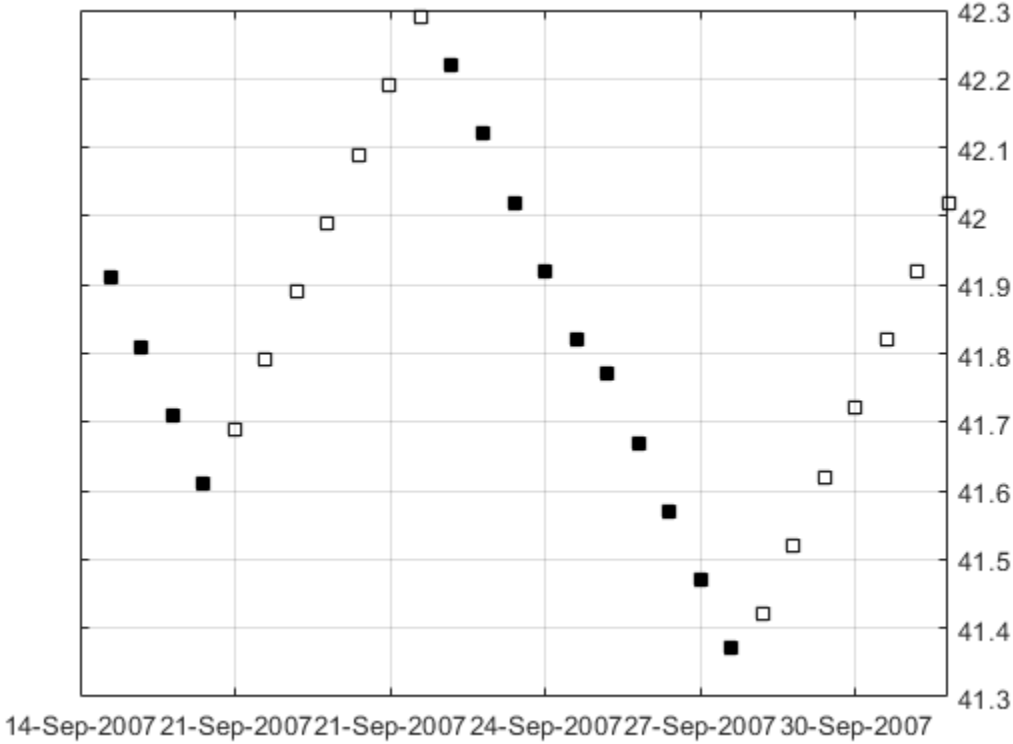
`renko(X,threshold)` plots the asset data, `X`, adding a new box only when the price has changed but at least the value specified by `threshold`.

Examples

Plot Asset Price With Respect to Dates

This example shows how to plot asset price with respect to dates, given asset X as an M-by-2 matrix of date numbers and asset prices, generate a Renko chart.

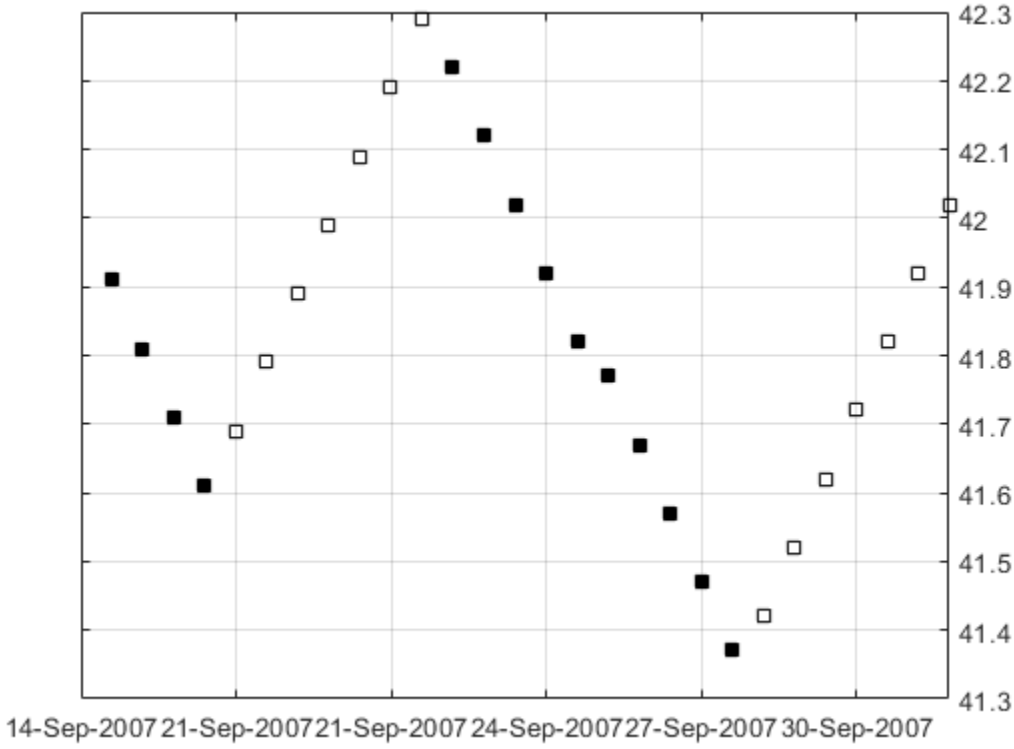
```
X = [...

733299.00          41.99;...
733304.00          41.98;...
733306.00          41.61;...
733307.00          42.29;...
733311.00          41.82;...
733314.00          41.37;...
733318.00          42.02];


renko(X,.1)
```



Plot Asset Price With Respect to Dates Using datetime Input

This example shows how to use `datetime` input to plot asset price with respect to dates, given asset `X` as an `M`-by-2 matrix of date numbers and asset prices, generate a Renko chart.

```
X = [...

733299.00          41.99;...
733304.00          41.98;...
733306.00          41.61;...
733307.00          42.29;...
733311.00          41.82;...
733314.00          41.37;...
733318.00          42.02];

dates = datetime(X(:,1),'ConvertFrom','datenum','Locale','en_US');
data = X(:,2);
t = table(dates,data);
renko(t,0.1)
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `datetime` | `highlow` | `kagi` | `linebreak` | `movavg` | `pointfig` | `priceandvol` | `volarea`

Topics

“Charting Financial Data” on page 2-14

Introduced in R2008a

resamplets

Downsample data

Syntax

```
newfts = resamplets(oldfts,samplestep)
```

Description

`newfts = resamplets(oldfts,samplestep)` downsamples the data contained in the financial time series object `oldfts` every `samplestep` periods. For example, to have the new financial time series object contain every other data element from `oldfts`, set `samplestep` to `2`.

`newfts` is a financial time series object containing the same data series (names) as the input `oldfts`.

See Also

`filter`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

ret2tick

Convert return series to price series

Syntax

```
[TickSeries,TickTimes] = ret2tick(RetSeries,StartPrice,RetIntervals,StartTime,Method)
```

Arguments

`RetSeries`	Number of observations `(NUMOBS)` by number of assets `(NASSETS)` time series array of asset returns associated with the prices in `TickSeries`. The *i*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. `StartTime` can be specified as a serial date number, date character vector, or datetime array. The default is `0`. However, if `StartTime` is a datetime array, `TickTimes` output is a datetime array. Otherwise, `TickTimes` output is an array of serial date numbers.

Method	(Optional) Character vector 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

Convert a Return Series to a Price Series

This example shows how to compute the price increase of two stocks over a year's time based on three incremental return observations.

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05];

RetIntervals = [182
                91
                92];

StartTime = datenum('18-Dec-2000');

[TickSeries,TickTimes] = ret2tick(RetSeries,[],RetIntervals,...
StartTime)

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 = 4x11 char array
    '18-Dec-2000'
    '18-Jun-2001'
    '17-Sep-2001'
    '18-Dec-2001'
```

Convert a Return Series to a Price Series Using datetime Input

This example shows how to use `datetime` input to compute the price increase of two stocks over a year's time based on three incremental return observations.

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05];

RetIntervals = [182
                91
                92];

StartTime = datetime('18-Dec-2000','Locale','en_US');
[TickSeries,TickTimes] = ret2tick(RetSeries,[],RetIntervals,...
StartTime)

TickSeries =

    1.0000    1.0000
    1.1000    1.1200
    1.1550    1.1648
    1.0973    1.2230


TickTimes = 4x1 datetime array
   18-Dec-2000
   18-Jun-2001
   17-Sep-2001
   18-Dec-2001
```

- “Data Transformation and Frequency Conversion” on page 12-12

See Also

`datetime` | `portsim` | `tick2ret`

Topics

“Data Transformation and Frequency Conversion” on page 12-12

Introduced before R2006a

ret2tick (fts)

Convert return series to price series for time series object

Syntax

```
priceFts = ret2tick(returnFts)

priceFts = ret2tick(returnFts,'PARAM1',VALUE1,'PARAM2',VALUE2', ...)
```

Arguments

`returnFts`	Financial time series object of returns.
`'PARAM1'`	(Optional) `StartPrice` is a `Numeric` value and is a scalar or `1`-by-`N` vector of initial prices for each asset. If `StartPrice` is unspecified or empty, the initial price of all assets is `1`.
`'PARAM2'`	(Optional) `StartTime` is `Date` value for a scalar date number or a single date character vector specifying the starting time for the first observation. This date is applied to the price series of all assets. **Note** The first period price value of the resulting price series will not be reported if `StartTime` is not specified. The resulting price series are scaled based on the `StartPrice`, even if `StartTime` is not supplied.
`'PARAM3'`	(Optional) *`Method`* is a character vector indicating the method to convert asset returns to prices. The value must be defined as `'Simple'` (default) or `'Continuous'`. If *`Method`* is `'Simple'`, `ret2tick` uses simple periodic returns. If *`Method`* is `'Continuous'`, the function uses continuously compounded returns. Case is ignored for *`Method`*.

Description

`priceFts = ret2tick(returnFts,'PARAM1',VALUE1,'PARAM2',VALUE2', ...)` generates a financial time series object of prices.

If *`Method`* is unspecified or `'Simple'`, the prices are

```
PriceSeries(i+1) = PriceSeries(i)*[1 + ReturnSeries(i)]
```

If *`Method`* is `'Continuous'`, the prices are

```
PriceSeries(i+1) = PriceSeries(i)*exp[ReturnSeries(i)]
```

Examples

Compute the price series from the following return series:

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05]
```

Use the following dates:

```
Dates = {'18-Jun-2001'; '17-Sep-2001'; '18-Dec-2001'}
```

where

```
ret = fints(Dates, RetSeries)
```

```
ret =
desc:  (none)
freq:  Unknown (0)

'dates:  (3)'    'series1:  (3)'    'series2:  (3)'
'18-Jun-2001'    [       0.1000]    [       0.1200]
'17-Sep-2001'    [       0.0500]    [       0.0400]
'18-Dec-2001'    [      -0.0500]    [       0.0500]
```

`PriceFtS` is computed as:

```
PriceFts = ret2tick(ret, 'StartPrice', 100, 'StartTime', '18-Dec-2000')

PriceFts =
```

```
desc:  (none)
freq:  Unknown (0)

'dates:  (4)'    'series1:  (4)'    'series2:  (4)'
'18-Dec-2000'    [          100]    [          100]
'18-Jun-2001'    [     110.0000]    [     112.0000]
'17-Sep-2001'    [     115.5000]    [     116.4800]
'18-Dec-2001'    [     109.7250]    [     122.3040]
```

See Also

`portsim` | `tick2ret`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

rmfield

Remove data series

Syntax

```
fts = rmfield(tsobj,fieldname)
```

Arguments

`tsobj`	Financial time series object.
`fieldname`	Character vector containing the data series name to remove a single series from the object. Cell array of character vectors for the 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`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

rsindex

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 character vector that represents the valid parameter name.

1 The relative strength factor is calculated by dividing the average of the gains by the average of the losses within a specified time period:

```
RS = (average gains)/(average losses)
```

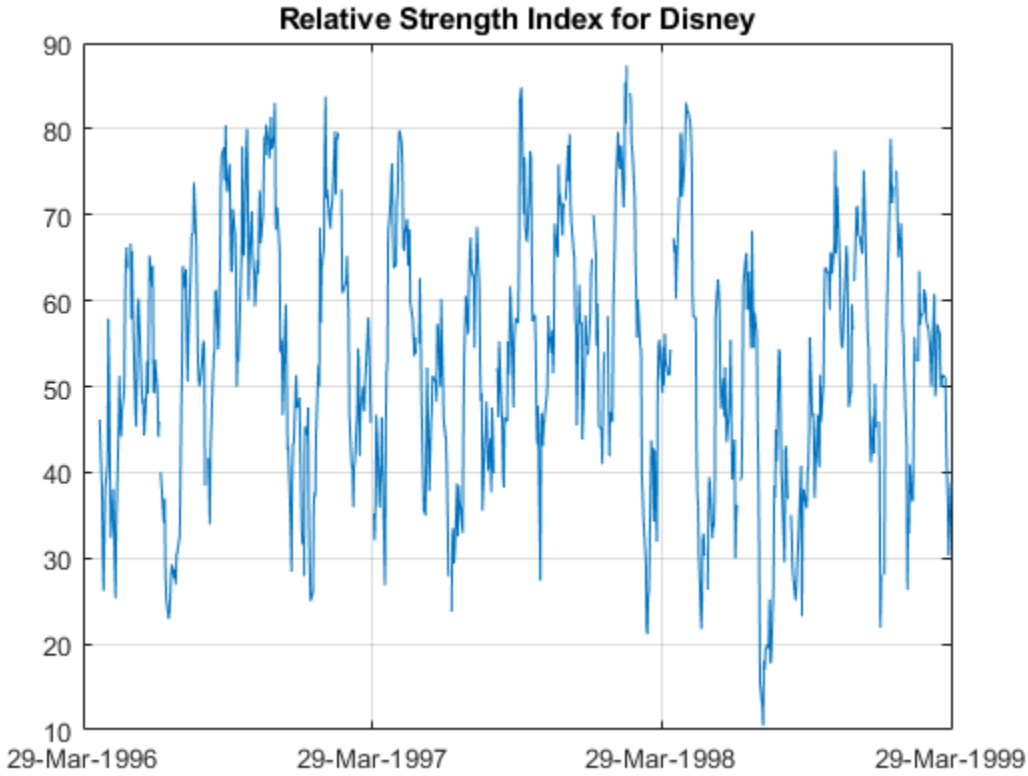
2 The first value of `RSI`, `RISI(1)`, is set as `NaN` to preserve the dimensions of `CLOSEP`.

Examples

Calculate the Relative Strength Index (RSI)

This example shows how to calculate the RSI for Disney stock and plot the results.

```
load disney.mat
dis_RSI = rsindex(dis);
plot(dis_RSI)
legend('off')
title('Relative Strength Index for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Murphy, John J. *Technical Analysis of the Futures Market.* New York Institute of Finance, 1986, pp. 295–302.

See Also

`negvolidx` | `posvolidx`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

sde class

Stochastic Differential Equation (SDE) model

Description

The `sde` constructor creates and displays general stochastic differential equation (`SDE`) models from user-defined drift and diffusion rate functions. Use `sde` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

An `sde` object enables you to simulate any vector-valued SDE of the form:
$dX_t = F(t,X_t)dt + G(t,X_t)dW_t$
where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.
- F is an `NVARS`-by-`1` vector-valued drift-rate function.
- G is an `NVARS`-by-`NBROWNS` matrix-valued diffusion-rate function.

Construction

`SDE = sde(DriftRate,DiffusionRate)` constructs a default `sde` object.

`SDE = sde(DriftRate,DiffusionRate,Name,Value)` constructs a `sde` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `sde` object, see `sde`.

Input Arguments

`DriftRate` — `DriftRate` is a user-defined drift-rate function and represents the parameter *F*

vector or object of class `Drift`

`DriftRate` is a user-defined drift-rate function and represents the parameter *F*, specified as a vector or object of class `drift`.

`DriftRate` is a function that returns an `NVARS`-by-`1` drift-rate vector when called with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Alternatively, `DriftRate` can also be an object of class `drift` that encapsulates the drift-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object. For more information on the `drift` object, see `drift`.

Data Types: `double`

`DiffusionRate` — `DiffusionRate` is a user-defined drift-rate function and represents the parameter *G*

matrix or object of class `Diffusion`

`DiffusionRate` is a user-defined drift-rate function and represents the parameter *G*, specified as a matrix or object of class `diffusion`.

`DiffusionRate` is a function that returns an `NVARS`-by-`NBROWNS` diffusion-rate matrix when called with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Alternatively, `DiffusionRate` can also be an object of class `diffusion` that encapsulates the diffusion-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object. For more information on the `diffusion` object, see `diffusion`.

Data Types: `double`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `sde`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the`drift` constructor) of the form:

$$F(t,X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by $(t, X_t$.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the constructor `diffusion` constructor):

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.

- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the`gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`Simulation` — User-defined simulation function or SDE simulation method

if you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `function_handle`

Methods

The following methods are from the `sde` class.

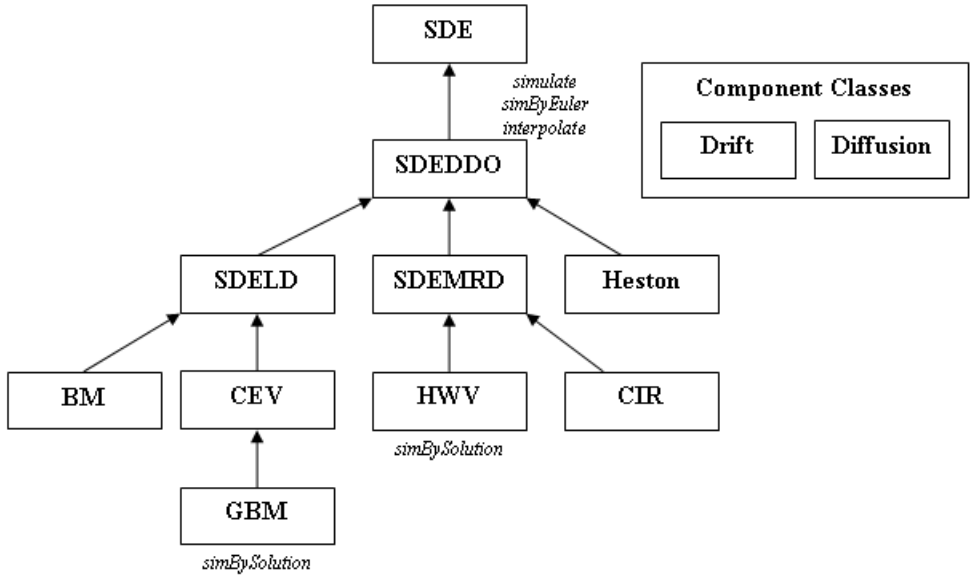
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create an SDE Object

Construct an SDE object obj to represent a univariate geometric Brownian Motion model of the form: $dX_t = 0.1X_t dt + 0.3X_t dW_t$.

Create drift and diffusion functions that are accessible by the common (t, X_t) interface:

```
F = @(t,X) 0.1 * X;
G = @(t,X) 0.3 * X;
```

Pass the functions to the sde constructor to create an object obj of class sde:

```
obj = sde(F, G)     % dX = F(t,X)dt + G(t,X)dW
```

```
obj =
   Class SDE: Stochastic Differential Equation
   ------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

obj displays like a MATLAB® structure, with the following information:

- The object's class
- A brief description of the object
- A summary of the dimensionality of the model

The object's displayed parameters are as follows:

- StartTime: The initial observation time (real-valued scalar)
- StartState: The initial state vector (NVARS-by-1 column vector)
- Correlation: The correlation structure between Brownian process
- Drift: The drift-rate function $F(t, X_t)$
- Diffusion: The diffusion-rate function $G(t, X_t)$

- `Simulation`: The simulation method or function.

Of these displayed parameters, only `Drift` and `Diffusion` are required inputs.

The only exception to the (t, X_t) evaluation interface is `Correlation`. Specifically, when you enter `Correlation` as a function, the SDE engine assumes that it is a deterministic function of time, $C(t)$. This restriction on `Correlation` as a deterministic function of time allows Cholesky factors to be computed and stored before the formal simulation. This inconsistency dramatically improves run-time performance for dynamic correlation structures. If `Correlation` is stochastic, you can also include it within the simulation architecture as part of a more general random number generation function.

- "Representing Market Models Using SDE Objects" on page 17-34
- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time *t* and a

state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sde` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y., *"Testing Continuous-Time Models of the Spot Interest Rate"*, The Review of Financial Studies, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y., *"Transition Densities for Interest Rate and Other Nonlinear Diffusions"*, The Journal of Finance, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeld` | `simByEuler` | `simulate`

Topics

"Representing Market Models Using SDE Objects" on page 17-34
"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
Class Attributes (MATLAB)

Property Attributes (MATLAB)
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

sde

Construct SDE model from user-specified functions

Syntax

```
SDE = sde(DriftRate, DiffusionRate)

SDE = sde(DriftRate, DiffusionRate, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`sde`

Description

This constructor creates and displays general stochastic differential equation (`SDE`) models from user-defined drift and diffusion rate functions. Use `sde` objects to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

This constructor enables you to simulate any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.
- F is an `NVARS`-by-`1` vector-valued drift-rate function.
- G is an `NVARS`-by-`NBROWNS` matrix-valued diffusion-rate function.

Input Arguments

`DriftRate`	User-defined drift-rate function, denoted by F. `DriftRate` is a function that returns an `NVARS`-by-`1` drift-rate vector when called with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t. Alternatively, `DriftRate` may also be an object of class `Drift` that encapsulates the drift-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object.
`DiffusionRate`	User-defined diffusion-rate function, denoted by G. `DiffusionRate` is a function that returns an `NVARS`-by-`NBROWNS` diffusion-rate matrix when called with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t. Alternatively, `DiffusionRate` may also be an object of class `Diffusion` that encapsulates the diffusion-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sde` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sde` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sde` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE`	Stochastic differential equation model (`SDE`) with the following parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Simulation`: A simulation function or method

Examples

- "Base SDE Models" on page 17-16
- Representing Market Models Using SDE Objects on page 17-34

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally

specified an input as an array, `sde` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5

"Performance Considerations" on page 17-76

Introduced in R2008a

sdeddo class

Stochastic Differential Equation (SDE) model from Drift and Diffusion components

Description

The `sdeddo` constructor creates and displays `sdeddo` objects, instantiated with objects of class `drift` and `diffusion`. These restricted `sdeddo` objects contain the input `drift` and `diffusion` objects; therefore, you can directly access their displayed parameters.

This abstraction also generalizes the notion of drift and diffusion-rate objects as functions that `sdeddo` evaluates for specific values of time t and state X_t. Like`sde` objects, `sdeddo` objects allow you to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `sdeddo` object enables you to simulate any vector-valued SDE of the form:

$$dX_t = F(t,X_t)dt + G(t,X_t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.
- F is an `NVARS`-by-`1` vector-valued drift-rate function.
- G is an `NVARS`-by-`NBROWNS` matrix-valued diffusion-rate function.

Construction

`SDE = sdeddo(DriftRate,DiffusionRate)` constructs a default `sdeddo` object.

`SDE = sdeddo(DriftRate,DiffusionRate,Name,Value)` constructs a `sdeddo` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `sdeddo` object, see `sdeddo`.

Input Arguments

`DriftRate` — `DriftRate` is a user-defined drift-rate function and represents the parameter *F*

vector or object of class `Drift`

`DriftRate` is a user-defined drift-rate function and represents the parameter *F*, specified as a vector or object of class `drift`.

`DriftRate` is a function that returns an `NVARS`-by-`1` drift-rate vector when called with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Alternatively, `DriftRate` can also be an object of class `drift` that encapsulates the drift-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object. For more information on the `drift` object, see `drift`.

Data Types: `double`

`DiffusionRate` — `DiffusionRate` is a user-defined drift-rate function and represents the parameter *G*

matrix or object of class `Diffusion`

`DiffusionRate` is a user-defined drift-rate function and represents the parameter *G*, specified as a matrix or object of class `diffusion`.

`DiffusionRate` is a function that returns an `NVARS`-by-`NBROWNS` diffusion-rate matrix when called with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Alternatively, `DiffusionRate` can also be an object of class `diffusion` that encapsulates the diffusion-rate specification. In this case, however, `sde` uses only the `Rate` parameter of the object. For more information on the `diffusion` object, see `diffusion`.

Data Types: `double`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `sdeddo`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)

value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the`drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$

- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by $(t, X_t$.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the constructor `diffusion` constructor):

$$G(t,X_t) = D(t,X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the`gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`Simulation` — User-defined simulation function or SDE simulation method

if you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `function_handle`

Methods

Inherited Methods

The following methods are inherited from the `sde` class.

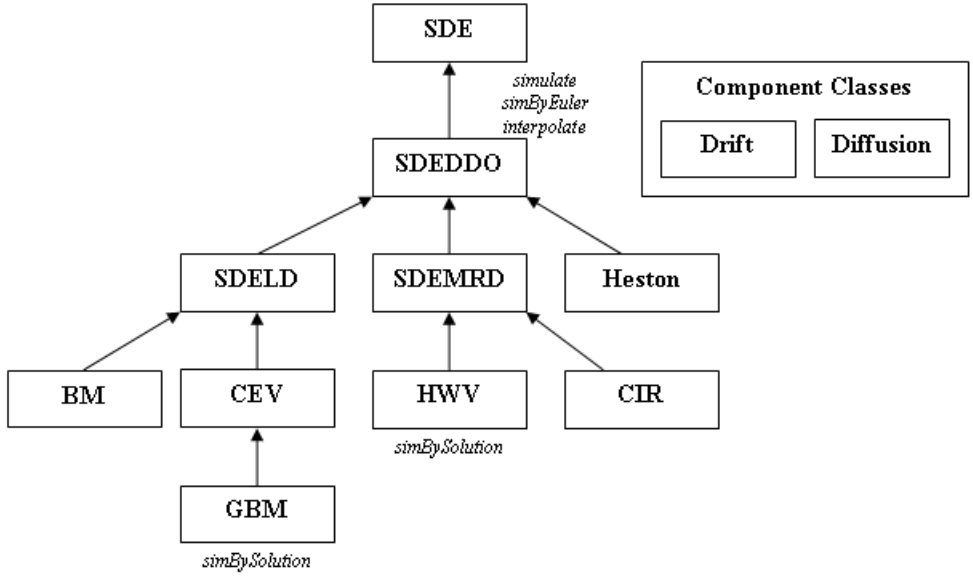
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a `sdeddo` Object

The `sdeddo` class derives from the base `sde` class. To use this class, you must pass drift and diffusion-rate objects to the `sdeddo` constructor. Create `drift` and `diffusion` rate objects:

```
F = drift(0, 0.1);       % Drift rate function F(t,X)
G = diffusion(1, 0.3);   % Diffusion rate function G(t,X)
```

Pass the functions to the `sdeddo` constructor to create an object `obj` of class `sdeddo`:

```
obj = sdeddo(F, G)       % dX = F(t,X)dt + G(t,X)dW
```

```
obj =
   Class SDEDDO: SDE from Drift and Diffusion Objects
   ---------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ---------------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
              A: 0
              B: 0.1
          Alpha: 1
          Sigma: 0.3
```

In this example, the object displays the additional parameters associated with input `drift` and `diffusion` objects.

- “Representing Market Models Using SDEDDO Objects” on page 17-36

- "Representing Market Models Using SDE Objects" on page 17-34
- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdeddo` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y., *"Testing Continuous-Time Models of the Spot Interest Rate"* , The Review of Financial Studies, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y., *"Transition Densities for Interest Rate and Other Nonlinear Diffusions"* , The Journal of Finance, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `interpolate` | `sdeld` | `simByEuler` | `simulate`

Topics

“Representing Market Models Using SDEDDO Objects” on page 17-36
“Representing Market Models Using SDE Objects” on page 17-34
“Simulating Equity Prices” on page 17-34
“Simulating Interest Rates” on page 17-59
“Stratified Sampling” on page 17-70
“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

sdeddo

Construct `sdeddo` model from Drift and Diffusion objects

Syntax

```
SDE = sdeddo(DriftRate, DiffusionRate)

SDE = sdeddo(DriftRate, DiffusionRate, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`sdeddo`

Description

This constructor creates and displays `sdeddo` objects, specifically instantiated with objects of class`drift` and `diffusion`. These restricted `sdeddo` objects contain the input `drift` and `diffusion` objects; therefore, you can directly access their displayed parameters.

This abstraction also generalizes the notion of drift and diffusion-rate objects as functions that `sdeddo` evaluates for specific values of time t and state X_t. Like`sde` objects, `sdeddo` objects allow you to simulate sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

This method enables you to simulate any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.
- F is an `NVARS`-by-`1` vector-valued drift-rate function.
- G is an `NVARS`-by-`NBROWNS` matrix-valued diffusion-rate function.

Input Arguments

`DriftRate`	Object of class`drift` that encapsulates a user-defined drift-rate specification, represented as F.
`DiffusionRate`	Object of class `diffusion` that encapsulates a user-defined diffusion-rate specification, represented as G.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.

`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sdeddo` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sdeddo` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sdeddo` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE`	Object of class `sdeddo` with the following parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `A`: Access function for the drift-rate property `A`, callable as a function of time and state • `B`: Access function for the drift-rate property `B`, callable as a function of time and state • `Alpha`: Access function for the diffusion-rate property `Alpha`, callable as a function of time and state • `Sigma`: Access function for the diffusion-rate property `Sigma`, callable as a function of time and state • `Simulation`: A simulation function or method

Examples

- "Drift and Diffusion Models" on page 17-19
- Representing Market Models Using SDEDDO Objects on page 17-36

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like

simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdeddo` treats as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sde`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84

"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

sdeld class

SDE with Linear Drift model

Description

The `sdeld` constructor creates and displays SDE objects whose drift rate is expressed in linear drift-rate form and that derive from the `sdeddo` (SDE from drift and diffusion objects class).

Use `sdeld` objects to simulate sample paths of `NVARS` state variables expressed in linear drift-rate form. They provide a parametric alternative to the mean-reverting drift form (see `sdemrd`).

These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes with linear drift-rate functions.

The `sdeld` object allows you to simulate any vector-valued SDE of the form:

$$dX_t = (A(t) + B(t)X_t)dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- A is an `NVARS`-by-`1` vector.
- B is an `NVARS`-by-`NVARS` matrix.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Construction

`SDE = sdeld(A,B,Alpha,Sigma)` constructs a default `sdeld` object.

`SDE = sdeld(A,B,Alpha,Sigma,Name,Value)` constructs a `sdeld` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `sdeld` object, see `sdeld`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`A` — `A` represents the parameter *A*

array or deterministic function of time or deterministic function of time and state

`A` represents the parameter *A*, specified as an array or deterministic function of time.

If you specify `A` as an array, it must be an `NVARS`-by-`1` column vector of intercepts.

As a deterministic function of time, when `A` is called with a real-valued scalar time `t` as its only input, `A` must produce an `NVARS`-by-`1` column vector. If you specify `A` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of intercepts when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`B` — `B` represents the parameter *B*

array or deterministic function of time or deterministic function of time and state

`B` represents the parameter *B*, specified as an array or deterministic function of time.

If you specify `A` as an array, it must be an `NVARS`-by-`NVARS` matrix of state vector coefficients.

As a deterministic function of time, when `B` is called with a real-valued scalar time `t` as its only input, `B` must produce an `NVARS`-by-`NVARS` matrix. If you specify `B` as a function of time and state, it must generate an `NVARS`-by-`NVARS` matrix of state vector coefficients when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Alpha` — `Alpha` represents the parameter *D*

array or deterministic function of time or deterministic function of time and state

`Alpha` represents the parameter *D*, specified as an array or deterministic function of time.

If you specify `Alpha` as an array, it represents an `NVARS`-by-`1` column vector of exponents.

As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` matrix.

If you specify it as a function of time and state, `Alpha` must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter *V*
array or deterministic function of time or deterministic function of time and state

`Sigma` represents the parameter *V*, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time *t*.
- An `NVARS`-by-`1` state vector X_t.

Although the`gbm` constructor enforces no restrictions on the sign of `Sigma` volatilities, they are specified as positive values.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `cev`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by $(t, X_t$.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the `gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`Simulation` — User-defined simulation function or SDE simulation method
if you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `function_handle`

Methods

Inherited Methods

The following methods are inherited from the `sde` class.

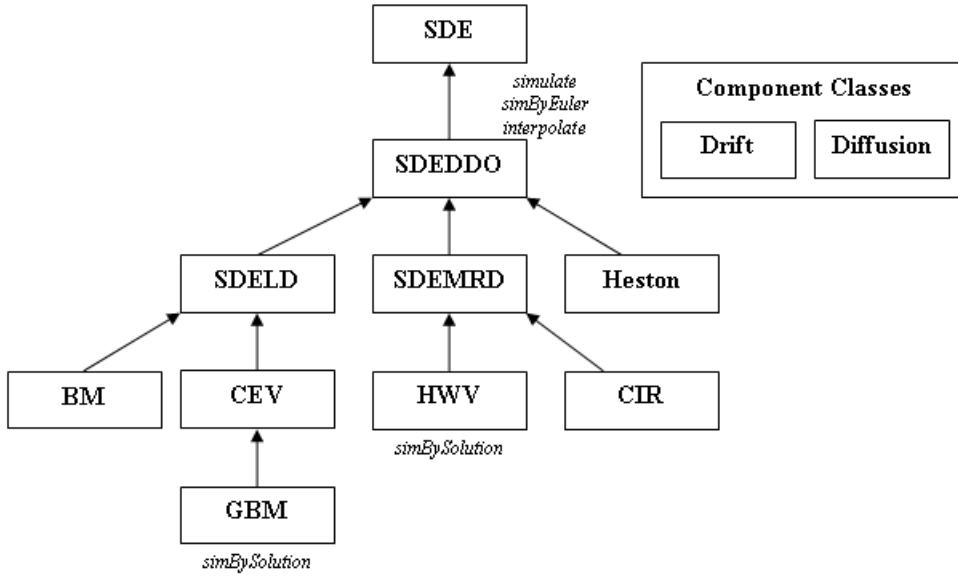
`interpolate`

`simulate`

`simByEuler`

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a sdeld Object

The `sdeld` class derives from the `sdeddo` class. These objects allow you to simulate correlated paths of `NVARS` state variables expressed in linear drift-rate form:

$$dX_t = (A(t) + B(t)X_t)dt + D(t, X_t^{\alpha(t)})V(t)dW_t.$$

```
obj = sdeld(0, 0.1, 1, 0.3) % (A, B, Alpha, Sigma)
```

```
obj =
   Class SDELD: SDE with Linear Drift
```

```
----------------------------------------
  Dimensions: State = 1, Brownian = 1
----------------------------------------
   StartTime: 0
  StartState: 1
 Correlation: 1
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
           A: 0
           B: 0.1
       Alpha: 1
       Sigma: 0.3
```

`sdeld` objects provide a parametric alternative to the mean-reverting drift form and also provide an alternative interface to the `sdeddo` parent class, because you can create an object without first having to create its drift and diffusion-rate components.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdeld` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y., *"Testing Continuous-Time Models of the Spot Interest Rate"* , The Review of Financial Studies, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y., *"Transition Densities for Interest Rate and Other Nonlinear Diffusions"* , The Journal of Finance, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeddo` | `simByEuler`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25

Class Attributes (MATLAB)
Property Attributes (MATLAB)
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

sdeld

Construct stochastic differential equation from linear drift-rate models

Syntax

```
SDE = sdeld(A, B, Alpha, Sigma)

SDE = sdeld(A, B, Alpha, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`sdeld`

Description

This constructor creates and displays SDE objects whose drift rate is expressed in linear drift-rate form and that derive from the `sdeddo` (SDE from drift and diffusion objects) class.

Use `SDELD` objects to simulate sample paths of `NVARS` state variables expressed in linear drift-rate form. They provide a parametric alternative to the mean-reverting drift form (see `sdemrd`).

These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes with linear drift-rate functions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = (A(t) + B(t)X_t)dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- A is an `NVARS`-by-`1` vector.
- B is an `NVARS`-by-`NVARS` matrix.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`A`	`A` represents the parameter A. If you specify `A` as an array, it must be an `NVARS`-by-`1` column vector of intercepts. As a deterministic function of time, when `A` is called with a real-valued scalar time `t` as its only input, `A` must produce an `NVARS`-by-`1` column vector. If you specify `A` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of intercepts when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.
`B`	`B` represents the parameter B. If you specify `B` as an array, it must be an `NVARS`-by-`NVARS` matrix of state vector coefficients. As a deterministic function of time, when `B` is called with a real-valued scalar time `t` as its only input, `B` must produce an `NVARS`-by-`NVARS` matrix. If you specify `B` as a function of time and state, it must generate an `NVARS`-by-`NVARS` matrix of state vector coefficients when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.
`Alpha`	`Alpha` determines the format of the parameter D. If you specify `Alpha` as an array, it represents an `NVARS`-by-`1` column vector of exponents. As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` column vector. If you specify it as a function of time and state, it must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.

`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it represents is an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify it as a function of time and state, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of `Alpha` or `Sigma`, each parameter is specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.

`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sdeld` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sdeld` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sdeld` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.
`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE`	Object of class `sdeld` with the following parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `A`: Access function for the input argument `A`, callable as a function of time and state • `B`: Access function for the input argument `B`, callable as a function of time and state • `Alpha`: Access function for the input argument `Alpha`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state • `Simulation`: A simulation function or method

Examples

- “Linear Drift Models” on page 17-23
- Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects on page 17-37

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdeld` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59

"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

sdemrd class

SDE with Mean-Reverting Drift model

Description

The `sdemrd` constructor creates and displays SDE objects whose drift rate is expressed in mean-reverting drift-rate form and which derive from the `sdeddo` class (SDE from drift and diffusion objects). Use `sdemrd` objects to simulate of sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form, and provide a parametric alternative to the linear drift form (see `sdeld`). These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes with mean-reverting drift-rate functions.

The `sdemrd` object allows you to simulate any vector-valued SDE of the form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- S is an `NVARS`-by-`NVARS` matrix of mean reversion speeds.
- L is an `NVARS`-by-`1` vector of mean reversion levels.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of α.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Construction

`SDE = sdemrd(Speed,Level,Alpha,Sigma)` constructs a default `sdemrd` object.

`SDE = sdemrd(Speed,Level,Alpha,Sigma,Name,Value)` constructs a `sdemrd` object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is its corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

For more information on constructing a `sdemrd` object, see `sdemrd`.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time *t* and state *X(t)* and is invoked with both input arguments.

`Speed` — `Speed` represents the parameter *S*

array or deterministic function of time or deterministic function of time and state

`Speed` represents the parameter *S*, specified as an array or deterministic function of time.

If you specify `Speed` as an array, it must be an `NVARS`-by-`NVARS` matrix of mean-reversion speeds (the rate at which the state vector reverts to its long-run average `Level`).

As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed`

as a function of time and state, it calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Level` — `Level` represents the parameter L
array or deterministic function of time or deterministic function of time and state

`Level` represents the parameter L, specified as an array or deterministic function of time.

If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels.

As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1` column vector. If you specify `Level` as a function of time and state, it must generate an `NVARS`-by-`1` column vector of reversion levels when called with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Alpha` — `Alpha` represents the parameter D
array or deterministic function of time or deterministic function of time and state

`Alpha` represents the parameter D, specified as an array or deterministic function of time.

If you specify `Alpha` as an array, it represents an `NVARS`-by-`1` column vector of exponents.

As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` matrix.

If you specify it as a function of time and state, `Alpha` must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

`Sigma` — `Sigma` represents the parameter V
array or deterministic function of time or deterministic function of time and state

`Sigma` represents the parameter V, specified as an array or a deterministic function of time.

If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` matrix of instantaneous volatility rates or as a deterministic function of time. In this case, each row of `Sigma` corresponds to a particular state variable. Each column corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty.

As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify `Sigma` as a function of time and state, it must return an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs:

- A real-valued scalar observation time t.
- An `NVARS`-by-`1` state vector X_t.

Data Types: `double` | `function_handle`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For more information on using optional name-value arguments, see `sdemrd`.

Properties

`Drift` — Drift rate component of continuous-time stochastic differential equations (SDEs)
value stored from drift-rate function (default) | drift object or function accessible by (t, X_t)

Drift rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t).

The drift rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `drift` class allows you to create drift-rate objects (using the `drift` constructor) of the form:

$$F(t, X_t) = A(t) + B(t)X_t$$

where:

- `A` is an `NVARS`-by-`1` vector-valued function accessible using the (t, X_t) interface.
- `B` is an `NVARS`-by-`NVARS` matrix-valued function accessible using the (t, X_t) interface.

The `drift` object's displayed parameters are:

- `Rate`: The drift-rate function, $F(t,X_t)$
- `A`: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B`: The first order term, $B(t,X_t)$, of $F(t,X_t)$

`A` and `B` enable you to query the original inputs. The function stored in `Rate` fully encapsulates the combined effect of `A` and `B`.

When specified as MATLAB double arrays, the inputs `A` and `B` are clearly associated with a linear drift rate parametric form. However, specifying either `A` or `B` as a function allows you to customize virtually any drift rate specification.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `F = drift(0, 0.1) % Drift rate function F(t,X)`

Attributes:

SetAccess	private

`GetAccess`	`public`

Data Types: `struct` | `double`

`Diffusion` — Diffusion rate component of continuous-time stochastic differential equations (SDEs)

value stored from diffusion-rate function (default) | diffusion object or functions accessible by (t, X_t)

Diffusion rate component of continuous-time stochastic differential equations (SDEs), specified as a drift object or function accessible by (t, X_t.

The diffusion rate specification supports the simulation of sample paths of `NVARS` state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

The `diffusion` class allows you to create diffusion-rate objects (using the `diffusion` constructor):

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t)$$

where:

- `D` is an `NVARS`-by-`NVARS` diagonal matrix-valued function.
- Each diagonal element of `D` is the corresponding element of the state vector raised to the corresponding element of an exponent `Alpha`, which is an `NVARS`-by-`1` vector-valued function.
- `V` is an `NVARS`-by-`NBROWNS` matrix-valued volatility rate function `Sigma`.
- `Alpha` and `Sigma` are also accessible using the (t, X_t) interface.

The `diffusion` object's displayed parameters are:

- `Rate`: The diffusion-rate function, $G(t,X_t)$.
- `Alpha`: The state vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$.
- `Sigma`: The volatility rate, $V(t,X_t)$, of $G(t,X_t)$.

`Alpha` and `Sigma` enable you to query the original inputs. (The combined effect of the individual `Alpha` and `Sigma` parameters is fully encapsulated by the function stored in `Rate`.) The `Rate` functions are the calculation engines for the `drift` and `diffusion` objects, and are the only parameters required for simulation.

Note You can express `drift` and `diffusion` classes in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: `G = diffusion(1, 0.3) % Diffusion rate function G(t,X)`

Attributes:

`SetAccess`	`private`
`GetAccess`	`public`

Data Types: `struct` | `double`

`StartTime` — Starting time of first observation, applied to all state variables

`0` (default) | scalar

Starting time of first observation, applied to all state variables, specified as a scalar

Attributes:

`SetAccess`	`public`
`GetAccess`	`public`

Data Types: `double`

`StartState` — Initial values of state variables

`1` (default) | scalar, column vector, or matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If `StartState` is a scalar, the `gbm` constructor applies the same initial value to all state variables on all trials.

If `StartState` is a column vector, the `gbm` constructor applies a unique initial value to each state variable on all trials.

If `StartState` is a matrix, the `gbm` constructor applies a unique initial value to each state variable on each trial.

Attributes:

SetAccess	public
GetAccess	public

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
if you do not specify a value for Simulation, the default method is simulation by Euler approximation (simByEuler) (default) | function or SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Attributes:

SetAccess	public
GetAccess	public

Data Types: function_handle

Methods

Inherited Methods

The following methods are inherited from thesde class.

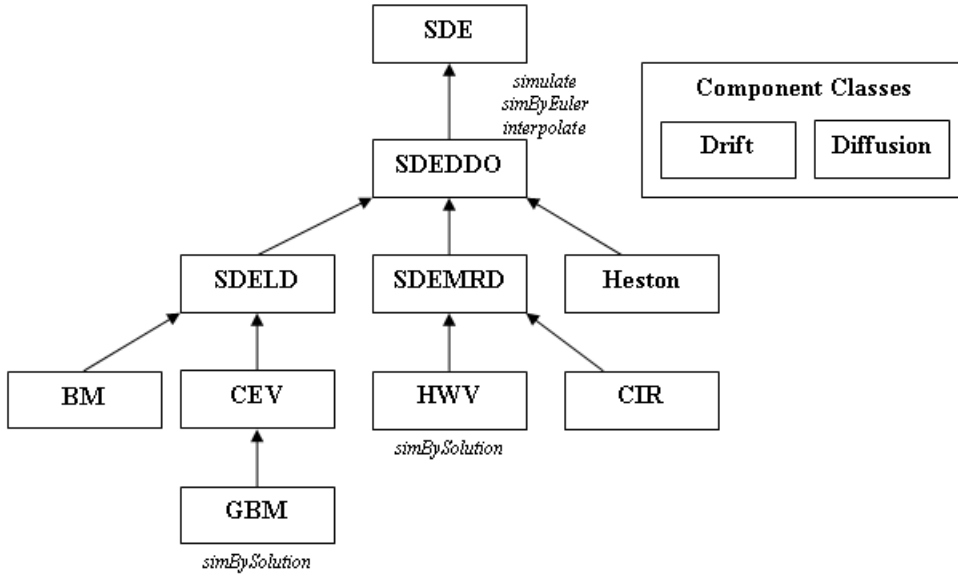
interpolate

simulate

simByEuler

Instance Hierarchy

The following figure illustrates the inheritance relationships among SDE classes.



For more information, see “SDE Class Hierarchy” on page 17-5.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a sdemrd Object

The `sdemrd` class derives directly from the `sdeddo` class. It provides an interface in which the drift-rate function is expressed in mean-reverting drift form:
$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\alpha(t)})V(t)dW_t$.

`sdemrd` objects provide a parametric alternative to the linear drift form by reparameterizing the general linear drift such that: $A(t) = S(t)L(t), B(t) = -S(t)$.

Create an `sdemrd` object `obj` with a square root exponent to represent the model:

$$dX_t = 0.2(0.1 - X_t)dt + 0.05X_t^{\frac{1}{2}}dW_t.$$

```
obj = sdemrd(0.2, 0.1, 0.5, 0.05)   % (Speed, Level, Alpha, Sigma)
```

```
obj =
   Class SDEMRD: SDE with Mean-Reverting Drift
   -------------------------------------------
     Dimensions: State = 1, Brownian = 1
   -------------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Alpha: 0.5
          Sigma: 0.05
          Level: 0.1
          Speed: 0.2
```

`sdemrd` objects display the familiar `Speed` and `Level` parameters instead of `A` and `B`.

- "Simulating Equity Prices" on page 17-34
- "Simulating Interest Rates" on page 17-59
- "Stratified Sampling" on page 17-70
- "Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
- "Base SDE Models" on page 17-16
- "Drift and Diffusion Models" on page 17-19
- "Linear Drift Models" on page 17-23
- "Parametric Models" on page 17-25

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdemrd` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y., *"Testing Continuous-Time Models of the Spot Interest Rate"* , The Review of Financial Studies, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y., *"Transition Densities for Interest Rate and Other Nonlinear Diffusions"*, The Journal of Finance, Vol. 54, No. 4, August 1999.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, New York: Springer-Verlag, 2004.

Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

Shreve, S. E., *Stochastic Calculus for Finance II: Continuous-Time Models*, New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeddo` | `simByEuler`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59

“Stratified Sampling” on page 17-70
“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
Class Attributes (MATLAB)
Property Attributes (MATLAB)
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

sdemrd

Construct stochastic differential equation from mean-reverting drift-rate models

Syntax

```
SDE = sdemrd(Speed, Level, Alpha, Sigma)

SDE = sdemrd(Speed, Level, Alpha, Sigma, 'Name1', Value1, 'Name2',
Value2, ...)
```

Class

`sdemrd`

Description

This constructor creates and displays SDE objects whose drift rate is expressed in mean-reverting drift-rate form and which derive from the`sdeddo` class (SDE from drift and diffusion objects). Use `sdemrd` objects to simulate of sample paths of `NVARS` state variables expressed in mean-reverting drift-rate form, and provide a parametric alternative to the linear drift form (see `sdeld`). These state variables are driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes with mean-reverting drift-rate functions.

This method allows you to simulate any vector-valued SDE of the form:

$$dX_t = S(t)[L(t) - X_t]dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

where:

- X_t is an `NVARS`-by-`1` state vector of process variables.
- S is an `NVARS`-by-`NVARS` matrix of mean reversion speeds.

- L is an `NVARS`-by-`1` vector of mean reversion levels.
- D is an `NVARS`-by-`NVARS` diagonal matrix, where each element along the main diagonal is the corresponding element of the state vector raised to the corresponding power of a.
- V is an `NVARS`-by-`NBROWNS` instantaneous volatility rate matrix.
- dW_t is an `NBROWNS`-by-`1` Brownian motion vector.

Input Arguments

Specify required input parameters as one of the following types:

- A MATLAB array. Specifying an array indicates a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- A MATLAB function. Specifying a function provides indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported via an interface, because all implementation details are hidden and fully encapsulated by the function.

Note You can specify combinations of array and function input parameters as needed.

Moreover, a parameter is identified as a deterministic function of time if the function accepts a scalar time `t` as its only input argument. Otherwise, a parameter is assumed to be a function of time t and state *X(t)* and is invoked with both input arguments.

The required input parameters are:

`Speed`	`Speed` represents the parameter S. If you specify `Speed` as an array, it represents an `NVARS`-by-`NVARS` 2-dimensional matrix of mean-reversion speeds (the rate or speed at which the state vector reverts to its long-run average `Level`). As a deterministic function of time, when `Speed` is called with a real-valued scalar time `t` as its only input, `Speed` must produce an `NVARS`-by-`NVARS` matrix. If you specify `Speed` as a function of time and state, `Speed` calculates the speed of mean reversion. This function must generate an `NVARS`-by-`NVARS` matrix of reversion rates when called with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.
`Level`	`Level` represents the parameter L. If you specify `Level` as an array, it must be an `NVARS`-by-`1` column vector of reversion levels. As a deterministic function of time, when `Level` is called with a real-valued scalar time `t` as its only input, `Level` must produce an `NVARS`-by-`1`column vector. If you specify `Level` as a function of time and state, must generate an `NVARS`-by-`1` column vector of reversion levels when invoked with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t.

`Alpha`	`Alpha` determines the format of the parameter *D*. If you specify `Alpha` as an array, it must be an `NVARS`-by-`1` column vector of exponents. As a deterministic function of time, when `Alpha` is called with a real-valued scalar time `t` as its only input, `Alpha` must produce an `NVARS`-by-`1` column vector. If you specify it as a function of time and state, it must return an `NVARS`-by-`1` column vector of exponents when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.
`Sigma`	`Sigma` represents the parameter *V*. If you specify `Sigma` as an array, it must be an `NVARS`-by-`NBROWNS` 2-dimensional matrix of instantaneous volatility rates. In this case, each row of `Sigma` corresponds to a particular state variable. Each column of `Sigma` corresponds to a particular Brownian source of uncertainty, and associates the magnitude of the exposure of state variables with sources of uncertainty. As a deterministic function of time, when `Sigma` is called with a real-valued scalar time `t` as its only input, `Sigma` must produce an `NVARS`-by-`NBROWNS` matrix. If you specify it as a function of time and state, it must generate an `NVARS`-by-`NBROWNS` matrix of volatility rates when invoked with two inputs: • A real-valued scalar observation time *t*. • An `NVARS`-by-`1` state vector X_t.

Note Although the constructor does not enforce restrictions on the signs of these input arguments, each argument is specified as a positive value.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`StartTime`	Scalar starting time of the first observation, applied to all state variables. If you do not specify a value for `StartTime`, the default is `0`.
`StartState`	Scalar, `NVARS`-by-`1` column vector, or `NVARS`-by-`NTRIALS` matrix of initial values of the state variables. If `StartState` is a scalar, `sdemrd` applies the same initial value to all state variables on all trials. If `StartState` is a column vector, `sdemrd` applies a unique initial value to each state variable on all trials. If `StartState` is a matrix, `sdemrd` applies a unique initial value to each state variable on each trial. If you do not specify a value for `StartState`, all variables start at `1`.

`Correlation`	Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes). Specify `Correlation` as an `NBROWNS`-by-`NBROWNS` positive semidefinite matrix, or as a deterministic function *C(t)* that accepts the current time *t* and returns an `NBROWNS`-by-`NBROWNS` positive semidefinite correlation matrix. A `Correlation` matrix represents a static condition. As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure. If you do not specify a value for `Correlation`, the default is an `NBROWNS`-by-`NBROWNS` identity matrix representing independent Gaussian processes.
`Simulation`	A user-defined simulation function or SDE simulation method. If you do not specify a value for `Simulation`, the default method is simulation by Euler approximation (`simByEuler`).

Output Arguments

`SDE`	Object of class `SDEMRD`, with the following parameters: • `StartTime`: Initial observation time • `StartState`: Initial state at time `StartTime` • `Correlation`: Access function for the `Correlation` input argument, callable as a function of time • `Drift`: Composite drift-rate function, callable as a function of time and state • `Diffusion`: Composite diffusion-rate function, callable as a function of time and state • `Speed`: Access function for the input argument `Speed`, callable as a function of time and state • `Level`: Access function for the input argument `Level`, callable as a function of time and state • `Alpha`: Access function for the input argument `Alpha`, callable as a function of time and state • `Sigma`: Access function for the input argument `Sigma`, callable as a function of time and state • `Simulation`: A simulation function or method

Examples

See “Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models” on page 17-28.

Algorithms

When you specify the required input parameters as arrays, they are associated with a specific parametric form. By contrast, when you specify either required input parameter as a function, you can customize virtually any specification.

Accessing the output parameters with no inputs simply returns the original input specification. Thus, when you invoke these parameters with no inputs, they behave like

simple properties and allow you to test the data type (double vs. function, or equivalently, static vs. dynamic) of the original input specification. This is useful for validating and designing methods.

When you invoke these parameters with inputs, they behave like functions, giving the impression of dynamic behavior. The parameters accept the observation time t and a state vector X_t, and return an array of appropriate dimension. Even if you originally specified an input as an array, `sdemrd` treats it as a static function of time and state, by that means guaranteeing that all parameters are accessible by the same interface.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`diffusion` | `drift` | `sdeddo`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84

"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

simByEuler

Euler simulation of stochastic differential equations (SDEs)

Syntax

```
[Paths, Times, Z] = simByEuler(MDL, NPERIODS)

[Paths, Times, Z] = simByEuler(MDL, NPERIODS, 'Name1', Value1,
'Name2', Value2, ...)
```

Classes

All classes in the "SDE Class Hierarchy" on page 17-5.

Description

This method simulates any vector-valued SDE of the form
$dX_t = F(t,X_t)dt + G(t,X_t)dW_t$

where:

- X is an *NVARS*-by-`1` state vector of process variables (for example, short rates or equity prices) to simulate.
- W is an *NBROWNS*-by-`1` Brownian motion vector.
- F is an *NVARS*-by-`1` vector-valued drift-rate function.
- G is an *NVARS*-by-*NBROWNS* matrix-valued diffusion-rate function.

`simByEuler` simulates `NTRIALS` sample paths of `NVARS` correlated state variables driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, using the Euler approach to approximate continuous-time stochastic processes.

Input Arguments

`MDL`	Stochastic differential equation object created with the `sdeddo` constructor.
`NPERIODS`	Positive scalar integer number of simulation periods. The value of `NPERIODS` determines the number of rows of the simulated output series.

Optional Input Arguments

Specify optional inputs as matching parameter name/value pairs as follows:

- Specify the parameter name as a character vector, followed by its corresponding value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`NTRIALS`	Positive scalar integer number of simulated trials (sample paths) of `NPERIODS` observations each. If you do not specify a value for this argument, the default is `1`, indicating a single path of correlated state variables.
`DeltaTime`	Scalar or `NPERIODS`-by-`1` column vector of positive time increments between observations. `DeltaTime` represents the familiar *dt* found in stochastic differential equations, and determines the times at which the simulated paths of the output state variables are reported. If you do not specify a value for this argument, the default is `1`.

`NSTEPS`	Positive scalar integer number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`). The `simByEuler` method partitions each time increment *dt* into `NSTEPS` subintervals of length *dt*/`NSTEPS`, and refines the simulation by evaluating the simulated state vector at `NSTEPS - 1` intermediate points. Although `simByEuler` does not report the output state vector at these intermediate points, the refinement improves accuracy by allowing the simulation to more closely approximate the underlying continuous-time process. If you do not specify a value for `NSTEPS`, the default is `1`, indicating no intermediate evaluation.
`Antithetic`	Scalar logical flag that indicates whether `simByEuler` uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes). When `Antithetic` is `TRUE` (logical `1`), `simByEuler` performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs: • Odd trials `(1,3,5,...)` correspond to the primary Gaussian paths. • Even trials `(2,4,6,...)` are the matching antithetic paths of each pair derived by negating the Gaussian draws of the corresponding primary (odd) trial. If you specify `Antithetic` to be any value other than `TRUE`, `simByEuler` assumes that it is `FALSE` (logical `0`) by default, and does not perform antithetic sampling. When you specify an input noise process (see `Z`), `simByEuler` ignores the value of `Antithetic`.

`Z`	Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. Specify this argument as a function, or as an `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` three-dimensional array of dependent random variates. If you specify `Z` as a function, it must return an `NBROWNS`-by-`1` column vector, and you must call it with two inputs: • A real-valued scalar observation time t. • An `NVARS`-by-`1` state vector X_t. If you do not specify a value for `Z`, `simByEuler` generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object.
`StorePaths`	Scalar logical flag that indicates how the output array `Paths` is stored and returned to the caller. If `StorePaths` is `TRUE` (the default value) or is unspecified, `simByEuler` returns `Paths` as a three-dimensional time series array. If `StorePaths` is `FALSE` (logical `0`), `simByEuler` returns the `Paths` output array as an empty matrix.
`Processes`	Function or cell array of functions that indicates a sequence of end-of-period processes or state vector adjustments of the form $X_t = P(t, X_t)$ `simByEuler` applies processing functions at the end of each observation period. These functions must accept the current observation time t and the current state vector X_t, and return a state vector that may be an adjustment to the input state. If you specify more than one processing function, `simByEuler` invokes the functions in the order in which they appear in the cell array. You can use this argument to specify boundary conditions, prevent negative prices, accumulate statistics, plot graphs, and more. If you do not specify a processing function, `simByEuler` makes no adjustments and performs no processing.

Output Arguments

`Paths`	`(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` three-dimensional time series array, consisting of simulated paths of correlated state variables. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t. When the input flag `StorePaths` = `FALSE`, `simByEuler` returns `Paths` as an empty matrix.
`Times`	`(NPERIODS + 1)`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with the corresponding row of `Paths`.
`Z`	`(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` three-dimensional time series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation.

Examples

Implementing Multidimensional Equity Market Models, Implementation 5: Using the simByEuler Method on page 17-41

Algorithms

- This simulation engine provides a discrete-time approximation of the underlying generalized continuous-time process. The simulation is derived directly from the stochastic differential equation of motion. Thus, the discrete-time process approaches the true continuous-time process only as `DeltaTime` approaches zero.
- The input argument `Z` allows you to directly specify the noise-generation process. This process takes precedence over the `Correlation` parameter of the`sde` object and the value of the `Antithetic` input flag. If you do not specify a value for `Z`, `simByEuler` generates correlated Gaussian variates, with or without antithetic sampling as requested.
- The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simByEuler` tests the state vector X_t for an all-`NaN` condition. Thus, to signal an early termination of a given trial, all elements of the state vector X_t must be `NaN`. This test enables a user-defined `Processes` function to signal early termination of a trial, and offers significant performance benefits in some situations (for example, pricing down-and-out barrier options).

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`simBySolution` | `simBySolution` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

simulate

Simulate multivariate stochastic differential equations (SDEs)

Syntax

```
[Paths, Times, Z] = simulate(MDL, ...)
```

Classes

All classes in the “SDE Class Hierarchy” on page 17-5.

Description

This method simulates any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where:

- X is an *NVARS*-by-`1` state vector of process variables (for example, short rates or equity prices) to simulate.
- W is an *NBROWNS*-by-`1` Brownian motion vector.
- F is an *NVARS*-by-`1` vector-valued drift-rate function.
- G is an *NVARS*-by-*NBROWNS* matrix-valued diffusion-rate function.

`[Paths, Times, Z] = simulate(MDL, ...)` simulates `NTRIALS` sample paths of `NVARS` correlated state variables, driven by `NBROWNS` Brownian motion sources of risk over `NPERIODS` consecutive observation periods, approximating continuous-time stochastic processes.

Input Arguments

`MDL`	Stochastic differential equation model.

Optional Input Arguments

The `simulate` method accepts any variable-length list of input arguments that the simulation method or function referenced by the `SDE.Simulation` parameter requires or accepts. It passes this input list directly to the appropriate SDE simulation method or user-defined simulation function.

Output Arguments

`Paths`	`(NPERIODS + 1)`-by-`NVARS`-by-`NTRIALS` three-dimensional time series array, consisting of simulated paths of correlated state variables. For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t.
`Times`	`(NPERIODS + 1)`-by-`1` column vector of observation times associated with the simulated paths. Each element of `Times` is associated with a corresponding row of `Paths`.
`Z`	`NTIMES`-by-`NBROWNS`-by-`NTRIALS` three-dimensional time series array of dependent random variates used to generate the Brownian motion vector (Wiener processes) that drove the simulated results found in `Paths`. `NTIMES` is the number of time steps at which `simulate` samples the state vector. `NTIMES` includes intermediate times designed to improve accuracy, which `simulate` does not necessarily report in the `Paths` output time series.

Examples

Antithetic Sampling

Simulation methods allow you to specify a popular *variance reduction* technique called *antithetic sampling*. This technique attempts to replace one sequence of random observations with another of the same expected value, but smaller variance.

In a typical Monte Carlo simulation, each sample path is independent and represents an independent trial. However, antithetic sampling generates sample paths in pairs. The first path of the pair is referred to as the *primary path*, and the second as the *antithetic path*. Any given pair is independent of any other pair, but the two paths within each pair are highly correlated. Antithetic sampling literature often recommends averaging the discounted payoffs of each pair, effectively halving the number of Monte Carlo trials.

This technique attempts to reduce variance by inducing negative dependence between paired input samples, ideally resulting in negative dependence between paired output samples. The greater the extent of negative dependence, the more effective antithetic sampling is.

This example applies antithetic sampling to a path-dependent barrier option. Consider a European up-and-in call option on a single underlying stock. The evolution of this stock's price is governed by a Geometric Brownian Motion (`GBM`) model with constant parameters:

$$dX_t = 0.05X_t dt + 0.3X_t dW_t$$

Assume the following characteristics:

- The stock currently trades at 105.
- The stock pays no dividends.
- The stock volatility is 30% per annum.
- The option strike price is 100.
- The option expires in three months.
- The option barrier is 120.
- The risk-free rate is constant at 5% per annum.

The goal is to simulate various paths of daily stock prices, and calculate the price of the barrier option as the risk-neutral sample average of the discounted terminal option payoff. Since this is a barrier option, you must also determine if and when the barrier is crossed.

This example performs antithetic sampling by explicitly setting the `Antithetic` flag to `true`, and then specifies an end-of-period processing function to record the maximum and terminal stock prices on a path-by-path basis.

1 Create a `GBM` model using the `gbm` constructor:

2 Perform a small-scale simulation that explicitly returns two simulated paths:

3 Perform antithetic sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs. Odd paths (1,3,5,...) correspond to the primary Gaussian paths. Even paths (2,4,6,...) are the matching antithetic paths of each pair, derived by negating the Gaussian draws of the corresponding primary (odd) path.

   Verify this by examining the matching paths of the primary/antithetic pair:

To price the European barrier option, specify an end-of-period processing function to record the maximum and terminal stock prices. This processing function is accessible by time and state, and is implemented as a nested function with access to shared information that allows the option price and corresponding standard error to be calculated. For more information on using an end-of-period processing function, see "Pricing Equity Options" on page 17-55.

1 Simulate 200 paths using the processing function method:

2 Approximate the option price with a 95% confidence interval:

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`gbm` | `simByEuler` | `simBySolution` | `simBySolution`

Topics

“Simulating Equity Prices” on page 17-34
“Simulating Interest Rates” on page 17-59
“Stratified Sampling” on page 17-70
“Pricing American Basket Options by Monte Carlo Simulation” on page 17-84
“Base SDE Models” on page 17-16
“Drift and Diffusion Models” on page 17-19
“Linear Drift Models” on page 17-23
“Parametric Models” on page 17-25
“SDEs” on page 17-2
“SDE Models” on page 17-8
“SDE Class Hierarchy” on page 17-5
“Performance Considerations” on page 17-76

Introduced in R2008a

second

Seconds of date or time

Syntax

```
Seconds = second(Date)
Seconds = second(___,F)
```

Description

`Seconds = second(Date)` returns the seconds given a serial date number or a date character vector.

`Seconds = second(___,F)` returns the second of one or more date character vectors, `Date`, using format defined by the optional input `F`. `Date` can be a character array where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`.

Examples

Determine the Seconds of the Date for Various Dates

Find the seconds of the day (`Date`) using a serial date number.

```
Seconds = second(738647.558427893)
```

```
Seconds = 8.1700
```

Find the seconds of the day (`Date`) using a date character vector format.

```
Seconds = second('06-May-2022, 13:24:08.17')
```

```
Seconds = 8.1700
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Date` — Date to determine second

serial date number | date character vector | cell array of date character vectors

Date to determine second, specified as a serial date number or date character vector.

`Date` can be an array of date character vectors, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`

Data Types: `single` | `double` | `char` | `cell`

`F` — Date format symbol

character vector designating date format

Date format symbol, specified as a character vector to designate the date format symbol for input argument `Date`. For more information on supported date character vector formats, see `datestr`. Note, formats with `'Q'` are not accepted.

Data Types: `char`

Output Arguments

`Seconds` — Seconds of date or time

serial date number | datetime array

Seconds of date or time, returned as a serial date number or date character vector.

See Also

`datevec` | `minute` | `second`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

selectreturn

Portfolio configurations from 3-D efficient frontier

Syntax

```
PortConfigs = selectreturn(AllMean,AllCovariance,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,AllCovariance,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`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

setAssetList

Set up list of identifiers for assets

Use the `setAssetList` function with a `Portfolio`, `PortfolioCVaR`, or `PorfolioMAD` object to set up list of identifiers for assets for a portfolio object.

For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-21, “PortfolioCVaR Object Workflow” on page 5-20, and “PortfolioMAD Object Workflow” on page 6-19.

Syntax

```
obj = setAssetList(obj,varargin)
```

Description

`obj = setAssetList(obj,varargin)` sets up the list of identifiers for assets for a portfolio object.

Examples

Create a Default List of Asset Names with Three Assets for a Portfolio Object

Create a default list of asset names with three assets.

```
p = Portfolio('NumAssets',3);
p = setAssetList(p);
disp(p.AssetList);
```

```
    'Asset1'    'Asset2'    'Asset3'
```

Create an Explicitly Named List of Asset Names with Three Assets for a Portfolio Object

Create a list of asset names for three equities AGG, EEM, and VEU.

```
p = Portfolio;
p = setAssetList(p, 'AGG', 'EEM', 'VEU');
disp(p.AssetList);
```

```
    'AGG'    'EEM'    'VEU'
```

Create a Default List of Asset Names with Three Assets for a PortfolioCVaR Object

Create a default list of asset names with three assets.

```
p = PortfolioCVaR('NumAssets',3);
p = setAssetList(p);
disp(p.AssetList);
```

```
    'Asset1'    'Asset2'    'Asset3'
```

Create an Explicitly Named List of Asset Names with Three Assets for a PortfolioCVaR Object

Create a list of asset names for three equities AGG, EEM, and VEU.

```
p = PortfolioCVaR;
p = setAssetList(p, 'AGG', 'EEM', 'VEU');
disp(p.AssetList);
```

```
    'AGG'    'EEM'    'VEU'
```

Create a Default List of Asset Names with Three Assets for a PortfolioMAD Object

Create a default list of asset names with three assets.

```
p = PortfolioMAD('NumAssets',3);
p = setAssetList(p);
disp(p.AssetList);
```

```
    'Asset1'    'Asset2'    'Asset3'
```

Create an Explicitly Named List of Asset Names with Three Assets for a PortfolioMAD Object

Create a list of asset names for three equities AGG, EEM, and VEU.

```
p = PortfolioMAD;
p = setAssetList(p, 'AGG', 'EEM', 'VEU');
disp(p.AssetList);
```

```
    'AGG'    'EEM'    'VEU'
```


- "Common Operations on the Portfolio Object" on page 4-36
- "Common Operations on the PortfolioCVaR Object" on page 5-36
- "Common Operations on the PortfolioMAD Object" on page 6-34
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`varargin` — Asset identifiers

comma-separated list of character vectors | cell array of character vectors

Asset identifiers, specified as a comma-separated list of character vectors or a cell array of character vectors where each character vector is an asset identifier.

If an asset list is entered as an input, this function overwrites an existing asset list in the object if one exists.

If no asset list is entered as an input, three actions can occur:

- If `NumAssets` is nonempty and `AssetList` is empty, `AssetList` becomes a numbered list of assets with default names according to the hidden property in `defaultforAssetList` (`'Asset'`).
- If `NumAssets` is nonempty and `AssetList` is nonempty, nothing happens.
- If `NumAssets` is empty and `AssetList` is empty, the default `NumAssets` =`1` is set and a default asset list is created (`'Asset1'`).

Data Types: `char` | `cell`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

The underlying object (`obj`) has a number of public hidden properties to format the asset list:

- `defaultforAssetList` — Default name for assets (`'Asset'`). Change this name to create default asset names such as `'ETF'`, `'Bond'`.
- `sortAssetList` — Reserved for future implementation.
- `uppercaseAssetList` — If `true`, make all asset identifiers uppercase character vectors. Otherwise do nothing. Default is `false`.

Tips

- You can also use dot notation to set up list of identifiers for assets.

```
obj = obj.setAssetList(varargin);
```

- To clear an `AssetList`, call this method with `[]` or `{[]}`.

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

Topics

“Common Operations on the Portfolio Object” on page 4-36
“Common Operations on the PortfolioCVaR Object” on page 5-36
“Common Operations on the PortfolioMAD Object” on page 6-34
“Portfolio Optimization Examples” on page 4-147

Introduced in R2011a

setAssetMoments

Set moments (mean and covariance) of asset returns for Portfolio object

Use the `setAssetMoments` function with a `Portfolio` object to set moments (mean and covariance) of asset returns.

For details on the workflow, see "Portfolio Object Workflow" on page 4-21.

Syntax

```
obj = setAssetMoments(obj,AssetMean)
obj = setAssetMoments(obj,AssetMean,AssetCovar,NumAssets)
```

Description

`obj = setAssetMoments(obj,AssetMean)` obtains mean and covariance of asset returns for a `Portfolio` object.

`obj = setAssetMoments(obj,AssetMean,AssetCovar,NumAssets)` obtains mean and covariance of asset returns for a `Portfolio` object with additional options for `AssetCovar` and `NumAssets`.

Examples

Set Asset Moments for a Portfolio Object

Set the asset moment properties, given the mean and covariance of asset returns in the variables `m` and `C`.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
```

```
m = m/12;
C = C/12;

p = Portfolio;
p = setAssetMoments(p, m, C);
[assetmean, assetcovar] = getAssetMoments(p)

assetmean =

    0.0042
    0.0083
    0.0100
    0.0150


assetcovar =

    0.0005    0.0003    0.0002         0
    0.0003    0.0024    0.0017    0.0010
    0.0002    0.0017    0.0048    0.0028
         0    0.0010    0.0028    0.0102
```

- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-48
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object. For more information on creating a portfolio object, see

- `Portfolio`

`AssetMean` — Mean of asset returns

vector

Mean of asset returns, specified as a vector.

Note If `AssetMean` is a scalar and the number of assets is known, scalar expansion occurs. If the number of assets cannot be determined, this method assumes that `NumAssets = 1`.

Data Types: `double`

`AssetCovar` — Covariance of asset returns

symmetric positive-semidefinite matrix

Covariance of asset returns, specified as a symmetric positive-semidefinite matrix.

Note If `AssetCovar` is a scalar and the number of assets is known, a diagonal matrix is formed with the scalar value along the diagonals. If it is not possible to determine the number of assets, this method assumes that `NumAssets = 1`.

If `AssetCovar` is a vector, a diagonal matrix is formed with the vector along the diagonal.

Data Types: `double`

`NumAssets` — Number of assets

integer

Number of assets, specified as an integer.

Note If `NumAssets` is not already set in the object, `NumAssets` can be entered to resolve array expansions with `AssetMean` or `AssetCovar`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio` object. For more information on creating a portfolio object, see

- `Portfolio`

Tips

- You can also use dot notation to set moments (mean and covariance) of the asset returns.

  ```
  obj = obj.setAssetMoments(obj, AssetMean, AssetCovar, NumAssets);
  ```

- To clear `NumAssets` and `AssetCovar`, use this function to set these respective inputs to `[]`.

See Also

`estimateAssetMoments` | `estimateFrontierByRisk`

Topics

"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-48
"Portfolio Optimization Examples" on page 4-147
"Portfolio Optimization Theory" on page 4-3

Introduced in R2011a

setBounds

Set up bounds for portfolio weights

Use the `setBounds` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up bounds for portfolio weights for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setBounds(obj,LowerBound)
obj = setBounds(obj,LowerBound,UpperBound,NumAssets)
```

Description

`obj = setBounds(obj,LowerBound)` sets up bounds for portfolio weights for portfolio objects.

`obj = setBounds(obj,LowerBound,UpperBound,NumAssets)` sets up bounds for portfolio weights for portfolio objects with additional options for `UpperBound` and `NumAssets`.

Given bound constraints `LowerBound` and `UpperBound`, every weight in a portfolio `Port` must satisfy the following:

```
LowerBound <= Port <= UpperBound
```

Examples

Set Bound Constraints for a Portfolio Object

Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. To set the bound constraints for a balanced fund.

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];

p = Portfolio;
p = setBounds(p, lb, ub);
disp(p.NumAssets);

     2

disp(p.LowerBound);

    0.5000
    0.2500

disp(p.UpperBound);

    0.7500
    0.5000
```

Set Bound Constraints for a PortfolioCVaR Object

Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. To set the bound constraints for a balanced fund.

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];

p = PortfolioCVaR;
p = setBounds(p, lb, ub);
disp(p.NumAssets);

     2

disp(p.LowerBound);
```

```
    0.5000
    0.2500

disp(p.UpperBound);

    0.7500
    0.5000
```

Set Bound Constraints for a PortfolioMAD Object

Suppose you have a balanced fund with stocks that can range from 50% to 75% of your portfolio and bonds that can range from 25% to 50% of your portfolio. To set the bound constraints for a balanced fund.

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];

p = PortfolioMAD;
p = setBounds(p, lb, ub);
disp(p.NumAssets);

     2

disp(p.LowerBound);

    0.5000
    0.2500

disp(p.UpperBound);

    0.7500
    0.5000
```

- “Working with Bound Constraints Using Portfolio Object” on page 4-72
- “Working with Bound Constraints Using PortfolioCVaR Object” on page 5-68
- “Working with Bound Constraints Using PortfolioMAD Object” on page 6-66
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`LowerBound` — Lower-bound weight for each asset

vector

Lower-bound weight for each asset, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note

- If either `LowerBound` or `UpperBound` are input as empties with `[]`, the corresponding attributes in the portfolio object are cleared and set to `[]`.
- If `LowerBound` or `UpperBound` are specified as scalars and `NumAssets` exists or can be imputed, then they undergo scalar expansion. The default value for `NumAssets` is `1`.
- If both `LowerBound` and `UpperBound` exist and they are not ordered correctly, the `setBounds` function switches bounds if necessary.

Data Types: `double`

`UpperBound` — Upper-bound weight for each asset

vector

Upper-bound weight for each asset, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note

- If either `LowerBound` or `UpperBound` are input as empties with `[]`, the corresponding attributes in the portfolio object are cleared and set to `[]`.
- If `LowerBound` or `UpperBound` are specified as scalars and `NumAssets` exists or can be imputed, then they undergo scalar expansion. The default value for `NumAssets` is `1`.
- If both `LowerBound` and `UpperBound` exist and they are not ordered correctly, the `setBounds` function switches bounds if necessary.

Data Types: `double`

`NumAssets` — Number of assets in portfolio

scalar

Number of assets in portfolio, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note `NumAssets` cannot be used to change the dimension of a portfolio object.

- If either `LowerBound` or `UpperBound` are input as empties with `[]`, the corresponding attributes in the portfolio object are cleared and set to `[]`.
- If `LowerBound` or `UpperBound` are specified as scalars and `NumAssets` exists or can be imputed, then they undergo scalar expansion. The default value for `NumAssets` is `1`.
- If both `LowerBound` and `UpperBound` exist and they are not ordered correctly, the `setBounds` function switches bounds if necessary.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to set up the bounds for portfolio weights.

```
obj = obj.setBounds(LowerBound, UpperBound, NumAssets);
```

See Also

`getBounds`

Topics

"Working with Bound Constraints Using Portfolio Object" on page 4-72
"Working with Bound Constraints Using PortfolioCVaR Object" on page 5-68
"Working with Bound Constraints Using PortfolioMAD Object" on page 6-66
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setBudget

Set up budget constraints

Use the `setBudget` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up budget constraints for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setBudget(obj,LowerBudget)
obj = setBudget(obj,LowerBudget,UpperBudget)
```

Description

`obj = setBudget(obj,LowerBudget)` sets up budget constraints for portfolio objects.

`obj = setBudget(obj,LowerBudget,UpperBudget)` sets up budget constraints for portfolio objects with an additional option for `UpperBudget`.

Examples

Set Budget Constraint for a Portfolio Object

Assume you have a fund that permits up to 10% leverage, which means that your portfolio can be from 100% to 110% invested in risky assets. Given a Portfolio object `p`, set the budget constraint.

```
p = Portfolio;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);
```

```
     1

disp(p.UpperBudget);

    1.1000
```

Set Budget Constraint for a PortfolioCVaR Object

Assume you have a fund that permits up to 10% leverage, which means that your portfolio can be from 100% to 110% invested in risky assets. Given a CVaR portfolio object `p`, set the budget constraint.

```
p = PortfolioCVaR;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);

     1

disp(p.UpperBudget);

    1.1000
```

Set Budget Constraint for a PortfolioMAD Object

Assume you have a fund that permits up to 10% leverage, which means that your portfolio can be from 100% to 110% invested in risky assets. Given PortfolioMAD object `p`, set the budget constraint.

```
p = PortfolioMAD;
p = setBudget(p, 1, 1.1);
disp(p.LowerBudget);

     1

disp(p.UpperBudget);

    1.1000
```

- “Working with Budget Constraints Using Portfolio Object” on page 4-75

- "Working with Budget Constraints Using PortfolioCVaR Object" on page 5-71
- "Working with Budget Constraints Using PortfolioMAD Object" on page 6-69
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`LowerBudget` — Lower-bound for budget constraint

scalar

Lower-bound for budget constraint, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note Given bounds for a budget constraint in either `LowerBudget` or `UpperBudget`, budget constraints require any portfolio in `Port` to satisfy:

```
LowerBudget <= sum(Port) <= UpperBudget
```

One or both constraints may be specified. The usual budget constraint for a fully invested portfolio is to have `LowerBudget = UpperBudget = 1`. However, if the portfolio has allocations in cash, the budget constraints can be used to specify the cash constraints. For example, if the portfolio can hold between 0% and 10% in cash, the budget constraint would be set up with

```
obj = setBudget(obj, 0.9, 1)
```

Data Types: `double`

`UpperBudget` — Upper-bound for budget constraint

scalar

Upper-bound for budget constraint, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note Given bounds for a budget constraint in either `LowerBudget` or `UpperBudget`, budget constraints require any portfolio in `Port` to satisfy:

```
LowerBudget <= sum(Port) <= UpperBudget
```

One or both constraints may be specified. The usual budget constraint for a fully invested portfolio is to have `LowerBudget = UpperBudget = 1`. However, if the portfolio has allocations in cash, the budget constraints can be used to specify the cash constraints. For example, if the portfolio can hold between 0% and 10% in cash, the budget constraint would be set up with

```
obj = setBudget(obj, 0.9, 1)
```

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to set up the budget constraints.

```
obj = obj.setBudget(LowerBudget, UpperBudget);
```

See Also

`getBudget`

Topics

"Working with Budget Constraints Using Portfolio Object" on page 4-75
"Working with Budget Constraints Using PortfolioCVaR Object" on page 5-71
"Working with Budget Constraints Using PortfolioMAD Object" on page 6-69
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setCosts

Set up proportional transaction costs

Use the `setCosts` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up proportional transaction costs for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setCosts(obj,BuyCost)

obj = setCosts(obj,BuyCost,SellCost,InitPort,NumAssets)
```

Description

`obj = setCosts(obj,BuyCost)` sets up proportional transaction costs for portfolio objects.

`obj = setCosts(obj,BuyCost,SellCost,InitPort,NumAssets)` sets up proportional transaction costs for portfolio objects with additional options specified for `SellCost`, `InitPort`, and `NumAssets`.

Given proportional transaction costs and an initial portfolio in the variables `BuyCost`, `SellCost`, and `InitPort`, the transaction costs for any portfolio `Port` reduce expected portfolio return by:

```
BuyCost' * max{0, Port - InitPort} + SellCost' * max{0, InitPort - Port}
```

Examples

Set Up Transaction Costs for a Portfolio Object

Assume you have the same costs and initial portfolio as in the previous example. Given a Portfolio object `p` with an initial portfolio already set, use the `setCosts` function to set up transaction costs.

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = Portfolio('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);

     5

disp(p.BuyCost);

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

disp(p.SellCost);

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

disp(p.InitPort);

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Set Up Transaction Costs for a PortfolioCVaR Object

Given a CVaR portfolio object `p` with an initial portfolio already set, use the `setCosts` function to set up transaction costs.

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioCVaR('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);
```

```
     5
```

```
disp(p.BuyCost);
```

```
    0.0013
    0.0013
    0.0013
    0.0013
    0.0013
```

```
disp(p.SellCost);
```

```
    0.0013
    0.0070
    0.0013
    0.0013
    0.0024
```

```
disp(p.InitPort);
```

```
    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

Set Up Transaction Costs for a PortfolioMAD Object

Given PortfolioMAD object `p` with an initial portfolio already set, use the `setCosts` function to set up transaction costs.

```
bc = [ 0.00125; 0.00125; 0.00125; 0.00125; 0.00125 ];
sc = [ 0.00125; 0.007; 0.00125; 0.00125; 0.0024 ];
x0 = [ 0.4; 0.2; 0.2; 0.1; 0.1 ];

p = PortfolioMAD('InitPort', x0);
p = setCosts(p, bc, sc);

disp(p.NumAssets);

     5

disp(p.BuyCost);

    0.0013
    0.0013
    0.0013
    0.0013
    0.0013

disp(p.SellCost);

    0.0013
    0.0070
    0.0013
    0.0013
    0.0024

disp(p.InitPort);

    0.4000
    0.2000
    0.2000
    0.1000
    0.1000
```

- “Working with Transaction Costs” on page 4-62
- “Working with Transaction Costs” on page 5-58
- “Working with Transaction Costs” on page 6-56

- “Portfolio Analysis with Turnover Constraints”
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`BuyCost` — Proportional transaction cost to purchase each asset

vector

Proportional transaction cost to purchase each asset, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note

- If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is `1`.
- Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Data Types: `double`

`SellCost` — Proportional transaction cost to sell each asset

vector

Proportional transaction cost to sell each asset, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note

- If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is `1`.
- Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Data Types: `double`

`InitPort` — Initial or current portfolio weights

vector

Initial or current portfolio weights, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If no `InitPort` is specified, that value is assumed to be `0`.

- If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is `1`.
- Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Data Types: `double`

`NumAssets` — Number of assets in portfolio

scalar

Number of assets in portfolio, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note `NumAssets` cannot be used to change the dimension of a portfolio object.

- If `BuyCost`, `SellCost`, or `InitPort` are specified as scalars and `NumAssets` exists or can be imputed, then these values undergo scalar expansion. The default value for `NumAssets` is `1`.
- Transaction costs in `BuyCost` and `SellCost` are positive valued if they introduce a cost to trade. In some cases, they can be negative valued, which implies trade credits.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up proportional transaction costs.

  ```
  obj = obj.setCosts(BuyCost, SellCost, InitPort, NumAssets);
  ```

- If `BuyCost` or `SellCost` are input as empties with `[]`, the corresponding attributes in the portfolio object are cleared and set to `[]`. If `InitPort` is set to empty with `[]`, it will only be cleared and set to `[]` if `BuyCost`, `SellCost`, and `Turnover` are also empty. Otherwise, it is an error.

See Also

`getCosts` | `setInitPort`

Topics

"Working with Transaction Costs" on page 4-62
"Working with Transaction Costs" on page 5-58
"Working with Transaction Costs" on page 6-56
"Portfolio Analysis with Turnover Constraints"
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setDefaultConstraints

Set up portfolio constraints with nonnegative weights that sum to 1

Use the `setDefaultConstraints` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up portfolio constraints with nonnegative weights that sum to 1 for portfolio objects.

For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-21, “PortfolioCVaR Object Workflow” on page 5-20, and “PortfolioMAD Object Workflow” on page 6-19.

Syntax

```
obj = setDefaultConstraints(obj)
obj = setDefaultConstraints(obj,NumAssets)
```

Description

`obj = setDefaultConstraints(obj)` sets up portfolio constraints with nonnegative weights that sum to 1.

`obj = setDefaultConstraints(obj,NumAssets)` sets up portfolio constraints with nonnegative weights that sum to 1 with an additional option for `NumAssets`.

A "default" portfolio set has `LowerBound` = `0` and `LowerBudget` = `UpperBudget` = `1` such that a portfolio `Port` must satisfy `sum(Port) = 1` with `Port >= 0`.

Examples

Define Default Constraints for the Portfolio Object

Assuming you have 20 assets, you can define the "default" portfolio set.

```
p = Portfolio('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);
```

```
  Portfolio with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
       AssetMean: []
      AssetCovar: []
   TrackingError: []
    TrackingPort: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
            Name: []
       NumAssets: 20
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [20x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
```

Define Default Constraints for the PortfolioCVaR Object

Assuming you have 20 assets, you can define the "default" portfolio set.

```
p = PortfolioCVaR('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);
```

```
  PortfolioCVaR with properties:

           BuyCost: []
          SellCost: []
      RiskFreeRate: []
    ProbabilityLevel: []
          Turnover: []
       BuyTurnover: []
      SellTurnover: []
      NumScenarios: []
              Name: []
         NumAssets: 20
         AssetList: []
          InitPort: []
       AInequality: []
       bInequality: []
         AEquality: []
         bEquality: []
        LowerBound: [20x1 double]
        UpperBound: []
       LowerBudget: 1
       UpperBudget: 1
       GroupMatrix: []
        LowerGroup: []
        UpperGroup: []
            GroupA: []
            GroupB: []
        LowerRatio: []
        UpperRatio: []
```

Define Default Constraints for the PortfolioMAD Object

Assuming you have 20 assets, you can define the "default" portfolio set.

```
p = PortfolioMAD('NumAssets', 20);
p = setDefaultConstraints(p);
disp(p);
```

```
  PortfolioMAD with properties:

        BuyCost: []
       SellCost: []
```

```
RiskFreeRate: []
    Turnover: []
 BuyTurnover: []
SellTurnover: []
NumScenarios: []
        Name: []
   NumAssets: 20
   AssetList: []
    InitPort: []
 AInequality: []
 bInequality: []
   AEquality: []
   bEquality: []
  LowerBound: [20x1 double]
  UpperBound: []
 LowerBudget: 1
 UpperBudget: 1
 GroupMatrix: []
  LowerGroup: []
  UpperGroup: []
      GroupA: []
      GroupB: []
  LowerRatio: []
  UpperRatio: []
```

- "Setting Default Constraints for Portfolio Weights Using Portfolio Object" on page 4-67
- "Setting Default Constraints for Portfolio Weights Using PortfolioCVaR Object" on page 5-63
- "Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`NumAssets` — Number of assets in portfolio

scalar

Number of assets in portfolio, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note `NumAssets` cannot be used to change the dimension of a portfolio object. The default for `NumAssets` is `1`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up the default portfolio set.

  ```
  obj = obj.setDefaultConstraints(NumAssets);
  ```

- This function does not modify any existing constraints in a portfolio object other than the bound and budget constraints. If an `UpperBound` constraint exists, it is cleared and set to `[]`.

See Also

`getBounds` | `setBounds` | `setBudget`

Topics

"Setting Default Constraints for Portfolio Weights Using Portfolio Object" on page 4-67
"Setting Default Constraints for Portfolio Weights Using PortfolioCVaR Object" on page 5-63
"Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object" on page 6-61
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setEquality

Set up linear equality constraints for portfolio weights

Use the `setEquality` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up linear equality constraints for portfolio weights for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj= setEquality(obj,AEquality,bEquality)
```

Description

`obj= setEquality(obj,AEquality,bEquality)` sets up linear equality constraints for portfolio weights for portfolio objects.

Given linear equality constraint matrix `AEquality` and vector `bEquality`, every weight in a portfolio `Port` must satisfy the following:

```
AEquality * Port = bEquality
```

Examples

Set Linear Equality Constraints for a Portfolio Object

Suppose you have a portfolio of five assets, and you want to ensure that the first three assets are 50% of your portfolio. Given a Portfolio object `p`, set the linear equality constraints with the following.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
```

```
p = Portfolio;
p = setEquality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AEquality);

     1     1     1     0     0

disp(p.bEquality);

    0.5000
```

Set Linear Equality Constraints for a PortfolioCVaR Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioCVaR object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setEquality(p, A, b);
disp(p.NumAssets);

     5

disp(p.AEquality);

     1     1     1     0     0

disp(p.bEquality);

    0.5000
```

Set Linear Equality Constraints for a PortfolioMAD Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are 50% of your portfolio. Given a PortfolioMAD object `p`, set the linear equality constraints and obtain the values for `AEquality` and `bEquality`:

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setEquality(p, A, b);
[AEquality, bEquality] = getEquality(p)

AEquality =

     1     1     1     0     0


bEquality = 0.5000
```

- "Working with Linear Equality Constraints Using Portfolio Object" on page 4-86
- "Working with Linear Equality Constraints Using PortfolioCVaR Object" on page 5-82
- "Working with Linear Equality Constraints Using PortfolioMAD Object" on page 6-79
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`AEquality` — Matrix to form linear equality constraints

matrix

Matrix to form linear equality constraints, returned as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note An error results if `AEquality` is empty and `bEquality` is nonempty.

Data Types: `double`

`bEquality` — Vector to form linear equality constraints

vector

Vector to form linear equality constraints, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note An error results if `AEquality` is nonempty and `bEquality` is empty.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up linear equality constraints for portfolio weights.

  ```
  obj = obj.setEquality(AEquality, bEquality);
  ```

- Linear equality constraints can be removed from a portfolio object by entering `[]` for each property you want to remove.

See Also

`addEquality` | `getEquality`

Topics

"Working with Linear Equality Constraints Using Portfolio Object" on page 4-86
"Working with Linear Equality Constraints Using PortfolioCVaR Object" on page 5-82
"Working with Linear Equality Constraints Using PortfolioMAD Object" on page 6-79
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setGroupRatio

Set up group ratio constraints for portfolio weights

Use the `setGroupRatio` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up group ratio constraints for portfolio weights for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setGroupRatio(obj,GroupA)
obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)
```

Description

`obj = setGroupRatio(obj,GroupA)` sets up group ratio constraints for portfolio weights for portfolio objects

`obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)` sets up group ratio constraints for portfolio weights for portfolio objects with additional options specified for `GroupB`, `LowerRatio`, and `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and `LowerRatio` or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

Examples

Set Group Ratio Constraints for a Portfolio Object

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolio never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinanical companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);

     6

disp(p.GroupA);

     1     1     1     0     0     0

disp(p.GroupB);

     0     0     0     1     1     1

disp(p.UpperRatio);

    0.5000
```

Set Group Ratio Constraints for a PortfolioCVaR Object

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolio never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinanical companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);
```

```
disp(p.NumAssets);

     6

disp(p.GroupA);

     1     1     1     0     0     0

disp(p.GroupB);

     0     0     0     1     1     1

disp(p.UpperRatio);

    0.5000
```

Set Group Ratio Constraints for a PortfolioMAD Object

Suppose you want to ensure that the ratio of financial to nonfinancial companies in your portfolio never exceeds 50%. Assume you have six assets with three financial companies (assets 1-3) and three nonfinanical companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);

     6

disp(p.GroupA);

     1     1     1     0     0     0

disp(p.GroupB);

     0     0     0     1     1     1

disp(p.UpperRatio);

    0.5000
```

- “Working with Group Ratio Constraints Using Portfolio Object” on page 4-82
- “Working with Group Constraints Using PortfolioCVaR Object” on page 5-74
- “Working with Group Constraints Using PortfolioMAD Object” on page 6-71
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`GroupA` — Matrix that forms base groups for comparison

matrix

Matrix that forms base groups for comparison, specified as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note The group matrices `GroupA` and `GroupB` are usually indicators of membership in groups, which means that their elements are usually either `0` or `1`. Because of this interpretation, `GroupA` and `GroupB` matrices can be either logical or numerical arrays.

Data Types: `double`

`GroupB` — Matrix that forms comparison groups

matrix

Matrix that forms comparison groups, specified as a matrix `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note The group matrices `GroupA` and `GroupB` are usually indicators of membership in groups, which means that their elements are usually either `0` or `1`. Because of this interpretation, `GroupA` and `GroupB` matrices can be either logical or numerical arrays.

Data Types: `double`

`LowerRatio` — Lower bound for ratio of `GroupB` groups to `GroupA` groups

vector

Lower bound for ratio of `GroupB` groups to `GroupA` groups, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If input is scalar, `LowerRatio` undergoes scalar expansion to be conformable with the group matrices.

Data Types: `double`

`UpperRatio` — Upper bound for ratio of `GroupB` groups to `GroupA` groups

vector

Upper bound for ratio of `GroupB` groups to `GroupA` groups, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If input is scalar, `UpperRatio` undergoes scalar expansion to be conformable with the group matrices.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up group ratio constraints for portfolio weight.

  ```
  obj = obj.setGroupRatio(GroupA, GroupB, LowerRatio, UpperRatio);
  ```

- To remove group ratio constraints, enter empty arrays for the corresponding arrays. To add to existing group ratio constraints, use `addGroupRatio`.

See Also

`addGroupRatio` | `getGroupRatio`

Topics

“Working with Group Ratio Constraints Using Portfolio Object” on page 4-82
“Working with Group Constraints Using PortfolioCVaR Object” on page 5-74
“Working with Group Constraints Using PortfolioMAD Object” on page 6-71
“Portfolio Optimization Examples” on page 4-147
“Portfolio Set for Optimization Using Portfolio Object” on page 4-10
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-10
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-10

Introduced in R2011a

setGroups

Set up group constraints for portfolio weights

Use the `setGroups` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up group constraints for portfolio weights for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setGroups(obj,GroupMatrix,LowerGroup)
obj = setGroups(obj,GroupMatrix,LowerGroup,UpperGroup)
```

Description

`obj = setGroups(obj,GroupMatrix,LowerGroup)` sets up group constraints for portfolio weights for portfolio objects.

`obj = setGroups(obj,GroupMatrix,LowerGroup,UpperGroup)` sets up group constraints for portfolio weights for portfolio objects with an additional option specified for `UpperGroup`.

Given `GroupMatrix` and either `LowerGroup` or `UpperGroup`, a portfolio `Port` must satisfy the following:

```
LowerGroup <= GroupMatrix * Port <= UpperGroup
```

Examples

Set Group Constraints for a Portfolio Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a Portfolio object `p`, set the group constraints with the following.

```
G = [ true true true false false ];
p = Portfolio;
p = setGroups(p, G, [], 0.3);

disp(p.NumAssets);

     5

disp(p.GroupMatrix);

     1     1     1     0     0

disp(p.UpperGroup);

    0.3000
```

Set Group Constraints for a PortfolioCVaR Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given a CVaR portfolio object `p`, set the group constraints with the following.

```
G = [ true true true false false ];
p = PortfolioCVaR;
p = setGroups(p, G, [], 0.3);

disp(p.NumAssets);

     5

disp(p.GroupMatrix);

     1     1     1     0     0

disp(p.UpperGroup);

    0.3000
```

Set Group Constraints for a PortfolioMAD Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets constitute at most 30% of your portfolio. Given PortfolioMAD object `p`, set the group constraints with the following.

```
G = [ true true true false false ];
p = PortfolioMAD;
p = setGroups(p, G, [], 0.3);

disp(p.NumAssets);

     5

disp(p.GroupMatrix);

     1     1     1     0     0

disp(p.UpperGroup);

    0.3000
```

- "Working with Group Constraints Using Portfolio Object" on page 4-78
- "Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
- "Working with Group Constraints Using PortfolioMAD Object" on page 6-71
- "Constraint Specification Using a Portfolio Object" on page 3-34
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`

- `PortfolioCVaR`
- `PortfolioMAD`

`GroupMatrix` — Group constraint matrix
logical or numeric matrix

Group constraint matrix, specified as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note The group matrix `GroupMatrix` is usually an indicator of membership in groups, which means that its elements are usually either `0` or `1`. Because of this interpretation, `GroupMatrix` can be either a logical or numerical matrix.

Data Types: `double`

`LowerGroup` — Lower bound for group constraints
vector

Lower bound for group constraints, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If input is scalar, `LowerGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

Data Types: `double`

`UpperGroup` — Upper bound for group constraints
vector

Upper bound for group constraints, returned as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If input is scalar, `UpperGroup` undergoes scalar expansion to be conformable with `GroupMatrix`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up group constraints for portfolio weights.

  ```
  obj = obj.setGroups(GroupMatrix, LowerGroup, UpperGroup);
  ```

- To remove group constraints, enter empty arrays for the corresponding arrays. To add to existing group constraints, use `addGroups`.

See Also

`addGroups` | `getGroups`

Topics

"Working with Group Constraints Using Portfolio Object" on page 4-78
"Working with Group Constraints Using PortfolioCVaR Object" on page 5-74
"Working with Group Constraints Using PortfolioMAD Object" on page 6-71
"Constraint Specification Using a Portfolio Object" on page 3-34
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setInequality

Set up linear inequality constraints for portfolio weights

Use the `setInequality` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up linear inequality constraints for portfolio weights for portfolio objects.

For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-21, “PortfolioCVaR Object Workflow” on page 5-20, and “PortfolioMAD Object Workflow” on page 6-19.

Syntax

```
obj = setInequality(obj,AInequality,bInequality)
```

Description

`obj = setInequality(obj,AInequality,bInequality)` sets up linear inequality constraints for portfolio weights for portfolio objects.

Given a linear inequality constraint matrix `AInequality` and vector `bInequality`, every weight in a portfolio `Port` must satisfy the following:

```
AInequality * Port <= bInequality
```

Examples

Set Linear Inequality Constraints for a Portfolio Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given a Portfolio object `p`, set the linear inequality constraints with the following.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
```

```
p = Portfolio;
p = setInequality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AInequality);

     1     1     1     0     0

disp(p.bInequality);

    0.5000
```

Set Linear Inequality Constraints for a PortfolioCVaR Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given a CVaR portfolio object `p`, set the linear inequality constraints with the following.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioCVaR;
p = setInequality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AInequality);

     1     1     1     0     0

disp(p.bInequality);

    0.5000
```

Set Linear Inequality Constraints for a PortfolioMAD Object

Suppose you have a portfolio of five assets and you want to ensure that the first three assets are no more than 50% of your portfolio. Given PortfolioMAD object `p`, set the linear inequality constraints with the following.

```
A = [ 1 1 1 0 0 ];
b = 0.5;
p = PortfolioMAD;
p = setInequality(p, A, b);

disp(p.NumAssets);

     5

disp(p.AInequality);

     1     1     1     0     0

disp(p.bInequality);

    0.5000
```

- "Working with Linear Inequality Constraints Using Portfolio Object" on page 4-89
- "Working with Linear Inequality Constraints Using PortfolioCVaR Object" on page 5-85
- "Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-82
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`

- `PortfolioMAD`

`AInequality` — Matrix to form linear inequality constraints

matrix

Matrix to form linear inequality constraints, specified as a matrix for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note An error results if `AInequality` is empty and `bInequality` is nonempty.

Data Types: `double`

`bInequality` — Vector to form linear inequality constraints

vector

Vector to form linear inequality constraints, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note An error results if `AInequality` is nonempty and `bInequality` is empty.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up linear inequality constraints for portfolio weights.

  ```
  obj = obj.setInequality(AInequality, bInequality);
  ```

- To remove inequality constraints, enter empty arguments. To add to existing inequality constraints, use `addInequality`.

See Also

`addInequality` | `getInequality`

Topics

"Working with Linear Inequality Constraints Using Portfolio Object" on page 4-89
"Working with Linear Inequality Constraints Using PortfolioCVaR Object" on page 5-85
"Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-82
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setInitPort

Set up initial or current portfolio

Use the `setInitPort` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up initial or current portfolio for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setInitPort(obj,InitPort)
obj = setInitPort(obj,InitPort,NumAssets)
```

Description

`obj = setInitPort(obj,InitPort)` sets up initial or current portfolio for portfolio objects.

`obj = setInitPort(obj,InitPort,NumAssets)` sets up initial or current portfolio for portfolio objects with an additional options specified for `NumAssets`.

Examples

Set the InitPort Property for a Portfolio Object

Given an initial portfolio in `x0`, use the `setInitPort` function to set the `InitPort` property.

```
p = Portfolio('NumAssets', 4);
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = setInitPort(p, x0);
disp(p.InitPort);
```

```
    0.3000
    0.2000
    0.2000
         0
```

Set InitPort to Create an Equally-Weighted Portfolio of Four Assets for a Portfolio Object

Create an equally weighted portfolio of four assets using the `setInitPort` function.

```
p = Portfolio('NumAssets', 4);
p = setInitPort(p, 1/4, 4);
disp(p.InitPort);
```

```
    0.2500
    0.2500
    0.2500
    0.2500
```

Set the InitPort Property for a PortfolioCVaR Object

Given an initial portfolio in `x0`, use the `setInitPort` function to set the `InitPort` property.

```
p = PortfolioCVaR('NumAssets', 4);
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = setInitPort(p, x0);
disp(p.InitPort);
```

```
    0.3000
    0.2000
    0.2000
         0
```

Set InitPort to Create an Equally-Weighted Portfolio of Four Assets for a PortfolioCVaR Object

Create an equally weighted portfolio of four assets using the `setInitPort` function.

```
p = PortfolioCVaR('NumAssets', 4);
p = setInitPort(p, 1/4, 4);
disp(p.InitPort);
```

```
    0.2500
    0.2500
    0.2500
    0.2500
```

Set the InitPort Property for a PortfolioMAD Object

Given an initial portfolio in `x0`, use the `setInitPort` function to set the `InitPort` property.

```
p = PortfolioMAD('NumAssets', 4);
x0 = [ 0.3; 0.2; 0.2; 0.0 ];
p = setInitPort(p, x0);
disp(p.InitPort);
```

```
    0.3000
    0.2000
    0.2000
         0
```

Set InitPort to Create an Equally-Weighted Portfolio of Four Assets for a PortfolioMAD Object

Create an equally weighted portfolio of four assets using the `setInitPort` function.

```
p = PortfolioMAD('NumAssets', 4);
p = setInitPort(p, 1/4, 4);
disp(p.InitPort);
```

```
    0.2500
    0.2500
    0.2500
    0.2500
```

- “Common Operations on the Portfolio Object” on page 4-36

- “Common Operations on the PortfolioCVaR Object” on page 5-36
- “Common Operations on the PortfolioMAD Object” on page 6-34
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`InitPort` — Initial or current portfolio weights

vector

Initial or current portfolio weights, specified as a vector for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If `InitPort` is specified as a scalar and `NumAssets` exists, then `InitPort` undergoes scalar expansion.

Data Types: `double`

`NumAssets` — Number of assets in portfolio

`1` (default) | scalar

Number of assets in portfolio, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is `1`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

- You can also use dot notation to set up an initial or current portfolio.

  ```
  obj = obj.setInitPort(InitPort, NumAssets);
  ```

- To remove an initial portfolio, call this method with an empty argument `[]` for `InitPort`.

See Also

`setCosts` | `setTurnover`

Topics

“Common Operations on the Portfolio Object” on page 4-36
“Common Operations on the PortfolioCVaR Object” on page 5-36
“Common Operations on the PortfolioMAD Object” on page 6-34
“Portfolio Optimization Examples” on page 4-147
“Portfolio Set for Optimization Using Portfolio Object” on page 4-10
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-10
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-10

Introduced in R2011a

setOneWayTurnover

Set up one-way portfolio turnover constraints

Use the `setOneWayTurnover` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up one-way portfolio turnover constraints for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setOneWayTurnover(obj,BuyTurnover)
obj = setOneWayTurnover(obj,BuyTurnover,SellTurnover,InitPort,
NumAssets)
```

Description

`obj = setOneWayTurnover(obj,BuyTurnover)` sets up one-way portfolio turnover constraints for portfolio objects.

`obj = setOneWayTurnover(obj,BuyTurnover,SellTurnover,InitPort, NumAssets)` sets up one-way portfolio turnover constraints for portfolio objects with additional options specified for `SellTurnover`, `InitPort`, and `NumAssets`.

Given an initial portfolio in `InitPort` and an upper bound for portfolio turnover on purchases in `BuyTurnover` or sales in `SellTurnover`, the one-way turnover constraints require any portfolio `Port` to satisfy the following:

```
1' * max{0, Port - InitPort} <= BuyTurnover
1' * max{0, InitPort - Port} <= SellTurnover
```

Note If `Turnover` =`BuyTurnover` = `SellTurnover`, the constraint is not equivalent to:

```
1' * | Port - InitPort | <= Turnover
```

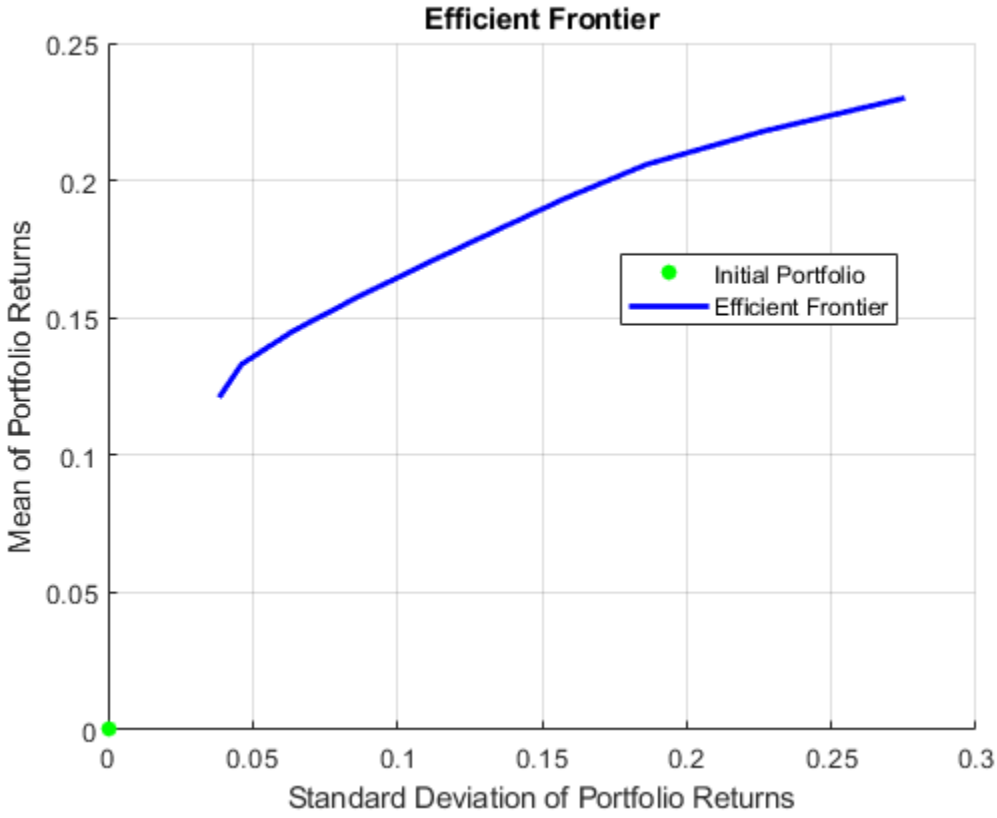
To set this constraint, use `setTurnover`.

Examples

Set One-Way Turnover Constraints for a Portfolio Object

Set one-way turnover constraints.

```
p = Portfolio('AssetMean',[0.1, 0.2, 0.15], 'AssetCovar',...
[ 0.005, -0.010,  0.004; -0.010,  0.040, -0.002;  0.004, -0.002,  0.023]);
p = setBudget(p, 1, 1);
p = setOneWayTurnover(p, 1.3, 0.3, 0); %130-30 portfolio
plotFrontier(p);
```



Set One-Way Turnover Constraints for a PortfolioCVaR Object

Set one-way turnover constraints.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);
disp(p.NumAssets);
```

```
    10
```

```
disp(p.BuyTurnover)
```

```
    0.3000

disp(p.SellTurnover)

    0.2000

disp(p.InitPort);

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

Set One-Way Turnover Constraints for a PortfolioMAD Object

Set one-way turnover constraints.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setOneWayTurnover(p, 0.3, 0.2);
disp(p.NumAssets);

    10

disp(p.BuyTurnover)

    0.3000

disp(p.SellTurnover)

    0.2000

disp(p.InitPort);

    0.1200
    0.0900
    0.0800
    0.0700
```

```
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

- "Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-96
- "Working with One-Way Turnover Constraints Using PortfolioCVaR Object" on page 5-92
- "Working with One-Way Turnover Constraints Using PortfolioMAD Object" on page 6-88
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`BuyTurnover` — Turnover constraint on purchases

nonnegative and finite scalar

Turnover constraint on purchases, specified as a nonnegative and finite scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`SellTurnover` — Turnover constraint on sales

nonnegative and finite scalar

Turnover constraint on sales, specified as a nonnegative and finite scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

`InitPort` — Initial or current portfolio weights
`0` (default) | finite vector with `NumAssets` > `0` elements

Initial or current portfolio weights, specified as a finite vector with `NumAssets` > `0` elements for a `Portfolio`, `PortfolioCVaR`, `PortfolioMAD` input object (`obj`).

Note If no `InitPort` is specified, that value is assumed to be `0`.

If `InitPort` is specified as a scalar and `NumAssets` exists, then `InitPort` undergoes scalar expansion.

Data Types: `double`

`NumAssets` — Number of assets in portfolio
`1` (default) | scalar

Number of assets in portfolio, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is `1`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object
object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Definitions

One-way Turnover Constraint

One-way turnover constraints ensure that estimated optimal portfolios differ from an initial portfolio by no more than specified amounts according to whether the differences are purchases or sales.

The constraints take the form

$$1^T \max\{0, x - x_0\} \le \tau_B$$

$$1^T \max\{0, x_0 - x\} \le \tau_S$$

with

- `x` — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper-bound for turnover constraint on purchases (scalar)
- τ_S — Upper-bound for turnover constraint on sales (scalar)

Specify one-way turnover constraints using the following properties in a supported portfolio object: `BuyTurnover` for τ_B, `SellTurnover` for τ_S, and `InitPort` for x_0.

Note The average turnover constraint (which is set using `setTurnover`) is not just the combination of the one-way turnover constraints with the same value for the constraint.

Tips

You can also use dot notation to set up one-way portfolio turnover constraints.

```
obj = obj.setOneWayTurnover(BuyTurnover,SellTurnover,InitPort,NumAssets)
```

See Also

`getOneWayTurnover` | `setCosts` | `setInitPort` | `setTurnover`

Topics

"Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-96
"Working with One-Way Turnover Constraints Using PortfolioCVaR Object" on page 5-92
"Working with One-Way Turnover Constraints Using PortfolioMAD Object" on page 6-88
"Portfolio Optimization Examples" on page 4-147
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

Introduced in R2011a

setProbabilityLevel

Set probability level for VaR and CVaR calculations

Use the `setProbabilityLevel` function with a `PortfolioCVaR` object to set probability level for VaR and CVaR calculations.

For details on the workflow, see "PortfolioCVaR Object Workflow" on page 5-20.

Syntax

```
obj = setProbabilityLevel(obj,ProbabilityLevel)
```

Description

`obj = setProbabilityLevel(obj,ProbabilityLevel)` sets probability level for VaR and CVaR calculations for a `PortfolioCVaR` object.

Examples

Set Probability Level for a PortfolioCVaR Object

Set the `ProbabilityLevel` for a CVaR portfolio object.

```
p = PortfolioCVaR;
p = setProbabilityLevel(p, 0.95);
disp(p.ProbabilityLevel)
```

```
    0.9500
```

- "What Are Scenarios?" on page 5-45

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` object.

For more information on creating a `PortfolioCVaR` object, see

- `PortfolioCVaR`

`ProbabilityLevel` — Probability level which is 1 minus the probability of losses greater than the value-at-risk

scalar with value from `0` to `1`

Probability level which is 1 minus the probability of losses greater than the value-at-risk, specified as a scalar with value from `0` to `1`.

Note `ProbabilityLevel` must be a value from `0` to `1` and, in most cases, should be a value from `0.9` to `0.99`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `PortfolioCVaR` object. For more information on creating a portfolio object, see

- `PortfolioCVaR`

Tips

You can also use dot notation to set the probability level for VaR and CVaR calculations:

```
obj = obj.setProbabilityLevel(ProbabilityLevel)
```

See Also

`setScenarios`

Topics

“What Are Scenarios?” on page 5-45
“Conditional Value-at-Risk” on page 5-6

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

setSolver

Choose main solver and specify associated solver options for portfolio optimization

Use the `setSolver` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to choose main solver and specify associated solver options for portfolio optimization for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setSolver(obj,solverType)

obj = setSolver(obj,solverType,varargin)
```

Description

`obj = setSolver(obj,solverType)` chooses main solver and specify associated solver options for portfolio optimization for portfolio objects.

`obj = setSolver(obj,solverType,varargin)` chooses main solver and specify associated solver options for portfolio optimization for portfolio objects with additional options specified for `varargin` by using one or more `Name,Value` pair arguments, or an `optimoptions` object.

After you specify a solver, the `varargin` argument accepts either name-value pair arguments to set options or, for the case of solvers from Optimization Toolbox software, a structure created by `optimoptions`.

Examples

Set Solver Type for a Portfolio Object

If you use the `quadprog` function as the `solverType`, the default is the `interior-point-convex` version of `quadprog`.

```
load CAPMuniverse
p = Portfolio('AssetList',Assets(1:12));
p = setDefaultConstraints(p);
p = setSolver(p, 'quadprog');
display(p.solverType);
```

```
quadprog
```

You can switch back to `lcprog` with:

```
p = setSolver(p, 'lcprog');
display(p.solverType);
```

```
lcprog
```

Set the Solver Type as 'fmincon'for a PortfolioCVaR Object

Use `'fmincon'` as the `solverType`.

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon');
display(p.solverType);
```

```
fmincon
```

Set the Solver Type as 'fmincon' and Use Name-Value Pair Arguments to Set the Algorithm for a PortfolioCVaR Object

Use `'fmincon'` as the `solverType` and use name-value pair arguments to set the algorithm to `'trust-region-reflective'` and to turn off the display.

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'off');
display(p.solverOptions.Algorithm);
```

```
trust-region-reflective

display(p.solverOptions.Display);

off
```

Set the Solver Type as 'fmincon' and Use an optimoptions Object to Set the Algorithm for a PortfolioCVaR Object

Use `'fmincon'` as the `solverType` and use an `optimoptions` object to set the algorithm to `'trust-region-reflective'` and to turn off the display.

```
p = PortfolioCVaR;
options = optimoptions('fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'o
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);

trust-region-reflective

display(p.solverOptions.Display);

off
```

Set 'cuttingplane' as the Solver Type with Default Options for a PortfolioCVaR Object

Use `'cuttingplane'` as the `solverType` with default options.

```
p = PortfolioCVaR;
p = setSolver(p,'cuttingplane');
display(p.solverType);

cuttingplane
```

Set 'cuttingplane' as the Solver Type with Maximum Iterations for a PortfolioCVaR Object

Use the Name-Value pair `'MaxIter'` to set the maximum number of iterations to `1500`.

```
p = PortfolioCVaR;
p = setSolver(p,'cuttingplane','MaxIter',1500);
display(p.solverType);
```

```
cuttingplane
```

```
display(p.solverOptions);
```

```
                MaxIter: 1500
                 AbsTol: 1.0000e-06
                 RelTol: 1.0000e-05
    MasterSolverOptions: [1x1 optim.options.Linprog]
```

Set 'cuttingplane' as the Solver Type and Change the Master Solver Option for a PortfolioCVaR Object

For the master solver, continue using the `dual-simplex` algorithm with no display, but tighten its termination tolerance to `1e-8`.

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','Dual-Simplex','Display','off','Optimality
p = setSolver(p,'cuttingplane','MasterSolverOptions',options);
display(p.solverType)
```

```
cuttingplane
```

```
display(p.solverOptions)
```

```
                MaxIter: 1000
                 AbsTol: 1.0000e-06
                 RelTol: 1.0000e-05
    MasterSolverOptions: [1x1 optim.options.Linprog]
```

```
display(p.solverOptions.MasterSolverOptions.Algorithm)
```

```
dual-simplex
```

```
display(p.solverOptions.MasterSolverOptions.Display)
```

```
off
```

```
display(p.solverOptions.MasterSolverOptions.TolFun)
```

```
   1.0000e-08
```

For the master solver, use the `interior-point` algorithm with no display, and with a termination tolerance of `1e-7`.

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','interior-point','Display','off','Optimali
p = setSolver(p,'cuttingplane','MasterSolverOptions',options);
display(p.solverType)

cuttingplane

display(p.solverOptions)

                MaxIter: 1000
                 AbsTol: 1.0000e-06
                 RelTol: 1.0000e-05
    MasterSolverOptions: [1x1 optim.options.Linprog]

display(p.solverOptions.MasterSolverOptions.Algorithm)

interior-point

display(p.solverOptions.MasterSolverOptions.Display)

off

display(p.solverOptions.MasterSolverOptions.TolFun)

   1.0000e-07
```

Set Solver Type as 'fmincon' for a PortfolioMAD Object

Use `'fmincon'` as the `solverType`.

```
p = PortfolioMAD;
p = setSolver(p, 'fmincon');
display(p.solverType);

fmincon
```

Set the Solver Type as 'fmincon' and Use Name-Value Pair Arguments to Set the Algorithm for a PortfolioMAD Object

Use `'fmincon'` as the `solverType` and use name-value pair arguments to set the algorithm to|'sqp'| and to turn on the display.

```
p = PortfolioMAD;
p = setSolver(p, 'fmincon', 'Algorithm', 'sqp', 'Display', 'final');
display(p.solverOptions.Algorithm);
```

```
sqp
```

```
display(p.solverOptions.Display);
```

```
final
```

Set Solver Type as 'fmincon' and Use an optimoptions Structure to Set the Algorithm for a PortfolioMAD Object

Use `'fmincon'` as the `solverType` and use an `optimoptions` object to set the algorithm to `'trust-region-reflective'` and to turn off the display.

```
p = PortfolioMAD;
options = optimoptions('fmincon', 'Algorithm', 'trust-region-reflective', 'Display', 'o
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
```

```
trust-region-reflective
```

```
display(p.solverOptions.Display);
```

```
off
```

Set the Solver Type as 'fmincon' and Use an optimoptions Structure to Set the Algorithm and Use of Gradients for a PortfolioMAD Object

Use `'fmincon'` as the `solverType` and use an `optimoptions` object to set the algorithm to `'active-set'` and to set the gradients flag `'on'` for `'GradObj'` and turn off the display.

```
p = PortfolioMAD;
options = optimoptions('fmincon','algorithm','active-set','display','off','gradobj','or
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);

active-set

display(p.solverOptions.Display);

off
```

- "Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-96
- "Working with One-Way Turnover Constraints Using PortfolioCVaR Object" on page 5-92
- "Working with One-Way Turnover Constraints Using PortfolioMAD Object" on page 6-88
- "Portfolio Optimization Examples" on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Name-Value Pair Arguments for varargin or optimoptions Object

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = setSolver(p,'cuttingplane','MasterSolverOptions',options)`

`solverType` — Solver to use for portfolio optimization
character vector

Solver to use for portfolio optimization, specified using a character vector, or `optimoptions` object.

The `SolverType` input argument depends on which type of object (`obj`) is being used for a portfolio optimization.

For a `Portfolio` object:

- The default `solverType` for the Portfolio object is `'lcprog'` with the control variables `'maxiter'`, `'tiebreak'`, `'tolpiv'`
- The Portfolio object can also use `'quadprog'`, which has several different options that can be set with `optimoptions`. Like Optimization Toolbox software which uses the `interior-point-convex` algorithm as the default algorithm for `quadprog`, the portfolio optimization tools also uses the `interior-point-convex` algorithm. For more information about `quadprog` and quadratic programming algorithms and options, see "Quadratic Programming Algorithms" (Optimization Toolbox).

For a `PortfolioCVaR` object:

- The supported `solverType` are:
  - `'fmincon'`
  - `'cuttingplane'`
- The default is `'fmincon'` using the `'sqp'` algorithm.

For a `PortfolioMAD` object:

- The supported `solverType` is:
  - `'fmincon'`
- The default is `'fmincon'` using the `'sqb'` algorithm and `'GradObj'` set to `'on'`

`varargin` — Options to control the solver specified in `solverType` as name-value pair arguments or an `optimoptions` object
optimoptions object | character vector

Options to control the solver specified in `solverType` as name-value pair arguments or an `optimoptions` object. Note, `optimoptions` is the default and recommended method to set solver options, however `optimset` is also supported.

The `varargin` input argument depends on which type of object (`obj`) is being used for a portfolio optimization

For a `Portfolio` object:

- The default `solverType` for the Portfolio object is `'lcprog'` with the control variables `'maxiter'`, `'tiebreak'`, `'tolpiv'`
- The Portfolio object can also use a `solverType` of `'quadprog'`, which has several different options that can be set with `optimoptions`. Like Optimization Toolbox software which uses the `interior-point-convex` algorithm as the default algorithm for `quadprog`, the portfolio optimization tools also uses the `interior-point-convex` algorithm. For more information about `quadprog` and quadratic programming algorithms and options, see "Quadratic Programming Algorithms" (Optimization Toolbox).

For a `PortfolioCVaR` object:

- For the default `solverType`, `'fmincon'`, `PortfolioCVaR` by default sets the algorithm to `'sqp'`, uses objective function gradients, and turns off the display. All `fmincon` options are supported, either as name-value pair arguments or using an `optimoptions` object. For more information about `fmincon` and quadratic programming algorithms and options, see "Quadratic Programming Algorithms" (Optimization Toolbox).
- For the `cuttingplane` solver, the following solver options are available as name-value pair arguments:
  - `MaxIter`
  - `AbsTol`
  - `RelTol`
  - `MasterSolverOptions`

For a `PortfolioMAD` object:

- For the default `solverType` `'fmincon'`, `PortfolioMAD`, by default, sets the algorithm to `'sqp'` and turns off the display. While all variations of `fmincon` from

Optimization Toolbox are accepted, use of `'sqp'` and `'active-set'` algorithms for `fmincon` are recommended and the use of `'interior-point'` and `'trust-region-reflective'` algorithms are not recommended for MAD portfolio optimization.

- All `fmincon` options are supported either as name-value pair arguments or using an `optimoptions` object.
- `optimoptions` is the default and recommended method to set solver options, however `optimset` is also supported. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see "Constrained Nonlinear Optimization Algorithms" (Optimization Toolbox).

Data Types: `char`

`MaxIter` — Maximum number of iterations
`1000` (default) | positive integer

Maximum number of iterations, specified as a positive integer when using a `PortfolioCVaR` object.

Data Types: `char` | `double`

`AbsTol` — Absolute stopping tolerance
`1e-6` (default) | positive scalar

Absolute stopping tolerance, specified as a positive scalar when using a `PortfolioCVaR` object.

Data Types: `char` | `double`

`RelTol` — Relative stopping tolerance
`1e-5` (default) | positive scalar

Relative stopping tolerance, specified as a positive scalar when using a `PortfolioCVaR` object.

Data Types: `char` | `double`

`MasterSolverOptions` — Options for the master solver
`optimoptions('linprog','Algorithm','Dual-Simplex','Display','off')` (default) | `optimoptions` object

Options for the master solver `linprog`, specified as an `optimoptions` object when using a `PortfolioCVaR` object. For more information about `linprog` and linear programming algorithms and options, see "Linear Programming Algorithms" (Optimization Toolbox).

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to choose the solver and specify associated solver options.

```
obj = obj.setSolver(solverType, varargin);
```

Algorithms

One version of the CVaR portfolio optimization problem minimizes risk, measured as the CVaR of the portfolio, subject to a target return and other linear constraints on the portfolio. For the definition of the CVaR function, see "Risk Proxy" on page 5-6.

$minimize_x CVaR_\alpha(x)$

subject to $\bar{y}^T x \geq TargetReturn$

$Ax \leq b$

$A_{eq} x = b_{eq}$

$$lb \le x \le ub$$

Vector $\bar{y}$ is the mean return vector (the column-wise mean of the scenario matrix Y), so that $\bar{y}^T x$ is the expected return of portfolio x. The first constraint says that the expected return must be at least as good as a target return.

An alternative version of the CVaR portfolio optimization problem maximizes the expected return of the portfolio, subject to a target risk and other linear constraints on the portfolio.

$$maximize_x \bar{y}^T x$$

$$\text{subject to} \quad CVaR_\alpha(x) \le CVaRLimit$$

$$Ax \le b$$

$$A_{eq} x = b_{eq}$$

$$lb \le x \le ub$$

The first constraint in this case says that the portfolio CVaR cannot exceed a given CVaR limit.

By default, the CVaR portfolio object uses `fmincon` to solve the CVaR portfolio optimization problems. For information about `fmincon` and quadratic programming algorithms and options, see "Quadratic Programming Algorithms" (Optimization Toolbox).

Alternatively, the CVaR portfolio optimization problems can be solved with `'cuttingplane'`, an implementation of Kelley's cutting-plane method. For more information, see Kelley [45] at "Portfolio Optimization" on page A-7.

References

[1] Kelley, J. E. "The Cutting-Plane Method for Solving Convex Programs." *Journal of the Society for Industrial and Applied Mathematics*. Vol. 8, No. 4, December 1960, pp. 703–712.

[2] Rockafellar, R. T. and S. Uryasev "Optimization of Conditional Value-at-Risk." *Journal of Risk*. Vol. 2, No. 3, Spring 2000, pp. 21–41.

[3] Rockafellar, R. T. and S. Uryasev "Conditional Value-at-Risk for General Loss Distributions." *Journal of Banking and Finance*. Vol. 26, 2002, pp. 1443–1471.

See Also

`getOneWayTurnover` | `setCosts` | `setInitPort` | `setTurnover`

Topics

“Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-96
“Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-92
“Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-88
“Portfolio Optimization Examples” on page 4-147
“Portfolio Set for Optimization Using Portfolio Object” on page 4-10
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-10
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-10

External Websites

CVaR Portfolio Optimization (5 min 33 sec)

Introduced in R2011a

setScenarios

Set asset returns scenarios by direct matrix

Use the `setScenarios` function with a `PortfolioCVaR` or `PortfolioMAD` objects to set asset returns scenarios by direct matrix.

For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setScenarios(obj,AssetScenarios)
obj = setScenarios(obj,AssetScenarios,Name,Value)
```

Description

`obj = setScenarios(obj,AssetScenarios)` sets asset returns scenarios by direct matrix for `PortfolioCVaR` or `PortfolioMAD` objects.

`obj = setScenarios(obj,AssetScenarios,Name,Value)` set asset returns scenarios by direct matrix for `PortfolioCVaR` or `PortfolioMAD` objects using additional options specified by one or more `Name,Value` pair arguments.

Examples

Set Asset Returns Scenarios for a PortfolioCVaR Object

Given a PortfolioCVaR object `p`, use the `setScenarios` function to set asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
```

```
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);
disp(p)

  PortfolioCVaR with properties:

           BuyCost: []
          SellCost: []
      RiskFreeRate: []
    ProbabilityLevel: 0.9500
           Turnover: []
        BuyTurnover: []
       SellTurnover: []
       NumScenarios: 20000
               Name: []
          NumAssets: 4
          AssetList: []
           InitPort: []
        AInequality: []
        bInequality: []
          AEquality: []
          bEquality: []
         LowerBound: [4x1 double]
         UpperBound: []
        LowerBudget: 1
        UpperBudget: 1
        GroupMatrix: []
         LowerGroup: []
         UpperGroup: []
             GroupA: []
             GroupB: []
         LowerRatio: []
         UpperRatio: []
```

Set Asset Returns Scenarios for a PortfolioMAD Object

Given PortfolioMAD object `p`, use the `setScenarios` function to set asset return scenarios.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
disp(p)
```

```
  PortfolioMAD with properties:

        BuyCost: []
       SellCost: []
    RiskFreeRate: []
       Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 20000
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
```

```
    GroupB: []
LowerRatio: []
UpperRatio: []
```

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-44
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-42

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` or `PortfolioMAD` object.

For more information on creating a `PortfolioCVaR` or `PortfolioMAD` object, see

- `PortfolioCVaR`
- `PortfolioMAD`

`AssetScenarios` — Scenarios for asset returns or prices

matrix

Scenarios for asset returns or prices, specified as a matrix. If the input data are prices, they can be converted into returns with the `'DataFormat'` name-value argument, where the default format is assumed to be `'Returns'`. Be careful using price data because portfolio optimization usually requires total returns and not simply price returns.

This function sets up a function handle to indirectly access input `AssetScenarios` without needing to make a copy of the data.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = setScenarios(p, AssetScenarios,'DataFormat','Returns','GetAssetList',false);`

`DataFormat` — Flag to convert input data as prices into returns

`'Returns'` (default) | character vector with values `'Returns'` or `'Prices'`

Flag to convert input data as prices into returns, specified using a character vector with the values:

- `'Returns'` — Data in `AssetReturns` contains asset total returns.
- `'Prices'` — Data in `AssetReturns` contains asset total return prices.

Data Types: `char`

`GetAssetList` — Flag indicating which asset names to use for the asset list

`false` (default) | logical with value `true` or `false`

Flag indicating which asset names to use for the asset list, specified as a logical with a value of `true` or `false`. Acceptable values for `GetAssetList` are:

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this function and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is currently `'Asset'`.

If the `GetAssetList` flag is `false`, no action occurs, which is the default behavior.

Data Types: `logical`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `PortfolioCVaR` or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to set asset return scenarios.

```
obj = obj.setScenarios(AssetScenarios);
```

See Also

`getScenarios`

Topics

"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
"Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

setTrackingError

Set up maximum portfolio tracking error constraint

Syntax

```
obj = setTrackingError(obj,TrackingError)
obj = setTrackingError(___,TrackingPort,NumAssets)
```

Description

`obj = setTrackingError(obj,TrackingError)` sets up a maximum portfolio tracking error constraint.

`obj = setTrackingError(___,TrackingPort,NumAssets)` sets up a maximum portfolio tracking error constraint using optional arguments for `TrackingPort` and `NumAssets`.

Examples

Set up a Tracking Error Constraint

Create a `Portfolio` object.

```
AssetMean = [ 0.05; 0.1; 0.12; 0.18 ];
AssetCovar = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio('mean', AssetMean, 'covar', AssetCovar, 'lb', 0, 'budget', 1)

p =
  Portfolio with properties:
```

```
          BuyCost: []
         SellCost: []
      RiskFreeRate: []
         AssetMean: [4x1 double]
        AssetCovar: [4x4 double]
     TrackingError: []
      TrackingPort: []
          Turnover: []
       BuyTurnover: []
      SellTurnover: []
              Name: []
         NumAssets: 4
         AssetList: []
          InitPort: []
       AInequality: []
       bInequality: []
         AEquality: []
         bEquality: []
        LowerBound: [4x1 double]
        UpperBound: []
       LowerBudget: 1
       UpperBudget: 1
       GroupMatrix: []
        LowerGroup: []
        UpperGroup: []
            GroupA: []
            GroupB: []
        LowerRatio: []
        UpperRatio: []
```

Estimate the Sharpe ratio for the Portfolio object `p` and define the tracking error.

```
x0 = estimateMaxSharpeRatio(p);
te = 0.08;
p = setTrackingError(p, te, x0);
display(p.NumAssets);
```

```
     4
```

```
display(p.TrackingError);
```

```
    0.0800
```

```
display(p.TrackingPort);
```

```
0.6608
0.1622
0.0626
0.1143
```

- “Working with Tracking Error Constraints Using Portfolio Object” on page 4-100
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object. For more information on creating a portfolio object, see `Portfolio`.

`TrackingError` — Upper bound for portfolio tracking error

nonnegative and finite scalar

Upper bound for portfolio tracking error, specified using a nonnegative and finite scalar.

Given an upper bound for portfolio tracking error in `TrackingError` and a tracking portfolio in `TrackingPort`, the tracking error constraint requires any portfolio in Port to satisfy

```
(Port - TrackingPort)'*AssetCovar*(Port - TrackingPort) <= TrackingError^2 .
```

For more information, see “Tracking Error Constraints” on page 4-16.

Data Types: `double`

`TrackingPort` — Tracking portfolio weights

finite vector

Tracking portfolio weights, specified using a vector. `TrackingPort` must be a finite vector with `NumAssets` > 0 elements.

If no `TrackingPort` is specified, it is assumed to be `0`. If `TrackingPort` is specified as a scalar and `NumAssets` exists, then `TrackingPort` undergoes scalar expansion.

Data Types: `double`

`NumAssets` — Number of assets in portfolio
scalar

Number of assets in portfolio, specified using a scalar. If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is `1`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object
object for portfolio

Updated portfolio object, returned as a `Portfolio` object. For more information on creating a portfolio object, see `Portfolio`.

Note The tracking error constraints can be used with any of the other supported constraints in the Portfolio object without restrictions. However, since the portfolio set necessarily and sufficiently must be a non-empty compact set, the application of a tracking error constraint can result in an empty portfolio set. Use `estimateBounds` to confirm that the portfolio set is non-empty and compact.

Tips

You can also use dot notation to set up a maximum portfolio tracking error constraint.

```
obj = obj.setTrackingError(TrackingError, NumAssets);
```

To remove a tracking portfolio, call this function with an empty argument (`[]`) for `TrackingError`.

```
obj = setTrackingError(obj, [ ]);
```

See Also

`Portfolio` | `setTrackingPort`

Topics

"Working with Tracking Error Constraints Using Portfolio Object" on page 4-100
"Portfolio Optimization Examples" on page 4-147
"Tracking Error Constraints" on page 4-16
"Setting Up a Tracking Portfolio" on page 4-45

Introduced in R2015b

setTrackingPort

Set up benchmark portfolio for tracking error constraint

Syntax

```
obj = setTrackingPort(obj,TrackingPort)
obj = setTrackingPort(___,NumAssets)
```

Description

`obj = setTrackingPort(obj,TrackingPort)` sets up a benchmark portfolio for a tracking error constraint.

`obj = setTrackingPort(___,NumAssets)` sets up a benchmark portfolio for a tracking error constraint using an optional input argument for `NumAssets`.

Examples

Set up a Tracking Port

Create a `Portfolio` object.

```
AssetMean = [ 0.05; 0.1; 0.12; 0.18 ];
AssetCovar = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

p = Portfolio('mean', AssetMean, 'covar', AssetCovar, 'lb', 0, 'budget', 1)

p =
  Portfolio with properties:

          BuyCost: []
```

```
         SellCost: []
     RiskFreeRate: []
        AssetMean: [4x1 double]
       AssetCovar: [4x4 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 4
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
```

Estimate the Sharpe ratio for the Portfolio object `p` and define the tracking port.

```
x0 = estimateMaxSharpeRatio(p);
p = setTrackingPort(p, x0);

display(p.NumAssets);
```

```
     4
```

```
display(p.TrackingPort);
```

```
    0.6608
    0.1622
    0.0626
    0.1143
```

- “Working with Tracking Error Constraints Using Portfolio Object” on page 4-100
- “Portfolio Optimization Examples” on page 4-147

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `Portfolio` object. For more information on creating a portfolio object, see `Portfolio`.

`TrackingPort` — Tracking portfolio weights

vector

Tracking portfolio weights, specified using a vector. If `TrackingPort` is specified as a scalar and `NumAssets` exists, then `TrackingPort` undergoes scalar expansion.

Data Types: `double`

`NumAssets` — Number of assets in portfolio

scalar

Number of assets in portfolio, specified using a scalar. If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is `1`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `Portfolio` object. For more information on creating a portfolio object, see `Portfolio`.

Note The tracking error constraints can be used with any of the other supported constraints in the Portfolio object without restrictions. However, since the portfolio set

necessarily and sufficiently must be a non-empty compact set, the application of a tracking error constraint can result in an empty portfolio set. Use `estimateBounds` to confirm that the portfolio set is non-empty and compact.

Tips

You can also use dot notation to set up a benchmark portfolio for tracking error constraint.

```
obj = obj.setTrackingPort(TrackingPort, NumAssets);
```

To remove a tracking portfolio, call this function with an empty argument (`[]`) for `TrackingPort`.

```
obj = setTrackingPort(obj, [ ]);
```

See Also

`Portfolio` | `setTrackingError`

Topics

"Working with Tracking Error Constraints Using Portfolio Object" on page 4-100
"Portfolio Optimization Examples" on page 4-147
"Tracking Error Constraints" on page 4-16
"Setting Up a Tracking Portfolio" on page 4-45

Introduced in R2015b

setTurnover

Set up maximum portfolio turnover constraint

Use the `setTurnover` function with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to set up maximum portfolio turnover constraint for portfolio objects.

For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-21, "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = setTurnover(obj,Turnover)
obj = setTurnover(obj,Turnover,InitPort,NumAssets)
```

Description

`obj = setTurnover(obj,Turnover)` sets up maximum portfolio turnover constraint for portfolio objects.

`obj = setTurnover(obj,Turnover,InitPort,NumAssets)` sets up maximum portfolio turnover constraint for portfolio objects with additional options specified for `Turnover`, `InitPort`, and `NumAssets`.

Given an upper bound for portfolio turnover in `Turnover` and an initial portfolio in `InitPort`, the turnover constraint requires any portfolio in `Port` to satisfy the following:

```
1' *1/2* | Port - InitPort | <= Turnover
```

Examples

Set Turnover Constraint for a Portfolio Object

Given a Portfolio object `p`, to ensure that average turnover is no more than 30% with an initial portfolio of 10 assets in a variable `x0`, use the `setTurnover` method to set the turnover constraint.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = Portfolio('InitPort', x0);
p = setTurnover(p, 0.3);

disp(p.NumAssets);

    10

disp(p.Turnover);

    0.3000

disp(p.InitPort);

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

Set Turnover Constraint for a CVaR Portfolio Object

Given a CVaR portfolio object `p`, to ensure that average turnover is no more than 30% with an initial portfolio of 10 assets in a variable `x0`, use the `setTurnover` method to set the turnover constraint.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioCVaR('InitPort', x0);
p = setTurnover(p, 0.3);

disp(p.NumAssets);
```

```
    10

disp(p.Turnover);

    0.3000

disp(p.InitPort);

    0.1200
    0.0900
    0.0800
    0.0700
    0.1000
    0.1000
    0.1500
    0.1100
    0.0800
    0.1000
```

Set Turnover Constraint for a MAD Portfolio Object

Given PortfolioMAD object `p`, to ensure that average turnover is no more than 30% with an initial portfolio of 10 assets in a variable `x0`, use the `setTurnover` method to set the turnover constraint.

```
x0 = [ 0.12; 0.09; 0.08; 0.07; 0.1; 0.1; 0.15; 0.11; 0.08; 0.1 ];
p = PortfolioMAD('InitPort', x0);
p = setTurnover(p, 0.3);

disp(p.NumAssets);

    10

disp(p.Turnover);

    0.3000

disp(p.InitPort);

    0.1200
    0.0900
    0.0800
    0.0700
```

```
0.1000
0.1000
0.1500
0.1100
0.0800
0.1000
```

- "Working with Average Turnover Constraints Using Portfolio Object" on page 4-92
- "Working with Average Turnover Constraints Using PortfolioCVaR Object" on page 5-88
- "Portfolio Optimization Examples" on page 4-147
- "Portfolio Analysis with Turnover Constraints"

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

`Turnover` — Portfolio turnover constraint

nonnegative and finite scalar

Portfolio turnover constraint, specified as a nonnegative and finite scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD`input object (`obj`).

`InitPort` — Initial or current portfolio weights

`0` (default) | finite vector with `NumAssets` > `0` elements.

Initial or current portfolio weights, specified as a finite vector with `NumAssets` > `0` elements for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If no `InitPort` is specified, that value is assumed to be `0`.

If `InitPort` is specified as a scalar and `NumAssets` exists, then `InitPort` undergoes scalar expansion.

Data Types: `double`

`NumAssets` — Number of assets in portfolio
`1` (default) | scalar

Number of assets in portfolio, specified as a scalar for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` input object (`obj`).

Note If it is not possible to obtain a value for `NumAssets`, it is assumed that `NumAssets` is `1`.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object
object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `Portfolio`
- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to set up the maximum portfolio turnover constraint.

```
obj = obj.setTurnover(Turnover, InitPort, NumAssets);
```

See Also

`getOneWayTurnover` | `setInitPort` | `setOneWayTurnover`

Topics

"Working with Average Turnover Constraints Using Portfolio Object" on page 4-92
"Working with Average Turnover Constraints Using PortfolioCVaR Object" on page 5-88
"Portfolio Optimization Examples" on page 4-147
"Portfolio Analysis with Turnover Constraints"
"Portfolio Set for Optimization Using Portfolio Object" on page 4-10
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-10
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-10

External Websites

CVaR Portfolio Optimization (5 min 33 sec)

Introduced in R2011a

simulateNormalScenariosByData

Simulate multivariate normal asset return scenarios from data

Use the `simulateNormalScenariosByData` function with a `PortfolioCVaR` or `PortfolioMAD` objects to simulate multivariate normal asset return scenarios from data.

For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = simulateNormalScenariosByData(obj,AssetReturns)

obj = simulateNormalScenariosByData(obj,AssetReturns,NumScenarios,
Name,Value)
```

Description

`obj = simulateNormalScenariosByData(obj,AssetReturns)` simulates multivariate normal asset return scenarios from data for portfolio object for `PortfolioCVaR` or `PortfolioMAD` objects.

`obj = simulateNormalScenariosByData(obj,AssetReturns,NumScenarios, Name,Value)` simulates multivariate normal asset return scenarios from data for portfolio object for `PortfolioCVaR` or `PortfolioMAD` objects using additional options specified by one or more `Name,Value` pair arguments.

This function estimates the mean and covariance of asset returns from either price or return data and then uses these estimates to generate the specified number of scenarios with the function `mvnrnd`.

Data can in be either a `NumSamples`-by-`NumAssets` matrix of `NumSamples` prices or returns at a given periodicity for a collection of `NumAssets` assets or a `fints` object with `NumSamples` observations and `NumAssets` time series.

Note If you want to use the method multiple times and you want to simulate identical scenarios each time the function is called, precede each function call with `rng`(*seed*) using a specified integer seed.

Examples

Simulate Multivariate Normal Asset Return Scenarios from Data for a PortfolioCVaR Object

Given a PortfolioCVaR object `p`, use the `simulateNormalScenariosByData` function to simulate multivariate normal asset return scenarios from data.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

RawData = mvnrnd(m, C, 240);
NumScenarios = 2000;

p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, RawData, NumScenarios)

p =
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: []
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: 2000
                Name: []
           NumAssets: 4
           AssetList: []
            InitPort: []
         AInequality: []
```

```
          bInequality: []
            AEquality: []
            bEquality: []
           LowerBound: []
           UpperBound: []
          LowerBudget: []
          UpperBudget: []
          GroupMatrix: []
           LowerGroup: []
           UpperGroup: []
               GroupA: []
               GroupB: []
           LowerRatio: []
           UpperRatio: []
```

```
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);

disp(p);
```

```
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: 0.9000
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: 2000
                Name: []
           NumAssets: 4
           AssetList: []
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: [4x1 double]
          UpperBound: []
         LowerBudget: 1
         UpperBudget: 1
         GroupMatrix: []
          LowerGroup: []
```

```
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
```

Simulate Multivariate Normal Asset Return Scenarios from Market Data for a PortfolioCVaR Object

Create a PortfolioCVaR object `p` and use the `simulateNormalScenariosByData` function with market data loaded from `CAPMuniverse.mat` to simulate multivariate normal asset return scenarios.

```
load CAPMuniverse

p = PortfolioCVaR('AssetList',Assets(1:12));
disp(p);
```

```
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: []
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: []
                Name: []
           NumAssets: 12
           AssetList: {1x12 cell}
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: []
          UpperBound: []
         LowerBudget: []
         UpperBudget: []
         GroupMatrix: []
          LowerGroup: []
```

```
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
```

Simulate the scenarios from the data for each of the 12 assets from `CAPMuniverse.mat`.

```
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
```

Estimate Mean and Covariance of Asset Returns from Data for a PortfolioMAD Object

Given a PortfolioMAD object `p`, use the `simulateNormalScenariosByData` function to simulate multivariate normal asset return scenarios from data.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

RawData = mvnrnd(m, C, 240);
NumScenarios = 2000;

p = PortfolioMAD;
p = simulateNormalScenariosByData(p, RawData, NumScenarios);
p = setDefaultConstraints(p);

disp(p);
```

```
  PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 2000
            Name: []
```

```
      NumAssets: 4
      AssetList: []
       InitPort: []
    AInequality: []
    bInequality: []
      AEquality: []
      bEquality: []
     LowerBound: [4x1 double]
     UpperBound: []
    LowerBudget: 1
    UpperBudget: 1
    GroupMatrix: []
     LowerGroup: []
     UpperGroup: []
         GroupA: []
         GroupB: []
     LowerRatio: []
     UpperRatio: []
```

Estimate Mean and Covariance of Asset Returns from Market Data for a PortfolioMAD Object

Create a PortfolioMAD object `p` and use the `simulateNormalScenariosByData` function with market data loaded from `CAPMuniverse.mat` to simulate multivariate normal asset return scenarios.

```
load CAPMuniverse

p = PortfolioMAD('AssetList',Assets(1:12));
disp(p);
```

```
  PortfolioMAD with properties:

        BuyCost: []
       SellCost: []
    RiskFreeRate: []
       Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
            Name: []
      NumAssets: 12
      AssetList: {1x12 cell}
```

```
      InitPort: []
    AInequality: []
    bInequality: []
      AEquality: []
      bEquality: []
     LowerBound: []
     UpperBound: []
    LowerBudget: []
    UpperBudget: []
    GroupMatrix: []
     LowerGroup: []
     UpperGroup: []
         GroupA: []
         GroupB: []
     LowerRatio: []
     UpperRatio: []
```

Simulate the scenarios from the data for each of the 12 assets from `CAPMuniverse.mat`.

```
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
```

- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` or `PortfolioMAD` object.

For more information on creating a `PortfolioCVaR` or `PortfolioMAD` object, see

- `PortfolioCVaR`
- `PortfolioMAD`

`AssetReturns` — Asset data that can be converted into asset returns

`fints` object | matrix

Asset data that can be converted into asset returns, specified as a `fints` object or `NumSamples`-by-`NumAssets` matrix

Data Types: `double`

`NumScenarios` — Number of scenarios to simulate
positive integer

Number of scenarios to simulate, specified as a positive integer.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = simulateNormalScenariosByData(p,RawData,NumScenarios,'DataFormat','Returns','MissingData',true,'GetAssetList',true)`

`DataFromat` — Flag to convert input data as prices into returns
`'Returns'` (default) | character vector with values `'Returns'` or `'Prices'`

Flag to convert input data as prices into returns, specified using a character vector with the values:

- `'Returns'` — Data in `AssetReturns` contains asset total returns.
- `'Prices'` — Data in `AssetReturns` contains asset total return prices.

Data Types: `char`

`MissingData` — Flag to use ECM algorithm to handle `NaN` values
`false` (default) | logical with values `true` or `false`

Flag to use ECM algorithm to handle `NaN` values, as a logical with a value of `true` or `false`.

- `false` — Do not use ECM algorithm to handle `NaN` values (exclude `NaN` values).
- `true` — Use ECM algorithm to handle `NaN` values.

Data Types: `logical`

`GetAssetList` — Flag indicating which asset names to use for the asset list

`false` (default) | logical with values `true` or `false`

Flag indicating which asset names to use for the asset list, specified as a logical with a value of `true` or `false`.

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from `fints` object.

If a `fints` object is passed into this function and the `GetAssetList` flag is `true`, the series names from the `fints` object are used as asset names in `obj.AssetList`.

If a matrix is passed and the `GetAssetList` flag is `true`, default asset names are created based on the `AbstractPortfolio` property `defaultforAssetList`, which is `'Asset'`.

If the `GetAssetList` flag is `false`, no action occurs, which is the default behavior.

Data Types: `logical`

Output Arguments

`obj` — Updated portfolio object

object for portfolio

Updated portfolio object, returned as a `PortfolioCVaR` or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to simulate multivariate normal asset return scenarios from data for a PortfolioCVaR or PortfolioMAD object.

```
obj = obj.simulateNormalScenariosByData(AssetReturns, NumScenarios, varargin);
```

See Also

`fints` | `rng` | `simulateNormalScenariosByMoments`

Topics

"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
"Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

simulateNormalScenariosByMoments

Simulate multivariate normal asset return scenarios from mean and covariance of asset returns

Use the `simulateNormalScenariosByMoments` function with a `PortfolioCVaR` or `PortfolioMAD` objects to simulate multivariate normal asset return scenarios from mean and covariance of asset returns.

For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-20, and "PortfolioMAD Object Workflow" on page 6-19.

Syntax

```
obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar,
NumScenarios)
obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar
NumScenarios,NumAssets)
```

Description

`obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar, NumScenarios)` simulates multivariate normal asset return scenarios from mean and covariance of asset returns for `PortfolioCVaR` or `PortfolioMAD` objects.

`obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar NumScenarios,NumAssets)` simulates multivariate normal asset return scenarios from mean and covariance of asset returns for PortfolioCVaR or PortfolioMAD objects using the optional input `NumScenarios`.

Note This function overwrites existing scenarios associated with `PortfolioCVaR` or `PortfolioMAD` objects, and also, possibly, `NumScenarios`.

If you want to use the function multiple times and you want to simulate identical scenarios each time the function is called, precede each function call with `rng`(*seed*) using a specified integer seed.

Examples

Simulate Multivariate Normal Asset Return Scenarios from Moments for a PortfolioCVaR Object

Given PortfolioCVaR object `p`, use the `simulateNormalScenariosByMoments` function to simulate multivariate normal asset return scenarios from moments.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

AssetMean = [.5]

AssetMean = 0.5000

AssetCovar = [.5]

AssetCovar = 0.5000

NumScenarios = 100

NumScenarios = 100


p = simulateNormalScenariosByMoments(p, AssetMean, AssetCovar, NumScenarios)

p =
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
```

```
    ProbabilityLevel: 0.9500
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: 100
                Name: []
           NumAssets: 4
           AssetList: []
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: [4x1 double]
          UpperBound: []
         LowerBudget: 1
         UpperBudget: 1
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
```

Simulate Multivariate Normal Asset Return Scenarios from Moments for a PortfolioMAD Object

Given PortfolioMAD object `p`, use the `simulateNormalScenariosByMoments` function to simulate multivariate normal asset return scenarios from moments.

```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);
```

```
p = PortfolioMAD;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);

AssetMean = [.5]

AssetMean = 0.5000

AssetCovar = [.5]

AssetCovar = 0.5000

NumScenarios = 100

NumScenarios = 100


p = simulateNormalScenariosByMoments(p, AssetMean, AssetCovar, NumScenarios)

p =
  PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 100
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4x1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
```

```
LowerRatio: []
UpperRatio: []
```

- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

Input Arguments

`obj` — Object for portfolio

object

Object for portfolio, specified using a `PortfolioCVaR` or `PortfolioMAD` object.

For more information on creating a `PortfolioCVaR` or `PortfolioMAD` object, see

- `PortfolioCVaR`
- `PortfolioMAD`

`AssetMean` — Mean of asset returns

vector

Mean of asset returns, specified as a vector.

Note If `AssetMean` is a scalar and the number of assets is known, scalar expansion occurs. If the number of assets cannot be determined, this function assumes that `NumAssets` = `1`.

Data Types: `double`

`AssetCovar` — Covariance of asset returns

symmetric positive-semidefinite matrix

Covariance of asset returns, specified as a symmetric positive-semidefinite matrix.

Note

- If `AssetCovar` is a scalar and the number of assets is known, a diagonal matrix is formed with the scalar value along the diagonals. If it is not possible to determine the number of assets, this method assumes that `NumAssets = 1`.
- If `AssetCovar` is a vector, a diagonal matrix is formed with the vector along the diagonal.

Data Types: `double`

`NumScenarios` — Number of scenarios to simulate
positive integer

Number of scenarios to simulate, specified as a positive integer.

Data Types: `double`

`NumAssets` — Number of assets
scalar

Number of assets, specified as a scalar.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object
object for portfolio

Updated portfolio object, returned as a `PortfolioCVaR` or `PortfolioMAD` object. For more information on creating a portfolio object, see

- `PortfolioCVaR`
- `PortfolioMAD`

Tips

You can also use dot notation to simulate multivariate normal asset return scenarios from a mean and covariance of asset returns for a `PortfolioCVaR` or `PortfolioMAD` object.

```
obj = obj.simulateNormalScenariosByMoments(AssetMean, AssetCovar, NumScenarios, NumAssets);
```

See Also

`rng` | `simulateNormalScenariosByData`

Topics

"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-44
"Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-42

External Websites

CVaR Portfolio Optimization (5 min 33 sec)
Analyzing Investment Strategies with CVaR Portfolio Optimization in MATLAB (50 min 42 sec)

Introduced in R2012b

setfield

Set content of specific field

Syntax

```
newfts = setfield(tsobj,field,V)

newfts = setfield(tsobj,field,{dates},V)
```

Description

`setfield` treats the contents of fields in a time series object (`tsobj`) as fields in a structure.

`newfts = setfield(tsobj,field,V)` sets the contents of the specified field to the value `V`. This is equivalent to the syntax `S.field = V`.

`newfts = setfield(tsobj,field,{dates},V)` sets the contents of the specified field for the specified dates. `dates` can be individual cells of date character vectors or a cell of a date character vector's range using the `::` operator, for example, `'03/01/99::03/31/99'`. Dates can contain time-of-day information.

Examples

Example 1. Set the closing value for all days to 3890.

```
load dji30short
format bank
myfts1 = setfield(myfts1, 'Close', 3890)
```

```
myfts1 =

    desc:  DJI30MAR94.dat
    freq:  Daily (1)

    'dates:  (20)'    'Open:  (20)'    'High:  (20)'    'Low:  (20)'    'Close:  (20)'
    '04-Mar-1994'     [    3830.90]    [    3868.04]    [   3800.50]    [     3890.00]
    '07-Mar-1994'     [    3851.72]    [    3882.40]    [   3824.71]    [     3890.00]
```

```
    '08-Mar-1994'    [   3858.48]    [   3881.55]    [   3822.45]    [   3890.00]
    '09-Mar-1994'    [   3853.97]    [   3874.52]    [   3817.95]    [   3890.00]
    '10-Mar-1994'    [   3852.57]    [   3865.51]    [   3801.63]    [   3890.00]
    '11-Mar-1994'    [   3832.58]    [   3872.83]    [   3806.69]    [   3890.00]
    '14-Mar-1994'    [   3870.29]    [   3894.21]    [   3835.96]    [   3890.00]
    '15-Mar-1994'    [   3863.41]    [   3888.46]    [   3826.85]    [   3890.00]
    '16-Mar-1994'    [   3851.03]    [   3879.53]    [   3819.94]    [   3890.00]
    '17-Mar-1994'    [   3853.62]    [   3891.34]    [   3821.66]    [   3890.00]
    '18-Mar-1994'    [   3865.42]    [   3911.78]    [   3838.65]    [   3890.00]
    '21-Mar-1994'    [   3878.38]    [   3898.25]    [   3838.65]    [   3890.00]
    '22-Mar-1994'    [   3865.71]    [   3896.23]    [   3840.66]    [   3890.00]
    '23-Mar-1994'    [   3868.88]    [   3901.41]    [   3839.80]    [   3890.00]
    '24-Mar-1994'    [   3849.88]    [   3865.42]    [   3792.58]    [   3890.00]
    '25-Mar-1994'    [   3827.13]    [   3826.85]    [   3774.73]    [   3890.00]
    '28-Mar-1994'    [   3776.46]    [   3793.45]    [   3719.74]    [   3890.00]
    '29-Mar-1994'    [   3757.17]    [   3771.86]    [   3689.23]    [   3890.00]
    '30-Mar-1994'    [   3688.36]    [   3718.88]    [   3612.36]    [   3890.00]
    '31-Mar-1994'    [   3639.71]    [   3673.10]    [   3544.12]    [   3890.00]
```

Example 2. Set values for specific times on specific days.

First create a financial time series containing time-of-day data.

```
dates = ['01-Jan-2001';'01-Jan-2001'; '02-Jan-2001'; ...
         '02-Jan-2001'; '03-Jan-2001';'03-Jan-2001'];
times = ['11:00';'12:00';'11:00';'12:00';'11:00';'12:00'];
dates_times = cellstr([dates, repmat(' ',size(dates,1),1),...
                       times]);
myfts = fints(dates_times,[(1:4)'; nan; 6],{'Data1'},1,...
              'My FINTS')
```

```
myfts =

    desc:  My FINTS
    freq:  Daily (1)

    'dates:  (6)'    'times:  (6)'    'Data1:  (6)'
    '01-Jan-2001'    '11:00'          [          1]
    '     "      '    '12:00'          [          2]
    '02-Jan-2001'    '11:00'          [          3]
    '     "      '    '12:00'          [          4]
    '03-Jan-2001'    '11:00'          [        NaN]
    '     "      '    '12:00'          [          6]
```

Now use `setfield` to replace the data in `myfts` with new data starting at 12:00 on January 1, 2001 and ending at 11:00 on January 3, 2001.

```
S = setfield(myfts,'Data1',...
           {'01-Jan-2001 12:00::03-Jan-2001 11:00'},(102:105)')
```

```
S =
```

```
desc:  My FINTS
freq:  Daily (1)

'dates:  (6)'    'times:  (6)'    'Data1:  (6)'
'01-Jan-2001'    '11:00'          [       1.00]
'     "     '    '12:00'          [     102.00]
'02-Jan-2001'    '11:00'          [     103.00]
'     "     '    '12:00'          [     104.00]
'03-Jan-2001'    '11:00'          [     105.00]
'     "     '    '12:00'          [       6.00]
```

See Also

`chfield` | `fieldnames` | `getfield` | `isfield` | `rmfield`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

sharpe

Compute Sharpe ratio for one or more assets

Syntax

```
sharpe(Asset)

sharpe(Asset,Cash)

Ratio = sharpe(Asset,Cash)
```

Arguments

`Asset`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` observations of asset returns for `NUMSERIES` asset return series.
`Cash`	(Optional) Either a scalar return for a riskless asset or a vector of asset returns to be a proxy for a riskless asset. In either case, the return periodicity must be the same as the periodicity of `Asset`. For example, if `Asset` is monthly data, then `Cash` must be monthly returns. If no value is supplied, the default value for `Cash` returns is 0.

Description

Given `NUMSERIES` assets with `NUMSAMPLES` returns for each asset in a `NUMSAMPLES`-by-`NUMSERIES` matrix `Asset` and given either a scalar `Cash` asset return or a vector of `Cash` asset returns, the Sharpe ratio is computed for each asset.

The output is `Ratio`, a `1`-by-`NUMSERIES` row vector of Sharpe ratios for each series in `Asset`. Any series in `Asset` with standard deviation of returns equal to 0 has a `NaN` value for its Sharpe ratio.

Note If `Cash` is a vector, `Asset` and `Cash` need not have the same number of returns but must have the same periodicity of returns. The classic Sharpe ratio assumes that `Cash` is riskless. In reality, a short-term cash rate is not necessarily riskless. `NaN` values in the data are ignored.

Examples

See “Sharpe Ratio” on page 7-6.

References

William F. Sharpe. "Mutual Fund Performance." *Journal of Business.* Vol. 39, No. 1, Part 2, January 1966, pp. 119–138.

See Also

`inforatio` | `portalpha`

Topics

“Sharpe Ratio” on page 7-6
“Performance Metrics Illustration” on page 7-4
“Performance Metrics Overview” on page 7-2

Introduced in R2006b

size

Number of dates and data series

Syntax

```
szfts = size(tsobj,dim)
```

```
[numRows,numCols] = size(tsobj)
```

Arguments

`tsobj`	Financial time series object.
`dim`	(Optional) A scalar that specifies the following 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.

`numRows` returns a scalar representing the number of dates (rows).

`numCols` returns a scalar representing the number of data series (columns).

See Also

`length` | `size`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

smoothts

Smooth data

Syntax

```
output = smoothts(input)

output = smoothts(input,'b',wsize)

output = smoothts(input,'g',wsize,stdev)

output = smoothts(input,'e',n)
```

Arguments

`input`	Financial time series object or a row-oriented matrix. In a row-oriented matrix, each row represents an individual set of observations.
`'b'`, `'g'`, or `'e'`	Smoothing method (essentially the type of filter used). Can be Exponential (`e`), Gaussian (`g`), or Box (`b`). Default = `b`.
`wsize`	Window size (scalar). Default = `5`.
`stdev`	Scalar that represents the standard deviation of the Gaussian window. Default = `0.65`.
`n`	For Exponential method, specifies window size or exponential factor, depending upon value. • `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`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

sortfts

Sort financial time series

Syntax

```
sfts = sortfts(tsobj)

sfts = sortfts(tsobj,flag)

sfts = sortfts(tsobj,seriesnames,flag)

[sfts,sidx] = sortfts(...)
```

Arguments

`tsobj`	Financial time series object.
`flag`	(Optional) Sort order: `flag` = `1`; increasing order (default) `flag` = `-1`; decreasing order
`seriesnames`	(Optional) Character vector containing a data series name or cell array of character vectors 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 character vector containing a data series name or a cell array of character vectors containing a list of data series names. If the optional `flag` is set to `-1`, the sort is in decreasing order.

`[sfts, sidx] = sortfts(...)` also returns the index of the original object `tsobj` sorted based on `'dates'` or specified data series name(s).

See Also

`issorted` | `sort` | `sortrows`

Topics

"Using Time Series to Predict Equity Return" on page 12-25
"What Is the Financial Time Series App?" on page 13-2

Introduced before R2006a

spctkd

Slow stochastics

Syntax

```
[spctk,spctd] = spctkd(fastpctk,fastpctd)

[spctk,spctd] = spctkd([fastpctk fastpctd])

[spctk,spctd] = spctkd(fastpctk,fastpctd,dperiods,dmamethod)

[spctk,spctd] = spctkd([fastpctk fastpctd], dperiods, dmamethod)

skdts = spctkd(tsobj)

skdts = spctkd(tsobj,dperiods,dmamethod)

skdts = spctkd(tsobj,dperiods,dmamethod,'ParameterName',ParameterValue, ...)
```

Arguments

`fastpctk`	Fast stochastic F%K (vector).
`fastpctd`	Fast stochastic F%D (vector).
`dperiods`	(Optional) %D periods. Default = `3`.
`dmamethod`	(Optional) %D moving average method. Default = `'e'` (exponential).
`tsobj`	Financial time series object.

Description

`[spctk,spctd] = spctkd(fastpctk,fastpctd)` calculates the slow stochastics S %K and S%D. `spctk` and `spctd` are column vectors representing the respective slow stochastics. The inputs must be single column-oriented vectors containing the fast stochastics F%K and F%D.

`[spctk,spctd] = spctkd([fastpctk fastpctd])` accepts a two-column matrix as input. The first column contains the fast stochastic F%K values, and the second contains the fast stochastic F%D values.

`[spctk,spctd] = spctkd(fastpctk,fastpctd,dperiods,dmamethod)` calculates the slow stochastics, S%K and S%D, using the value of `dperiods` to set the number of periods and `dmamethod` to indicate the moving average method. The inputs `fastpctk` and `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)` lets you specify the length and the method of the moving average used to calculate S%D values.

`skdts = spctkd(tsobj,dperiods,dmamethod,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `KName`: F%K series name
- `DName`: F%D series name

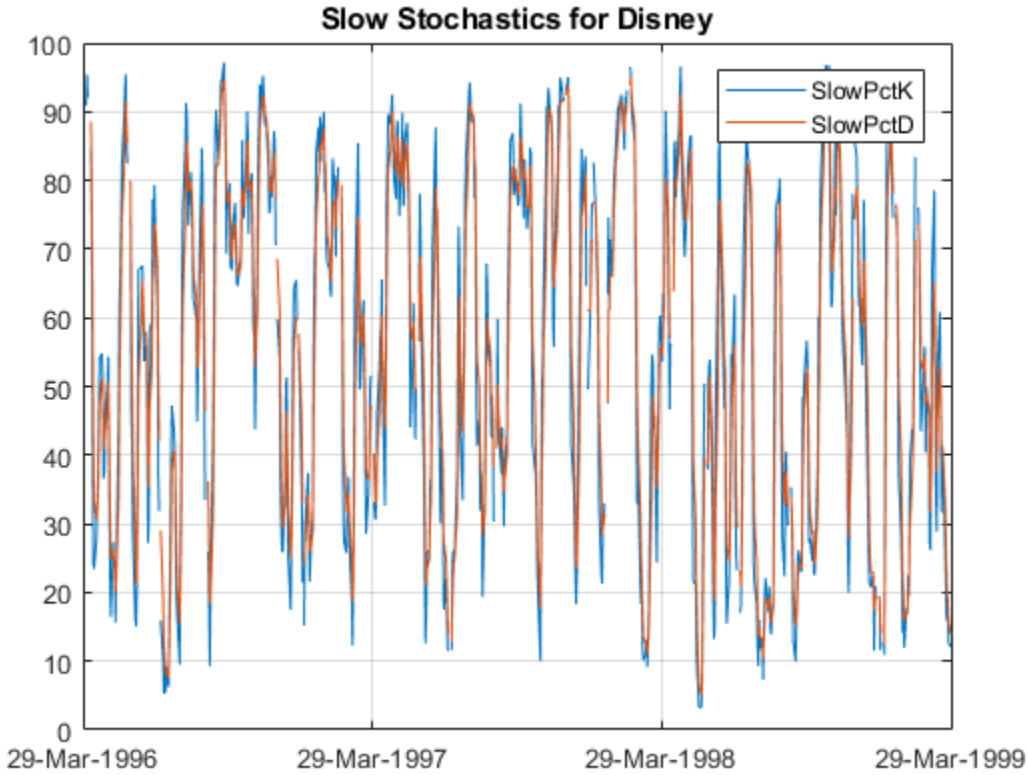
Parameter values are the character vectors that represent the valid parameter names.

Examples

Calculate the Slow Stochastics

This example shows how to calculate the slow stochastics for Disney stock and plot the results.

```
load disney.mat
dis_FastStoch = fpctkd(dis);
dis_SlowStoch = spctkd(dis_FastStoch);
plot(dis_SlowStoch)
title('Slow Stochastics for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 268–271.

See Also

`fpctkd` | `stochosc` | `tsmovavg`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

std

Standard deviation

Syntax

```
tsstd = std(tsobj)

tsstd = std(tsobj,flag)
```

Arguments

`tsobj`	Financial time series object.
`flag`	(Optional) Normalization factor: `flag = 1` normalizes by n (number of observations). `flag = 0` normalizes by n-1.

Description

`tsstd = std(tsobj)` computes the standard deviation of each data series in the financial time series object `tsobj` and returns the results in `tsstd`. The `tsstd` output is a structure with field name(s) identical to the data series name(s).

`tsstd = std(tsobj,flag)` normalizes the data as indicated by `flag`.

See Also

`hist` | `mean`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

stochosc

Stochastic oscillator

Syntax

```
stosc = stochosc(highp,lowp,closep)

stosc = stochosc([highp lowp closep])

stosc = stochosc(highp,lowp,closep,kperiods,dperiods,dmamethod)

stosc = stochosc([highp lowp closep],kperiods,dperiods,dmamethod)

stoscts = stochosc(tsobj,kperiods,dperiods,dmamethod)

stoscts = stochosc(tsobj,kperiods,dperiods,dmamethod,'ParameterName',ParameterValue, ..
```

Arguments

`highp`	High price (vector).
`lowp`	Low price (vector).
`closep`	Closing price (vector).
`kperiods`	(Optional) %K periods. Default = `10`.
`dperiods`	(Optional) %D periods. Default = `3`.
`damethod`	(Optional) %D moving average method. Default = `'e'` (exponential).
`tsobj`	Financial time series object.

Description

`stosc = stochosc(highp,lowp,closep)` calculates the fast stochastics F%K and F %D from the stock price data `highp` (high prices), `lowp` (low prices), and `closep` (closing

prices). `stosc` is a two-column matrix whose first column is the F%K values and second is the F%D values.

`stosc = stochosc([highp lowp closep])` accepts a three-column matrix of high (`highp`), low (`lowp`), and closing prices (`closep`), in that order.

`stosc = stochosc(highp,lowp,closep,kperiods,dperiods,dmamethod)` calculates the fast stochastics F%K and F%D from the stock price data `highp` (high prices), `lowp` (low prices), and `closep` (closing prices). `kperiods` sets the %K period. `dperiods` sets the %D period. `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

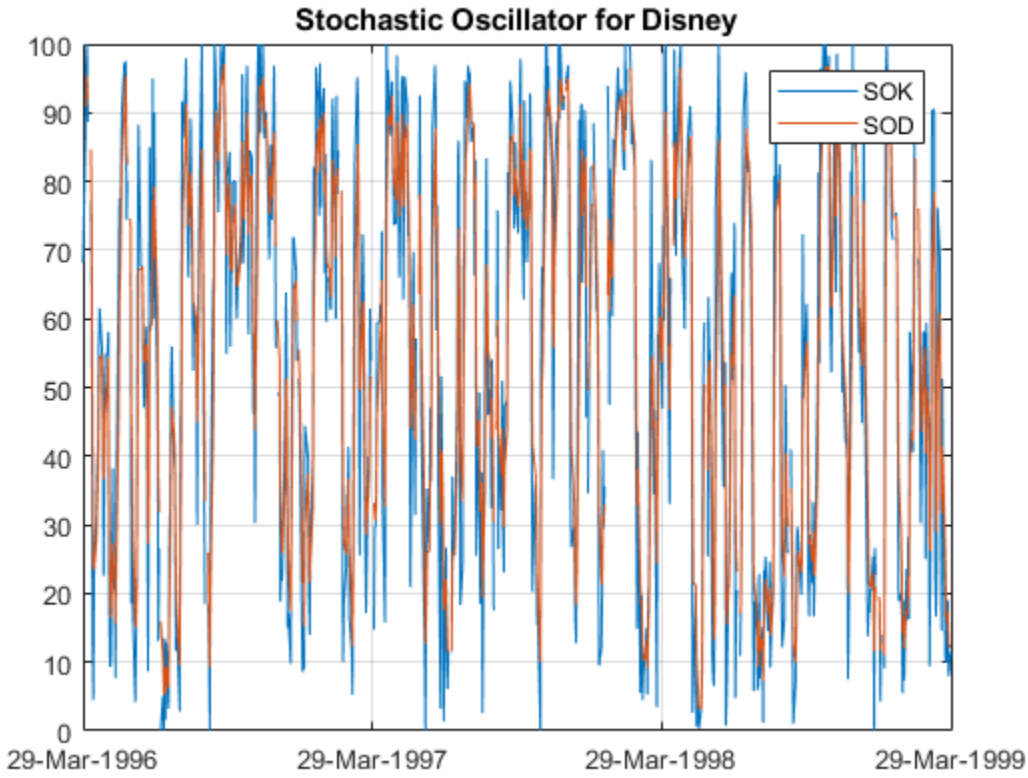
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Stochastic Oscillator

This example shows how to compute the stochastic oscillator for Disney stock and plot the results.

```
load disney.mat
dis_StochOsc = stochosc(dis);
plot(dis_StochOsc)
title('Stochastic Oscillator for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 268–271.

See Also

`fpctkd` | `spctkd`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

subsasgn

Content assignment

Syntax

```
subasgn
```

Description

`subasgn` assigns content to a component within a financial time series object. `subasgn` supports integer indexing or date character vector indexing into the time series object with values assigned to the designated components. *Serial date numbers cannot be used as indices.* To use date character vector indexing, enclose the date character vector(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 character vectors, enclose the character vector 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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

subsref

Subscripted reference

Syntax

```
subref
```

Description

`subsref` implements indexing for a financial time series object. Integer indexing or date (and time) character vector indexing is allowed. *Serial date numbers cannot be used as indices.*

To use date character vector indexing, enclose the date character vector(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 character vector, enclose the indexing character vector 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`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

targetreturn

Portfolio weight accuracy

Syntax

```
return = targetreturn(Universe,Window,Offset,Weights)
```

Arguments

`Universe`	Number of observations (`NUMOBS`) by number of assets plus one (`NASSETS + 1`) array containing total return data for a group of securities. Each row represents an observation. Column 1 contains MATLAB serial date numbers. The remaining columns contain the total return data for each security.
`Window`	Number of data periods used to calculate frontier.
`Offset`	Increment in number of periods at which each frontier is generated.
`Weights`	Number of assets (`NASSETS`) by number of curves (`NCURVES`) matrix of asset allocation weights needed to obtain the target rate of return.

Description

`return = targetreturn(Universe,Window,Offset,Weights)` computes target return values for each window of data and given portfolio weights. These values should match the input target return used with `selectreturn`.

See Also

`frontier` | `portopt`

Topics

"Portfolio Construction Examples" on page 3-7
"Portfolio Optimization Functions" on page 3-4

Introduced before R2006a

taxedrr

After-tax rate of return

Syntax

```
Return = taxedrr(PreTaxReturn,TaxRate)
```

Arguments

`PreTaxReturn`	Nominal rate of return. Enter as a decimal fraction.
`TaxRate`	Tax rate. Enter as a decimal fraction.

Description

`Return = taxedrr(PreTaxReturn,TaxRate)` calculates the after-tax rate of return.

Examples

Calculate the After-Tax Rate of Return

This example shows how to calculate the after-tax rate of return, given an investment that has a 12% nominal rate of return and is taxed at a 30% rate.

```
Return = taxedrr(0.12, 0.30)
```

```
Return = 0.0840
```

- “Analyzing and Computing Cash Flows” on page 2-21

See Also

`effrr` | `irr` | `mirr` | `nomrr` | `xirr`

Topics

"Analyzing and Computing Cash Flows" on page 2-21

Introduced before R2006a

tbilldisc2yield

Convert Treasury bill discount to equivalent yield

Syntax

```
[BEYield,MMYield] = tbilldisc2yield(Discount,Settle,Maturity)
```

Description

`[BEYield,MMYield] = tbilldisc2yield(Discount,Settle,Maturity)` converts the discount rate on Treasury bills into their respective money-market or bond-equivalent yields.

Examples

Convert the Discount Rate on Treasury Bills

This example shows how to convert the discount rate on Treasury bills into their respective money-market or bond-equivalent yields, given a Treasury bill with the following characteristics.

```
Discount = 0.0497;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

[BEYield MMYield] = tbilldisc2yield(Discount, Settle, Maturity)

BEYield = 0.0517

MMYield = 0.0510
```

Convert the Discount Rate on Treasury Bills Using datetime Inputs

This example shows how to use `datetime` inputs to convert the discount rate on Treasury bills into their respective money-market or bond-equivalent yields, given a Treasury bill with the following characteristics.

```
Discount = 0.0497;
Settle = datetime('01-Oct-02','Locale','en_US');
Maturity = datetime('31-Mar-03','Locale','en_US');
[BEYield MMYield] = tbilldisc2yield(Discount, Settle, Maturity)
```

```
BEYield = 0.0517
```

```
MMYield = 0.0510
```

- "Computing Treasury Bill Price and Yield" on page 2-41

Input Arguments

`Discount` — Discount rate of Treasury bills

decimal

Discount rate of the Treasury bills, specified as a scalar of a `NTBILLS`-by-`1` vector of decimal values. The discount rate basis is actual/360.

Data Types: `double`

`Settle` — Settlement date of Treasury bill

serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill

serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`BEYield` — Bond equivalent yields of Treasury bills
numeric

Bond equivalent yields of the Treasury bills, returned as a `NTBILLS`-by-`1` vector. The bond-equivalent yield basis is actual/365.

`MMYield` — Money-market yields of Treasury bills
numeric

Money-market yields of the Treasury bills, returned as a `NTBILLS`-by-`1` vector. The money-market yield basis is actual/360.

References

[1] *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*. Volume 1, 3rd edition, pp. 44–45.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`datetime` | `tbillyield2disc` | `zeroyield`

Topics

“Computing Treasury Bill Price and Yield” on page 2-41
“Treasury Bills Defined” on page 2-40

Introduced before R2006a

tbillprice

Price Treasury bill

Syntax

```
Price = tbillprice(Rate,Settle,Maturity)
Price = tbillprice(___,Type)
```

Description

`Price = tbillprice(Rate,Settle,Maturity)` computes the price of a Treasury bill given a yield or discount rate.

`Price = tbillprice(___,Type)` adds an optional argument for `Type`.

Examples

Compute the Price of a Treasury Bill Using the Bond-Equivalent Yield

Given a Treasury bill with the following characteristics, compute the price of the Treasury bill using the bond-equivalent yield (`Type = 2`) as input.

```
Rate = 0.045;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

Type = 2;

Price = tbillprice(Rate, Settle, Maturity, Type)

Price = 97.8172
```

Price a Portfolio of Treasury Bills

Use `tbillprice` to price a portfolio of Treasury bills.

```
Rate = [0.045; 0.046];
Settle = {'02-Jan-02'; '01-Mar-02'};
Maturity = {'30-June-02'; '30-June-02'};
Type = [2 3];

Price = tbillprice(Rate, Settle, Maturity, Type)
```

```
Price =

   97.8408
   98.4539
```

Price a Portfolio of Treasury Bills Using `datetime` Input

Use `tbillprice` to price a portfolio of Treasury bills using `datetime` input.

```
Rate = [0.045; 0.046];
Type = [2 3];

Settle = datetime({'02-Jan-2002'; '01-Mar-2002'},'Locale','en_US');
Maturity = datetime({'30-June-2002'; '30-June-2002'},'Locale','en_US');
Price = tbillprice(Rate, Settle, Maturity, Type)
```

```
Price =

   97.8408
   98.4539
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`Rate` — Bond-equivalent yield, money-market yield, or discount rate

decimal

Bond-equivalent yield, money-market yield, or discount rate (defined by the input `Type`), specified as a scalar of a `NTBILLS`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date of Treasury bill
serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill
serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Type` — (Optional) Rate type
`2` (default) | numeric with values `1` = money market, `2` = bond-equivalent, `3` = discount rate

Rate type (determines how to interpret values entered in `Rate`), specified as a numeric value of `1`,`2`, or `3` using a scalar or a `NTBILLS`-by-`1` vector.

Note The bond-equivalent yield basis is actual/365. The money-market yield basis is actual/360. The discount rate basis is actual/360.

Data Types: `double`

Output Arguments

`Price` — Treasury bill price for every $100 face
numeric

Treasury bill prices for every $100 face, returned as a `NTBILLS`-by-`1` vector.

References

[1] *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*. Volume 1, 3rd edition, pp. 44–45.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`datetime` | `tbillyield` | `zeroprice`

Topics

“Computing Treasury Bill Price and Yield” on page 2-41
“Treasury Bills Defined” on page 2-40

Introduced before R2006a

tbillrepo

Break-even discount of repurchase agreement

Syntax

```
TBEDiscount = tbillrepo(RepoRate,InitialDiscount,PurchaseDate,
SaleDate,Maturity)
```

Description

`TBEDiscount = tbillrepo(RepoRate,InitialDiscount,PurchaseDate, SaleDate,Maturity)` computes the true break-even discount of a repurchase agreement.

`TBEDiscount`

Examples

Compute the True Break-Even Discount

This example shows how to compute the true break-even discount of a Treasury bill repurchase agreement.

```
RepoRate = [0.045; 0.0475];
InitialDiscount = 0.0475;
PurchaseDate = '3-Jan-2002';
SaleDate = '3-Feb-2002';
Maturity = '3-Apr-2002';

TBEDiscount = tbillrepo(RepoRate, InitialDiscount,...
PurchaseDate, SaleDate, Maturity)

TBEDiscount =
```

```
    0.0491
    0.0478
```

Compute the True Break-Even Discount Using datetime Inputs

This example shows how to use `datetime` inputs to compute the true break-even discount of a Treasury bill repurchase agreement.

```
RepoRate = [0.045; 0.0475];
InitialDiscount = 0.0475;
PurchaseDate = datetime('3-Jan-2002','Locale','en_US');
SaleDate = datetime('3-Feb-2002','Locale','en_US');
Maturity = datetime('3-Apr-2002','Locale','en_US');
TBEDiscount = tbillrepo(RepoRate, InitialDiscount,...
PurchaseDate, SaleDate, Maturity)

TBEDiscount =

    0.0491
    0.0478
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`RepoRate` — Annualized, 360-day based repurchase rate

decimal

Annualized, 360-day based repurchase rate, specified as a scalar of a `NTBILLS`-by-`1` vector of decimal values.

Data Types: `double`

`InitialDiscount` — Discount on Treasury bill on day of purchase

decimal

Discount on the Treasury bill on the day of purchase, specified as a scalar of a `NTBILLS`-by-`1` vector of decimal values.

Data Types: `double`

`PurchaseDate` — Date Treasury bill is purchased
serial date number | date character vector | datetime

Date the Treasury bill is purchased, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`SaleDate` — Date Treasury bill repurchase term is due
serial date number | date character vector | datetime

Date the Treasury bill repurchase term is due, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill
serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`TBEDiscount` — True break-even discount of repurchase agreement
numeric

True break-even discount of a repurchase agreement, returned as a scalar or `NTBILLS`-by-`1` vector.

References

[1] *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest.* Volume 1, 3rd edition, pp. 44–45.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`datetime` | `tbillprice` | `tbillval01` | `tbillyield`

Topics

"Computing Treasury Bill Price and Yield" on page 2-41
"Treasury Bills Defined" on page 2-40

Introduced before R2006a

tbillval01

Value of one basis point

Syntax

```
[Val01Disc,Val01MMY,Val01BEY] = tbillval01(Settle,Maturity)
```

Description

`[Val01Disc,Val01MMY,Val01BEY] = tbillval01(Settle,Maturity)` calculates the value of one basis point of $100 Treasury bill face value on the discount rate, money-market yield, or bond-equivalent yield.

Examples

Compute the Value of One Basis Point

This example shows how to compute the value of one basis point, given a Treasury bill with the following settle and maturity dates.

```
Settle = '01-Mar-03';
Maturity = '30-June-03';
[Val01Disc, Val01MMY, Val01BEY] = tbillval01(Settle, Maturity)

Val01Disc = 0.0034

Val01MMY = 0.0034

Val01BEY = 0.0033
```

Compute the Value of One Basis Point Using datetime Inputs

This example shows how to use `datetime` inputs to compute the value of one basis point, given a Treasury bill with the following settle and maturity dates.

```
Settle = datetime('01-Mar-03','Locale','en_US');
Maturity = datetime('30-June-03','Locale','en_US');
[Val01Disc, Val01MMY, Val01BEY] = tbillval01(Settle, Maturity)
```

```
Val01Disc = 0.0034
```

```
Val01MMY = 0.0034
```

```
Val01BEY = 0.0033
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`Settle` — Settlement date of Treasury bill

serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill

serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`Val01Disc` — Value of one basis point of discount rate for every $100 face

numeric

Value of one basis point of discount rate for every $100 face, returned as a `NTBILLS`-by-`1` vector.

`Val01MMY` — Value of one basis point of money-market yield for every $100 face
numeric

Value of one basis point of money-market yield for every $100 face, returned as a `NTBILLS`-by-`1` vector.

`Val01BEY` — Value of one basis point of bond-equivalent yield for every $100 face
numeric

Value of one basis point of bond-equivalent yield for every $100 face, returned as a `NTBILLS`-by-`1` vector.

References

[1] *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*. Volume 1, 3rd edition, pp. 44–45.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`datetime` | `tbillyield2disc` | `zeroyield`

Topics

“Computing Treasury Bill Price and Yield” on page 2-41
“Treasury Bills Defined” on page 2-40

Introduced before R2006a

tbillyield

Yield on Treasury bill

Syntax

```
[MMYield,BEYield,Discount] = tbillyield(Price,Settle,Maturity)
```

Description

`[MMYield,BEYield,Discount] = tbillyield(Price,Settle,Maturity)` computes the yield of US Treasury bills given `Price`, `Settle`, and `Maturity`.

Examples

Compute the Yield of U.S. Treasury Bills

This example shows how to compute the yield of U.S. Treasury bills, given a Treasury bill with the following characteristics.

```
Price = 98.75;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

[MMYield, BEYield, Discount] = tbillyield(Price, Settle,...
Maturity)

MMYield = 0.0252

BEYield = 0.0255

Discount = 0.0249
```

Compute the Yield of U.S. Treasury Bills Using datetime Inputs

This example shows how to use `datetime` inputs to compute the yield of U.S. Treasury bills, given a Treasury bill with the following characteristics.

```
Price = 98.75;
Settle = datetime('01-Oct-2002','Locale','en_US');
Maturity = datetime('31-Mar-2003','Locale','en_US');
[MMYield, BEYield, Discount] = tbillyield(Price, Settle,Maturity)
```

```
MMYield = 0.0252
```

```
BEYield = 0.0255
```

```
Discount = 0.0249
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`Price` — Price of Treasury bills for every $100 face value

numeric

Price of Treasury bills for every $100 face value, specified as a scalar of a `NTBILLS`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date of Treasury bill

serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill

serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

Output Arguments

`MMYield` — Money-market yields of Treasury bills

numeric

Money-market yields of the Treasury bills, returned as a `NTBILLS`-by-`1` vector.

`BEYield` — Bond equivalent yields of Treasury bills

numeric

Bond equivalent yields of the Treasury bills, returned as a `NTBILLS`-by-`1` vector.

`Discount` — Discount rates of Treasury bills

numeric

Discount rates of the Treasury bills, returned as a `NTBILLS`-by-`1` vector.

References

[1] *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest.* Volume 1, 3rd edition, pp. 44–45.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations.* Wiley, 2002.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation.* McGraw-Hill, 1996.

See Also

`datetime` | `tbilldisc2yield` | `tbillprice` | `tbillyield2disc` | `zeroyield`

Topics

“Computing Treasury Bill Price and Yield” on page 2-41
“Treasury Bills Defined” on page 2-40

Introduced before R2006a

tbillyield2disc

Convert Treasury bill yield to equivalent discount

Syntax

```
Discount = tbillyield2disc(Yield,Settle,Maturity)
Discount = tbillyield2disc(___,Type)
```

Description

`Discount = tbillyield2disc(Yield,Settle,Maturity)` converts the yield on some Treasury bills into their respective discount rates.

`Discount = tbillyield2disc(___,Type)` adds an optional argument for `Type`.

Examples

Compute the Discount Rate on a Money-Market Basis

Given a Treasury bill with these characteristics, compute the discount rate.

```
Yield = 0.0497;
Settle = '01-Oct-02';
Maturity = '31-Mar-03';

Discount = tbillyield2disc(Yield,Settle,Maturity)

Discount = 0.0485
```

Compute the Discount Rate on a Money-Market Basis Using datetime Inputs

Given a Treasury bill with these characteristics, compute the discount rate using datetime inputs.

```
Yield = 0.0497;
Settle = datetime('01-Oct-2002','Locale','en_US');
Maturity = datetime('31-Mar-2003','Locale','en_US');

Discount = tbillyield2disc(Yield,Settle,Maturity)

Discount = 0.0485
```

- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`Yield` — Yield of Treasury bills

decimal

Yield of Treasury bills, specified as a scalar of a `NTBILLS`-by-`1` vector of decimal values.

Data Types: `double`

`Settle` — Settlement date of Treasury bill

serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill

serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Type` — Yield type

`1` (default) | numeric with values `1` = money market, `2` = bond-equivalent

(Optional) Yield type (determines how to interpret values entered in `Yield`), specified as a numeric value of `1` or `2` using a scalar or a `NTBILLS`-by-`1` vector.

Note The bond-equivalent yield basis is actual/365. The money-market yield basis is actual/360.

Data Types: `double`

Output Arguments

`Discount` — Discount rates of Treasury bills

numeric

Discount rates of the Treasury bills, returned as a `NTBILLS`-by-`1` vector.

References

[1] *SIA Fixed Income Securities Formulas for Price, Yield, and Accrued Interest*. Volume 1, 3rd edition, pp. 44–45.

[2] Krgin, D. *Handbook of Global Fixed Income Calculations*. Wiley, 2002.

[3] Stigum, M., Robinson, F. *Money Market and Bond Calculation*. McGraw-Hill, 1996.

See Also

`datetime` | `tbilldisc2yield`

Topics

“Computing Treasury Bill Price and Yield” on page 2-41
“Treasury Bills Defined” on page 2-40

Introduced before R2006a

tbl2bond

Treasury bond parameters given Treasury bill parameters

Syntax

```
[TBondMatrix,Settle] = tbl2bond(TBillMatrix)
```

Description

`[TBondMatrix,Settle] = tbl2bond(TBillMatrix)` restates US Treasury bill market parameters in US Treasury bond form as zero-coupon bonds. This function makes Treasury bills directly comparable to Treasury bonds and notes.

Examples

Restate U.S. Treasury Bill in U.S. Treasury Bond Form

This example shows how to restate U.S. Treasury bill market parameters in U.S. Treasury bond form, given published Treasury bill market parameters for December 22, 1997.

```
TBill = [datenum('jan 02 1998')  10  0.0526  0.0522  0.0530
         datenum('feb 05 1998')  44  0.0537  0.0533  0.0544
         datenum('mar 05 1998')  72  0.0529  0.0527  0.0540];

TBond = tbl2bond(TBill)

TBond =

   1.0e+05 *

         0    7.2976    0.0010    0.0010    0.0000
         0    7.2979    0.0010    0.0010    0.0000
         0    7.2982    0.0010    0.0010    0.0000
```

Restate U.S. Treasury Bill in U.S. Treasury Bond Form Using datetime Input

This example shows how to use `datetime` input to restate U.S. Treasury bill market parameters in U.S. Treasury bond form, given published Treasury bill market parameters for December 22, 1997.

```
TBill = [datenum('jan 02 1998')  10  0.0526  0.0522  0.0530
         datenum('feb 05 1998')  44  0.0537  0.0533  0.0544
         datenum('mar 05 1998')  72  0.0529  0.0527  0.0540];

dates = datetime(TBill(:,1), 'ConvertFrom', 'datenum','Locale','en_US');
data = TBill(:,2:end);
t=[table(dates) array2table(data)];
[TBond, Settle] = tbl2bond(t)
```

```
TBond=3x5 table null
    CouponRate          Maturity            Bid      Asked     AskYield
    __________    ____________________    _______    ______    ________

    0             02-Jan-1998 00:00:00    99.854     99.855     0.053
    0             05-Feb-1998 00:00:00    99.344     99.349    0.0544
    0             05-Mar-1998 00:00:00    98.942     98.946     0.054


Settle = 3x1 datetime array
   22-Dec-1997 00:00:00
   22-Dec-1997 00:00:00
   22-Dec-1997 00:00:00
```

- “Term Structure of Interest Rates” on page 2-45
- “Computing Treasury Bill Price and Yield” on page 2-41

Input Arguments

`TBillMatrix` — Treasury bill parameters

table | matrix

Treasury bill parameters, specified as a 5-column table or a `N`-by-`5` matrix of bond information where the table columns or matrix columns contains:

- `Maturity` (Required) Maturity date of Treasury bills, specified as a serial date number when using a matrix. Use `datenum` to convert date character vectors to serial date numbers. If the input `TBillMatrix` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- `DaysMaturity` (Required) Days to maturity, specified as an integer. Days to maturity are quoted on a skip-day basis; the actual number of days from settlement to maturity is `DaysMaturity + 1`.
- `Bid` (Required) Bid bank-discount rate (the percentage discount from face value at which the bill could be bought, annualized on a simple-interest basis), specified as a decimal fraction.
- `Asked` (Required) Asked bank-discount rate, specified as a decimal fraction.
- `AskYield` (Required) Asked yield (the bond-equivalent yield from holding the bill to maturity, annualized on a simple-interest basis and assuming a 365-day year), specified as a decimal fraction.

Data Types: `double` | `table`

Output Arguments

`TBondMatrix` — Treasury bond parameters

table | matrix

Treasury bond parameters, returned as a table or matrix depending on the `TBillMatrix` input.

When `TBillMatrix` is a table, `TBondMatrix` is also a table, and the variable type for the `Maturity` dates in `TBondMatrix` (column 1) matches the variable type for `Maturity` in `TBillMatrix`. For example, if `Maturity` dates are datetime arrays in `TBillMatrix`, they will also be datetime arrays in `TBondMatrix`.

When `TBillMatrix` input is a `N`-by-`5` matrix, then each row describes a Treasury bond.

The parameters or columns returned for `TBondMatrix` are:

- `CouponRate` (Column 1) Coupon rate, which is always `0` since the Treasury bills are, by definition, a zero coupon instrument.

  .
- `Maturity` (Column 2) Maturity date for each bond in the portfolio as a serial date number. The format of the dates matches the format used for `Maturity` in `TBillMatrix` (serial date number, date character vector, or datetime array).
- `Bid` (Column 3) Bid price based on $100 face value.
- `Asked` (Column 4) Asked price based on $100 face value.
- `AskYield` (Column 5) Asked yield to maturity: the effective return from holding the bond to maturity, annualized on a compound-interest basis.

`Settle` — Settlement dates implied by maturity dates and number of days to maturity quote
serial date number | datetime

Settlement dates implied by the maturity dates and the number of days to maturity quote, returned as a `N`-by-`5` vector containing serial date numbers, by default. `Settle` is returned as a datetime array only if the input `TBillMatrix` is a table containing datetime arrays for `Maturity` in the first column.

See Also

`datetime` | `tr2bonds`

Topics

“Term Structure of Interest Rates” on page 2-45
“Computing Treasury Bill Price and Yield” on page 2-41
“Treasury Bills Defined” on page 2-40

Introduced before R2006a

thirdwednesday

Find third Wednesday of month

Syntax

```
[BeginDates,EndDates] = thirdwednesday(Month,Year)
[BeginDates,EndDates] = thirdwednesday( ___ ,outputType)
```

Description

`[BeginDates,EndDates] = thirdwednesday(Month,Year)` computes the beginning and end period date for a LIBOR contract (third Wednesdays of delivery months).

`[BeginDates,EndDates] = thirdwednesday( ___ ,outputType)`, using optional input arguments, computes the beginning and end period date for a LIBOR contract (third Wednesdays of delivery months).

The type of the outputs depends on the input `outputType`. If this variable is `'datenum'`, `BeginDates` and `EndDates` are serial date numbers. If `outputType` is `'datetime'`, then `BeginDates` and `EndDates` are datetime arrays. By default, `outputType` is set to `'datenum'`.

Examples

Determine the Third Wednesday for Given Months and Years

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 = 3x11 char array
    '16-Oct-2002'
    '15-Oct-2003'
    '20-Oct-2004'


datestr(EndDates)

ans = 3x11 char array
    '16-Jan-2003'
    '15-Jan-2004'
    '20-Jan-2005'
```

Find the third Wednesday dates for swaps commencing in the month of October in the years 2002, 2003, and 2004 using an `outputType` of `'datetime'`.

```
Months = [10; 10; 10];
Year = [2002; 2003; 2004];
[BeginDates, EndDates] = thirdwednesday(Months, Year,'datetime')

BeginDates = 3x1 datetime array
   16-Oct-2002
   15-Oct-2003
   20-Oct-2004


EndDates = 3x1 datetime array
   16-Jan-2003
   15-Jan-2004
   20-Jan-2005
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Month` — Month of delivery for Eurodollar futures

integer from `1` through `12` | vector of integers from `1` through `12`

Month of delivery for Eurodollar futures, specified as an `N`-by-`1` vector of integers from `1` through `12`.

Duplicate dates are returned when identical months and years are supplied.

Data Types: `single` | `double`

`Year` — Delivery year for Eurodollar futures/Libor contracts corresponding to `Month`
four-digit nonnegative integer | vector of four-digit nonnegative integers

Delivery year for Eurodollar futures/Libor contracts corresponding to `Month`, specified as an `N`-by-`1` vector of our-digit nonnegative integers.

Duplicate dates are returned when identical months and years are supplied.

Data Types: `single` | `double`

`outputType` — Output date format
`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

Output date format, specified as a character vector with values `'datenum'` or `'datetime'`. If `outputType` is `'datenum'`, then `BeginDates` and `EndDates` are serial date numbers. However, if `outputType` is `'datetime'`, then `BeginDates` and `EndDates` are datetime arrays.

Data Types: `char`

Output Arguments

`BeginDates` — Third Wednesday of given month and year
serial date number | date character vector

Third Wednesday of given month and year, returned as serial date numbers or date character vectors, or datetime arrays. This is also the beginning of the 3-month period contract.

The type of the outputs depends on the input `outputType`. If this variable is `'datenum'`, `BeginDates` and `EndDates` are serial date numbers. If `outputType` is `'datetime'`, then`BeginDates` and `EndDates` are datetime arrays. By default, `outputType` is set to `'datenum'`.

`EndDates` — End of three-month period contract for given month and year
serial date number | date character vector

End of three-month period contract for given month and year, returned as serial date numbers or date character vectors, or datetime arrays.

The type of the outputs depends on the input `outputType`. If this variable is `'datenum'`, `BeginDates` and `EndDates` are serial date numbers. If `outputType` is `'datetime'`, then`BeginDates` and `EndDates` are datetime arrays. By default, `outputType` is set to `'datenum'`.

See Also

`datetime` | `tr2bonds`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2
"UICalendar User Interface" on page 15-4

Introduced before R2006a

thirtytwo2dec

Thirty-second quotation to decimal

Syntax

```
OutNumber = thirtytwo2dec(InNumber,InFraction)
```

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.

Examples

Change the Price Quotation for a Bond or Bond Future From a Fraction

This example shows how to change the price quotation for a bond or bond future from a fraction with a denominator of 32 to a decimal, given two bonds that are quoted as 101-25 and 102-31.

```
InNumber  = [101; 102];
InFraction = [25; 31];

OutNumber = thirtytwo2dec(InNumber, InFraction)

OutNumber =

  101.7813
  102.9688
```

Input Arguments

`InNumber` — Input number

integer

Input number, specified as a scalar or an `N`-by-`1` vector of integers representing price without the fractional components.

Data Types: `double`

`InFraction` — Fractional portions of each element in `InNumber`

numeric decimal fraction

Fractional portions of each element in `InNumber`, specified as a scalar or an `N`-by-`1` vector of numeric decimal fractions.

Data Types: `double`

Output Arguments

`OutNumber` — Output number that represents sum of `InNumber` and `InFraction`

decimal

Output number that represents sum of `InNumber` and `InFraction`, returned as a decimal.

See Also

`dec2thirtytwo`

Introduced before R2006a

tick2ret

Convert price series to return series

Syntax

```
[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes, Method)
```

Arguments

`TickSeries`	Number of observations (`NUMOBS`) by number of assets (`NASSETS`) matrix of prices of equity assets. Each column is a price series of an individual asset. First row is oldest observation. Last row is most recent. Observations across a given row occur at the same time for all columns.
`TickTimes`	(Optional) `NUMOBS`-by-`1` increasing vector of observation times associated with the prices in `TickSeries`. Times are specified as a serial date numbers, date character vectors, or datetime arrays. If `TickTimes` is empty or missing, sequential observation times from 1, 2, ... `NUMOBS` are assumed.
`Method`	(Optional) Character vector 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

Convert Price Series to Return Series

This example shows how to convert price series to return series, given periodic returns of two stocks observed in the first, second, third, and fourth quarters.

```
TickSeries = [100 80
              110 90
              115 88
              110 91];

TickTimes = [0
             6
             9
             12];

[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes)

RetSeries =

    0.1000    0.1250
    0.0455   -0.0222
   -0.0435    0.0341


RetIntervals =

     6
```

```
     3
     3
```

Convert Price Series to Return Series Using datetime Input

This example shows how to use `datetime` input to convert price series to return series, given periodic returns of two stocks observed in the first, second, third, and fourth quarters.

```
TickSeries = [100 80
110 90
115 88
110 91];
TickTimes = datenum({'1/1/2015','1/7/2015','1/16/2015','1/28/2015'});

TickTimes = datetime(TickTimes, 'ConvertFrom','datenum','Locale','en_US');
[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes)

RetSeries =

    0.1000    0.1250
    0.0455   -0.0222
   -0.0435    0.0341


RetIntervals =

     6
     9
    12
```

- “Data Transformation and Frequency Conversion” on page 12-12

See Also

`datetime` | `ewstats` | `ret2tick`

Topics

“Data Transformation and Frequency Conversion” on page 12-12

Introduced before R2006a

tick2ret (fts)

Convert price series to return series for time series object

Syntax

```
returnFts = tick2ret(priceFts)

returnFts = tick2ret(priceFts,'PARAM1',VALUE1,'PARAM2',VALUE2', ...)
```

Arguments

`priceFts`	Financial time series object of prices.
`'PARAM1'`	(Optional) *`Method`* is a character vector indicating the method to convert asset returns to prices. The value must be defined as `'Simple'` (default) or `'Continuous'`. If *`Method`* is `'Simple'`, `tick2ret` uses simple periodic returns. If *`Method`* is `'Continuous'`, the function uses continuously compounded returns. Case is ignored for *`Method`*.

Description

`returnFts = tick2ret(priceFts,'PARAM1',VALUE1,'PARAM2',VALUE2', ...)` generates a financial time series object of returns.

Note The *i*'th return is quoted for the period `PriceSeries`(*i*) to `PriceSeries`(*i*+1) and is not normalized by the time increment between successive price observations.

If *`Method`* is unspecified or `'Simple'`, the prices are

```
ReturnSeries(i) = PriceSeries(i+1)/PriceSeries(i)-1
```

If *`Method`* is `'Continuous'`, the prices are

```
ReturnSeries(i) = log[PriceSeries(i+1)/PriceSeries(i)]
```

Examples

Convert Price Series to Return Series for a `fints` Object

Compute the return series from the following price series:

```
PriceSeries = [100.0000  100.0000
110.0000  112.0000
115.5000  116.4800
109.7250  122.3040]
```

```
PriceSeries =

  100.0000  100.0000
  110.0000  112.0000
  115.5000  116.4800
  109.7250  122.3040
```

Use the following dates:

```
Dates = {'18-Dec-2000'
'18-Jun-2001'
'17-Sep-2001'
'18-Dec-2001'}
```

```
Dates = 4x1 cell array
    {'18-Dec-2000'}
    {'18-Jun-2001'}
    {'17-Sep-2001'}
    {'18-Dec-2001'}
```

The `fints` object is:

```
p = fints(Dates, PriceSeries)
```

```
p =
```

```
    desc:  (none)
    freq:  Unknown (0)

    'dates:  (4)'    'series1:  (4)'    'series2:  (4)'
    '18-Dec-2000'    [          100]    [          100]
    '18-Jun-2001'    [          110]    [          112]
    '17-Sep-2001'    [     115.5000]    [     116.4800]
    '18-Dec-2001'    [     109.7250]    [     122.3040]
```

`returnFts` is computed as:

```
 tick2ret(p)


ans =

    desc:  (none)
    freq:  Unknown (0)

    'dates:  (3)'    'series1:  (3)'    'series2:  (3)'
    '18-Jun-2001'    [       0.1000]    [       0.1200]
    '17-Sep-2001'    [       0.0500]    [       0.0400]
    '18-Dec-2001'    [      -0.0500]    [       0.0500]
```

Note that for `n` dates in the original time series, there are (`n`-1) dates returned for `returnFts` from `tick2ret`. The formula for the date output dates is described as: `RetDate(i) = PriceDate (i+1)`.

- “Technical Analysis Examples” on page 16-4

See Also

`portsim | ret2tick`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

time2date

Dates from time and frequency

Syntax

```
Dates = time2date(Settle,TFactors)
Dates = time2date(___,Compounding,Basis,EndMonthRule)
```

Description

`Dates = time2date(Settle,TFactors)` computes `Dates` corresponding to compounded rate quotes between `Settle` and `TFactors`. `time2date` is the inverse of `date2time`.

`Dates = time2date(___,Compounding,Basis,EndMonthRule)` computes `Dates` corresponding to compounded rate quotes between `Settle` and `TFactors` using optional input arguments for `Compounding`, `Basis`, and `EndMonthRule`. `time2date` is the inverse of `date2time`.

Examples

Calculate Dates Using `time2date`

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 = 4x11 char array
    '31-Aug-2005'
    '28-Feb-2006'
    '15-Jun-2006'
    '31-Dec-2006'
```

Show `time2date` support for `datetime` input for `Settle`.

```
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);
Dates_calc = time2date(datetime(Settle,'Locale','en_US'), TFactors,...
Compounding, Basis, EndMonthRule)

Dates_calc = 4x1 datetime array
   31-Aug-2005
```

```
28-Feb-2006
15-Jun-2006
31-Dec-2006
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Settle` — Settlement date

serial date number | date character vector | datetime object

Settlement date, specified as a serial date number, date character vector, or datetime array.

Data Types: `double` | `char` | `datetime`

`TFactors` — Time factors

vector

Time factors, corresponding to the compounding value, specified as a vector. `TFactors` must be equal to or greater than zero.

Data Types: `double`

`Compounding` — Rate at which input zero rates are compounded when annualized

`2` (Semiannual compounding) (default) | scalar with numeric values of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `12`, `365`, `-1`

Rate at which input zero rates are compounded when annualized, specified as a scalar with numeric values of: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `12`, `365`, or `-1`. Allowed values are defined as:

- `0` — Simple interest (no compounding)
- `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

The optional `Compounding` argument determines the formula for the discount factors (`Disc`):

- `Compounding` = `1`, `2`, `3`, `4`, `6`, `12`
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, for example, `T = F` is one year.
- `Compounding` = `365`
  - `Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.
- `Compounding` = `-1`
  - `Disc = exp(-T*Z)`, where `T` is time in years.

`Basis` — Day-count basis

`0` (actual/actual) (default) | numeric with value `0` through `13` | vector of numerics with values `0` through `13`

Day-count basis, specified as an integer with a value of `0` through `13` or a `N`-by-`1` vector of integers with values `0` through `13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

`EndMonthRule` — End-of-month rule flag for month having 30 or fewer days
`1` (in effect) (default) | nonnegative integer `[0,1]`

End-of-month rule flag for month having 30 or fewer days, specified as scalar nonnegative integer [`0`, `1`] or a using a `N`-by-`1` vector of values. 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 payment date is always the same numerical day of the month.
- `1` = Set rule on, meaning that a payment date is always the last actual day of the month.

Data Types: `logical`

Output Arguments

`Dates` — Dates corresponding to compounded rate quotes between `Settle` and `TFactors`
serial date number | datetime array

Dates corresponding to compounded rate quotes between `Settle` and `TFactors`, returned as a scalar or a `N`-by-`1` vector using serial date numbers or datetime arrays.

Data Types: `double` | `datetime`

See Also

`cfamounts` | `cftimes` | `date2time` | `datetime`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

times

Financial time series multiplication

Syntax

```
newfts = tsobj_1 .* tsobj_2

newfts = tsobj .* array

newfts = array .* tsobj
```

Arguments

`tsobj_1, tsobj_2`	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` and `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`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

tmfactor

Time factors of arbitrary dates

Syntax

```
TFactors = tmfactor(Settle,Maturity)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers, date character vectors, or datetime arrays. `Settle` must be earlier than `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers, date character vectors, or datetime arrays.

Description

`TFactors = tmfactor(Settle,Maturity)` determines the time factors from a vector of Settlement dates to a vector of Maturity dates.

Examples

Find the `TFactors` for `Settle` and `Maturity` dates.

```
TFactors = tmfactor('1-Jan-2015','1-Jan-2016')
```

```
TFactors =

     2
```

Find the `TFactors` for `Settle` and `Maturity` dates using a datetime array.

```
TFactors = tmfactor(datetime('1-Jan-2015','Locale','en_US'),'1-Jan-2016')
```

```
TFactors =

     2
```

See Also

`cfamounts` | `cftimes` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

toannual

Convert to annual

Syntax

```
newfts = toannual(oldfts)

newfts = toannual(oldfts,'ParameterName',ParameterValue, ...)
```

Arguments

`oldfts`	Financial time series object.

Description

`newfts = toannual(oldfts)` converts a financial time series of any frequency to one of an annual frequency. The default end-of-year is the last business day of the December. `toannual` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as `'00:00'` for those days that did not previously exist in `oldfts`.

Empty (`[ ]`) passed as inputs for parameter pair values for `toannual` triggers the use of the defaults.

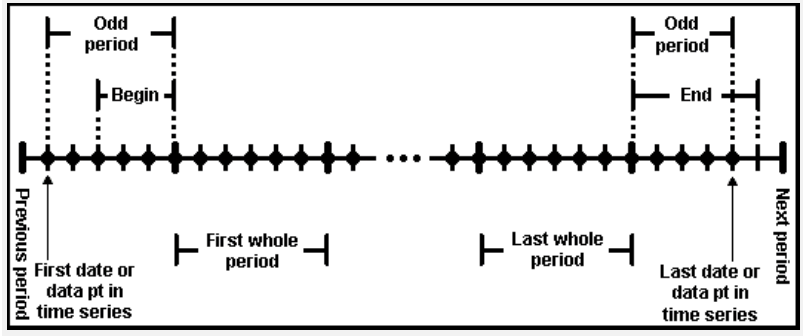
`newfts = toannual(oldfts,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

Parameter Name	Parameter Value	Description
`CalcMethod`	`CumSum`	Returns the cumulative sum of the values within each year. Data for missing dates are given the value `0`.
`CalcMethod`	`Exact`	Returns the exact value at the end-of-year date. No data manipulation occurs.
`CalcMethod`	`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.
`CalcMethod`	`SimpAvg`	Returns an averaged annual value that only takes into account dates with data (non-`NaN`) within each year.
`CalcMethod`	`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 the following parameter name/parameter value pairs are not supported.		
`BusDays`	`0`	Returns a financial time series that ranges from (or between) the first date to the last date in `oldfts` (includes NYSE nonbusiness days and holidays).
`BusDays`	`1`	(Default) Generates a monthly financial time series that ranges from the first date to the last date in `oldfts` (excludes NYSE nonbusiness days and holidays and weekends based on `AltHolidays` and `Weekend`). If an end-of-month date falls on a nonbusiness day or NYSE holiday, returns the last business day of the month. NYSE market closures, holidays, and weekends are observed if `AltHolidays` and `Weekend` are not supplied or empty (`[]`).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all annual dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1. **Note** The default is to create a time series with every date at the specified periodicity, which is with DateFilter = Absolute. If you use DateFilter = Relative, the endpoint effects do not apply since only your data defines which dates appear in the output time series object.
DateFilter	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.
ED	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 (for example, 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).

Parameter Name	Parameter Value	Description
EndPtTol	[Begin, End]	Denotes the minimum number of days that constitute an odd annual period at the endpoints of the time series (before the first-time series date and after the last end-of-year date). 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 endpoint data is ignored.

The following diagram is a general depiction of the factors involved in the determination of endpoints for this function.



TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.

Parameter Name	Parameter Value	Description
`TimeSpec`	`Last`	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
`AltHolidays`		Vector of dates specifying an alternate set of market closure dates.
`AltHolidays`	`-1`	Excludes all holidays.
`Weekend`		Vector of length 7 containing 0's and 1's. The value `1` indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then `Weekend = [1 0 0 0 0 0 1]`.

Examples

Transform Time Series Object from Weekly to Annual Values

This example shows how to transform a time series object from weekly to annual values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
    '05-Feb-1999'     [     134.4882]
```

```
'12-Feb-1999'     [     117.5581]
'19-Feb-1999'     [     106.6683]
'26-Feb-1999'     [     118.2912]
'05-Mar-1999'     [     105.6835]
'12-Mar-1999'     [     128.5836]
'19-Mar-1999'     [     115.1746]
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
'10-Dec-1999'     [      33.8457]
```

```
    '17-Dec-1999'     [      33.3868]
    '24-Dec-1999'     [      32.7737]
    '31-Dec-1999'     [      28.5665]
```

Use `toannual` to obtain the annual aggregate for the `x0` times series.

```
x1 = toannual(x0)
```

```
x1 =

    desc:  TOANNUAL: Index
    freq:  Annual (6)

    'dates:  (1)'    'Metric:  (1)'
    '31-Dec-1999'    [     28.5665]
```

- "Data Transformation and Frequency Conversion" on page 12-12
- "Using Time Series to Predict Equity Return" on page 12-25

See Also

`convertto` | `fints` | `todaily` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

todaily

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. `todaily` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as `'00:00'` for those days that did not previously exist in `oldfts`.

Empty (`[ ]`) passed as inputs for parameter pair values for `todaily` trigger the use of the defaults.

`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 at specific dates/times. No data manipulation occurs.

Parameter Name	Parameter Value	Description
`CalcMethod`	`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 the following parameter name/parameter value pairs are not supported.		
`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).
`BusDays`	`1`	(Default) Generates a daily financial time series that ranges from the first date to the last date in `oldfts` (excludes NYSE nonbusiness days and holidays and weekends based on `AltHolidays` and `Weekend`). NYSE market closures, holidays, and weekends are observed if `AltHolidays` and `Weekend` are not supplied or empty (`[]`).
`DateFilter`	`Absolute`	(Default) Displays all daily dates between the start and end dates of `oldfts`. Some dates may be disregarded if `BusDays = 1`. **Note** The default is to create a time series with every date at the specified periodicity, which is with `DateFilter` = `Absolute`. If you use `DateFilter` = `Relative`, the endpoint effects do not apply since only your data defines which dates appear in the output time series object.
`DateFilter`	`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.
`TimeSpec`	`Last`	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.

Parameter Name	Parameter Value	Description
`AltHolidays`		Vector of dates specifying an alternate set of market closure dates.
`AltHolidays`	`-1`	Excludes all holidays.
`Weekend`		Vector of length 7 containing 0's and 1's. The value `1` indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then `Weekend = [1 0 0 0 0 0 1]`.

Examples

Transform Time Series Object from Weekly to Daily Values

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
    '05-Feb-1999'     [     134.4882]
    '12-Feb-1999'     [     117.5581]
    '19-Feb-1999'     [     106.6683]
    '26-Feb-1999'     [     118.2912]
    '05-Mar-1999'     [     105.6835]
    '12-Mar-1999'     [     128.5836]
```

```
'19-Mar-1999'     [     115.1746]
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
'10-Dec-1999'     [      33.8457]
'17-Dec-1999'     [      33.3868]
'24-Dec-1999'     [      32.7737]
'31-Dec-1999'     [      28.5665]
```

Use `todaily` to obtain the daily aggregate for the `x0` times series.

```
x1 = todaily(x0)

x1 =

    desc:  TODAILY: Index
    freq:  Daily (1)

    'dates:  (252)'    'Metric:  (252)'
    '04-Jan-1999'      [           NaN]
    '05-Jan-1999'      [           NaN]
    '06-Jan-1999'      [           NaN]
    '07-Jan-1999'      [           NaN]
    '08-Jan-1999'      [       97.0847]
    '11-Jan-1999'      [           NaN]
    '12-Jan-1999'      [           NaN]
    '13-Jan-1999'      [           NaN]
    '14-Jan-1999'      [           NaN]
    '15-Jan-1999'      [      109.6312]
    '19-Jan-1999'      [           NaN]
    '20-Jan-1999'      [           NaN]
    '21-Jan-1999'      [           NaN]
    '22-Jan-1999'      [      105.5743]
    '25-Jan-1999'      [           NaN]
    '26-Jan-1999'      [           NaN]
    '27-Jan-1999'      [           NaN]
    '28-Jan-1999'      [           NaN]
    '29-Jan-1999'      [      108.4028]
    '01-Feb-1999'      [           NaN]
    '02-Feb-1999'      [           NaN]
    '03-Feb-1999'      [           NaN]
    '04-Feb-1999'      [           NaN]
    '05-Feb-1999'      [      134.4882]
    '08-Feb-1999'      [           NaN]
    '09-Feb-1999'      [           NaN]
    '10-Feb-1999'      [           NaN]
    '11-Feb-1999'      [           NaN]
    '12-Feb-1999'      [      117.5581]
    '16-Feb-1999'      [           NaN]
    '17-Feb-1999'      [           NaN]
    '18-Feb-1999'      [           NaN]
    '19-Feb-1999'      [      106.6683]
    '22-Feb-1999'      [           NaN]
    '23-Feb-1999'      [           NaN]
    '24-Feb-1999'      [           NaN]
```

```
'25-Feb-1999'      [           NaN]
'26-Feb-1999'      [      118.2912]
'01-Mar-1999'      [           NaN]
'02-Mar-1999'      [           NaN]
'03-Mar-1999'      [           NaN]
'04-Mar-1999'      [           NaN]
'05-Mar-1999'      [      105.6835]
'08-Mar-1999'      [           NaN]
'09-Mar-1999'      [           NaN]
'10-Mar-1999'      [           NaN]
'11-Mar-1999'      [           NaN]
'12-Mar-1999'      [      128.5836]
'15-Mar-1999'      [           NaN]
'16-Mar-1999'      [           NaN]
'17-Mar-1999'      [           NaN]
'18-Mar-1999'      [           NaN]
'19-Mar-1999'      [      115.1746]
'22-Mar-1999'      [           NaN]
'23-Mar-1999'      [           NaN]
'24-Mar-1999'      [           NaN]
'25-Mar-1999'      [           NaN]
'26-Mar-1999'      [      131.2854]
'29-Mar-1999'      [           NaN]
'30-Mar-1999'      [           NaN]
'31-Mar-1999'      [           NaN]
'01-Apr-1999'      [           NaN]
'05-Apr-1999'      [           NaN]
'06-Apr-1999'      [           NaN]
'07-Apr-1999'      [           NaN]
'08-Apr-1999'      [           NaN]
'09-Apr-1999'      [      123.1684]
'12-Apr-1999'      [           NaN]
'13-Apr-1999'      [           NaN]
'14-Apr-1999'      [           NaN]
'15-Apr-1999'      [           NaN]
'16-Apr-1999'      [      107.2975]
'19-Apr-1999'      [           NaN]
'20-Apr-1999'      [           NaN]
'21-Apr-1999'      [           NaN]
'22-Apr-1999'      [           NaN]
'23-Apr-1999'      [       91.5625]
'26-Apr-1999'      [           NaN]
'27-Apr-1999'      [           NaN]
'28-Apr-1999'      [           NaN]
```

```
'29-Apr-1999'      [           NaN]
'30-Apr-1999'      [       78.5738]
'03-May-1999'      [           NaN]
'04-May-1999'      [           NaN]
'05-May-1999'      [           NaN]
'06-May-1999'      [           NaN]
'07-May-1999'      [       65.2904]
'10-May-1999'      [           NaN]
'11-May-1999'      [           NaN]
'12-May-1999'      [           NaN]
'13-May-1999'      [           NaN]
'14-May-1999'      [       70.8581]
'17-May-1999'      [           NaN]
'18-May-1999'      [           NaN]
'19-May-1999'      [           NaN]
'20-May-1999'      [           NaN]
'21-May-1999'      [       72.4807]
'24-May-1999'      [           NaN]
'25-May-1999'      [           NaN]
'26-May-1999'      [           NaN]
'27-May-1999'      [           NaN]
'28-May-1999'      [       72.9190]
'01-Jun-1999'      [           NaN]
'02-Jun-1999'      [           NaN]
'03-Jun-1999'      [           NaN]
'04-Jun-1999'      [       64.3460]
'07-Jun-1999'      [           NaN]
'08-Jun-1999'      [           NaN]
'09-Jun-1999'      [           NaN]
'10-Jun-1999'      [           NaN]
'11-Jun-1999'      [       59.8743]
'14-Jun-1999'      [           NaN]
'15-Jun-1999'      [           NaN]
'16-Jun-1999'      [           NaN]
'17-Jun-1999'      [           NaN]
'18-Jun-1999'      [       55.0026]
'21-Jun-1999'      [           NaN]
'22-Jun-1999'      [           NaN]
'23-Jun-1999'      [           NaN]
'24-Jun-1999'      [           NaN]
'25-Jun-1999'      [       49.4032]
'28-Jun-1999'      [           NaN]
'29-Jun-1999'      [           NaN]
'30-Jun-1999'      [           NaN]
```

```
'01-Jul-1999'      [           NaN]
'02-Jul-1999'      [       49.9485]
'06-Jul-1999'      [           NaN]
'07-Jul-1999'      [           NaN]
'08-Jul-1999'      [           NaN]
'09-Jul-1999'      [       47.8061]
'12-Jul-1999'      [           NaN]
'13-Jul-1999'      [           NaN]
'14-Jul-1999'      [           NaN]
'15-Jul-1999'      [           NaN]
'16-Jul-1999'      [       61.0517]
'19-Jul-1999'      [           NaN]
'20-Jul-1999'      [           NaN]
'21-Jul-1999'      [           NaN]
'22-Jul-1999'      [           NaN]
'23-Jul-1999'      [       58.9313]
'26-Jul-1999'      [           NaN]
'27-Jul-1999'      [           NaN]
'28-Jul-1999'      [           NaN]
'29-Jul-1999'      [           NaN]
'30-Jul-1999'      [       53.9584]
'02-Aug-1999'      [           NaN]
'03-Aug-1999'      [           NaN]
'04-Aug-1999'      [           NaN]
'05-Aug-1999'      [           NaN]
'06-Aug-1999'      [       44.8472]
'09-Aug-1999'      [           NaN]
'10-Aug-1999'      [           NaN]
'11-Aug-1999'      [           NaN]
'12-Aug-1999'      [           NaN]
'13-Aug-1999'      [       45.0463]
'16-Aug-1999'      [           NaN]
'17-Aug-1999'      [           NaN]
'18-Aug-1999'      [           NaN]
'19-Aug-1999'      [           NaN]
'20-Aug-1999'      [       45.1088]
'23-Aug-1999'      [           NaN]
'24-Aug-1999'      [           NaN]
'25-Aug-1999'      [           NaN]
'26-Aug-1999'      [           NaN]
'27-Aug-1999'      [       56.4897]
'30-Aug-1999'      [           NaN]
'31-Aug-1999'      [           NaN]
'01-Sep-1999'      [           NaN]
```

```
'02-Sep-1999'      [           NaN]
'03-Sep-1999'      [       61.2449]
'07-Sep-1999'      [           NaN]
'08-Sep-1999'      [           NaN]
'09-Sep-1999'      [           NaN]
'10-Sep-1999'      [       58.1012]
'13-Sep-1999'      [           NaN]
'14-Sep-1999'      [           NaN]
'15-Sep-1999'      [           NaN]
'16-Sep-1999'      [           NaN]
'17-Sep-1999'      [       50.8974]
'20-Sep-1999'      [           NaN]
'21-Sep-1999'      [           NaN]
'22-Sep-1999'      [           NaN]
'23-Sep-1999'      [           NaN]
'24-Sep-1999'      [       46.5143]
'27-Sep-1999'      [           NaN]
'28-Sep-1999'      [           NaN]
'29-Sep-1999'      [           NaN]
'30-Sep-1999'      [           NaN]
'01-Oct-1999'      [       38.0806]
'04-Oct-1999'      [           NaN]
'05-Oct-1999'      [           NaN]
'06-Oct-1999'      [           NaN]
'07-Oct-1999'      [           NaN]
'08-Oct-1999'      [       33.6664]
'11-Oct-1999'      [           NaN]
'12-Oct-1999'      [           NaN]
'13-Oct-1999'      [           NaN]
'14-Oct-1999'      [           NaN]
'15-Oct-1999'      [       34.2992]
'18-Oct-1999'      [           NaN]
'19-Oct-1999'      [           NaN]
'20-Oct-1999'      [           NaN]
'21-Oct-1999'      [           NaN]
'22-Oct-1999'      [       33.4202]
'25-Oct-1999'      [           NaN]
'26-Oct-1999'      [           NaN]
'27-Oct-1999'      [           NaN]
'28-Oct-1999'      [           NaN]
'29-Oct-1999'      [       36.9287]
'01-Nov-1999'      [           NaN]
'02-Nov-1999'      [           NaN]
'03-Nov-1999'      [           NaN]
```

```
'04-Nov-1999'      [           NaN]
'05-Nov-1999'      [       35.1278]
'08-Nov-1999'      [           NaN]
'09-Nov-1999'      [           NaN]
'10-Nov-1999'      [           NaN]
'11-Nov-1999'      [           NaN]
'12-Nov-1999'      [       41.8128]
'15-Nov-1999'      [           NaN]
'16-Nov-1999'      [           NaN]
'17-Nov-1999'      [           NaN]
'18-Nov-1999'      [           NaN]
'19-Nov-1999'      [       35.8199]
'22-Nov-1999'      [           NaN]
'23-Nov-1999'      [           NaN]
'24-Nov-1999'      [           NaN]
'26-Nov-1999'      [       36.9495]
'29-Nov-1999'      [           NaN]
'30-Nov-1999'      [           NaN]
'01-Dec-1999'      [           NaN]
'02-Dec-1999'      [           NaN]
'03-Dec-1999'      [       36.2880]
'06-Dec-1999'      [           NaN]
'07-Dec-1999'      [           NaN]
'08-Dec-1999'      [           NaN]
'09-Dec-1999'      [           NaN]
'10-Dec-1999'      [       33.8457]
'13-Dec-1999'      [           NaN]
'14-Dec-1999'      [           NaN]
'15-Dec-1999'      [           NaN]
'16-Dec-1999'      [           NaN]
'17-Dec-1999'      [       33.3868]
'20-Dec-1999'      [           NaN]
'21-Dec-1999'      [           NaN]
'22-Dec-1999'      [           NaN]
'23-Dec-1999'      [           NaN]
'27-Dec-1999'      [           NaN]
'28-Dec-1999'      [           NaN]
'29-Dec-1999'      [           NaN]
'30-Dec-1999'      [           NaN]
'31-Dec-1999'      [       28.5665]
```

- “Data Transformation and Frequency Conversion” on page 12-12
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`convertto` | `fints` | `toannual` | `tomonthly` | `toquarterly` | `tosemi` | `toweekly`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

today

Current date

Syntax

```
Date = today
Date = today(outputType)
```

Description

`Date = today` returns the current date as a serial date number.

`Date = today(outputType)` returns the current date using an optional `outputType`. The type of output is determined by an optional `outputType` variable input.

Examples

Return the Current Date

Use `today` to return the current date with the default serial date number.

```
Date = today
```

```
Date = 736939
```

Use the optional argument `outputType` to return a datetime array.

```
Date = today('datetime')
```

```
Date = datetime
   01-Sep-2017
```

- “Data Transformation and Frequency Conversion” on page 12-12
- “Using Time Series to Predict Equity Return” on page 12-25

Input Arguments

`outputType` — Type of output

`'datenum'` (default) | character vector with values `'datetime'` or `'datenum'`

Type of output, specified as a character vector with values `'datetime'` or `'datenum'`.

If `outputType` is `'datenum'`, then `Date` is a serial date number. If `outputType` is `'datetime'`, then `Date` is a datetime array. By default, `outputType` is `'datenum'`.

Data Types: `char`

Output Arguments

`Date` — Current date

serial date number or datetime array

Current date, returned as a serial date number or datetime array, depending on the optional input argument `outputType`.

See Also

`datenum` | `datestr` | `datetime` | `now`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

todecimal

Fractional to decimal conversion

Syntax

```
usddec = todecimal(quote,fracpart)
```

Description

`usddec = todecimal(quote,fracpart)` returns the decimal equivalent, `usddec`, of a security whose price is normally quoted as a whole number and a fraction (`quote`). `fracpart` indicates the fractional base (denominator) with which the security is normally quoted (default = `32`).

Examples

In the *Wall Street Journal,* bond prices are quoted in fractional form based on a denominator of 32. For example, if you see the quoted price is 100:05 it means 100 5/32. To find the equivalent decimal value, enter

```
usddec = todecimal(100.05)

usddec =
   100.1563

usddec = todecimal(97.04, 16)

usddec =
   97.2500
```

Note The convention of using `.` (period) as a substitute for `:` (colon) in the input is adopted from Excel software.

See Also

`toquoted`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

tomonthly

Convert to monthly

Syntax

```
newfts = tomonthly(oldfts)

newfts = tomonthly(oldfts,'ParameterName',ParameterValue, ...)
```

Arguments

`oldfts`	Financial time series object.

Description

`newfts = tomonthly(oldfts)` converts a financial time series of any frequency to a monthly frequency. The default end-of-month day is the last business day of the month. `tomonthly` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as `00:00` for those days that did not previously exist in `oldfts`.

Empty (`[ ]`) passed as inputs for parameter pair values for `tomonthly` triggers the use of the defaults.

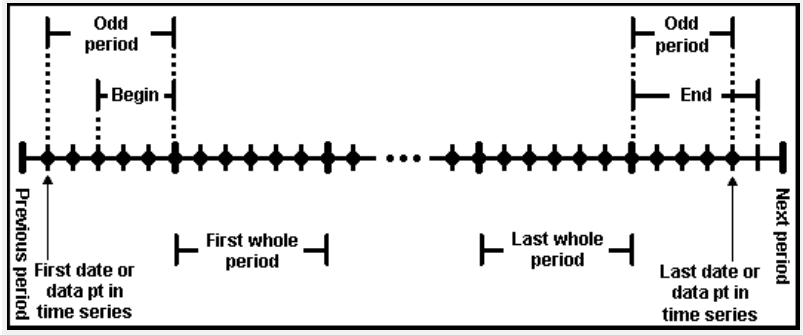
`newfts = tomonthly(oldfts,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values within each month. Data for missing dates are given the value 0.
	Exact	Returns the exact value at the end-of-month date. No data manipulation occurs.
	Nearest	(Default) Returns the values located at the end-of-month date. If there is missing data, 'Nearest' returns the nearest data point preceding the end-of-month date.
	SimpAvg	Returns an averaged monthly value that only takes into account dates with data (non-NaN) 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 the following parameter name/parameter value pairs are not supported.	
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 and weekends based on AltHolidays and Weekend). If an end-of-month date falls on a nonbusiness day or NYSE holiday, returns the last business day of the month. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ([]).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all monthly dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1. **Note** The default is to create a time series with every date at the specified periodicity, which is with DateFilter = Absolute. If you use DateFilter = Relative, the endpoint effects do not apply since only your data defines which dates appear in the output time series object.
	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 (for example, 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.



TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.
	Last	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.

Parameter Name	Parameter Value	Description
`AltHolidays`		Vector of dates specifying an alternate set of market closure dates.
	`-1`	Excludes all holidays.
`Weekend`		Vector of length 7 containing 0's and 1's. The value `1` indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then `Weekend = [1 0 0 0 0 0 1]`.

Examples

Transform Time Series Object from Weekly to Monthly Values

This example shows how to transform a time series object from weekly to monthly values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
    '05-Feb-1999'     [     134.4882]
    '12-Feb-1999'     [     117.5581]
    '19-Feb-1999'     [     106.6683]
```

```
'26-Feb-1999'     [     118.2912]
'05-Mar-1999'     [     105.6835]
'12-Mar-1999'     [     128.5836]
'19-Mar-1999'     [     115.1746]
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
'10-Dec-1999'     [      33.8457]
'17-Dec-1999'     [      33.3868]
```

```
    '24-Dec-1999'     [      32.7737]
    '31-Dec-1999'     [      28.5665]
```

Use `tomonthly` to obtain the monthly aggregate for the `x0` times series.

```
x1 = tomonthly(x0)
```

```
x1 =

    desc:  TOMONTHLY: Index
    freq:  Monthly (3)

    'dates:  (12)'    'Metric:  (12)'
    '29-Jan-1999'     [     108.4028]
    '26-Feb-1999'     [     118.2912]
    '31-Mar-1999'     [     131.2854]
    '30-Apr-1999'     [      78.5738]
    '28-May-1999'     [      72.9190]
    '30-Jun-1999'     [      49.4032]
    '30-Jul-1999'     [      53.9584]
    '31-Aug-1999'     [      56.4897]
    '30-Sep-1999'     [      46.5143]
    '29-Oct-1999'     [      36.9287]
    '30-Nov-1999'     [      36.9495]
    '31-Dec-1999'     [      28.5665]
```

- “Data Transformation and Frequency Conversion” on page 12-12
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`convertto` | `fints` | `toannual` | `todaily` | `toquarterly` | `tosemi` | `toweekly`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

toquarterly

Convert to quarterly

Syntax

```
newfts = toquarterly(oldfts)

newfts = toquarterly(oldfts,'ParameterName',ParameterValue, ...)
```

Arguments

`oldfts`	Financial time series object

Description

`newfts = toquarterly(oldfts)` converts a financial time series of any frequency to a quarterly frequency. The default quarterly days are the last business day of March, June, September, and December. `toquarterly` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as `00:00` for those days that did not previously exist in `oldfts`.

Empty (`[ ]`) passed as inputs for parameter pair values for `toquarterly` triggers the use of the defaults.

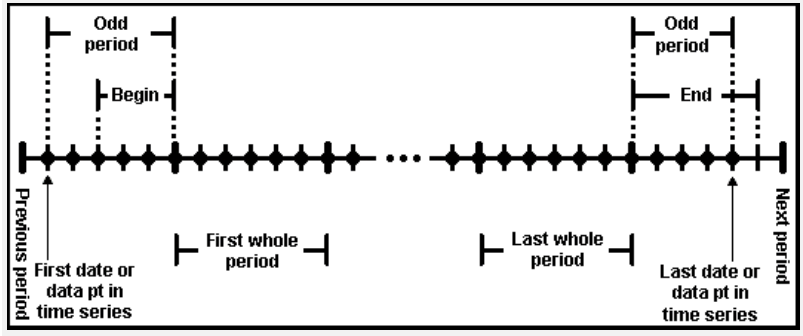
`newfts = toquarterly(oldfts,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

Parameter Name	Parameter Value	Description
`CalcMethod`	`CumSum`	Returns the cumulative sum of the values between each quarter. Data for missing dates are given the value `0`.
	`Exact`	Returns the exact value at the end-of-quarter date. No data manipulation occurs.
	`Nearest`	(Default) Returns the values located at the end-of-quarter date. If there is missing data, `Nearest` returns the nearest data point preceding the end-of-quarter date.
	`SimpAvg`	Returns an averaged quarterly value that only takes into account dates with data (non-`NaN`) 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 the following parameter name/parameter value pairs are not supported.	
`BusDays`	`0`	Generates a financial time series that ranges from (or between) the first date to the last date in `oldfts` (includes NYSE nonbusiness days and holidays).
	`1`	(Default) Generates a financial time series that ranges from the first date to the last date in `oldfts` (excludes NYSE nonbusiness days and holidays and weekends based on `AltHolidays` and `Weekend`). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter. NYSE market closures, holidays, and weekends are observed if `AltHolidays` and `Weekend` are not supplied or empty ([]).

Parameter Name	Parameter Value	Description
DateFilter	Absolute	(Default) Returns all quarterly dates between the start and end dates of oldfts. Some dates may be disregarded if BusDays = 1. **Note** The default is to create a time series with every date at the specified periodicity, which is with DateFilter = Absolute. If you use DateFilter = Relative, the endpoint effects do not apply since only your data defines which dates appear in the output time series object.
	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 (for example, 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 an odd quarter at the endpoints of the time series (before the first whole period and after the last whole period). `Begin` and `End` must be `-1` or any positive integer greater than or equal to `0`. A single value input for `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 endpoints for this function.



`TimeSpec`	`First`	Returns only the observation that occurs at the first (earliest) time for a specific date.

Parameter Name	Parameter Value	Description
	`Last`	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
`AltHolidays`		Vector of dates specifying an alternate set of market closure dates.
	`-1`	Excludes all holidays.
`Weekend`		Vector of length 7 containing 0's and 1's. The value `1` indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then `Weekend = [1 0 0 0 0 0 1]`.

Examples

Transform Time Series Object from Weekly to Quarterly Values

This example shows how to transform a time series object from weekly to quarterly values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
```

```
'05-Feb-1999'     [     134.4882]
'12-Feb-1999'     [     117.5581]
'19-Feb-1999'     [     106.6683]
'26-Feb-1999'     [     118.2912]
'05-Mar-1999'     [     105.6835]
'12-Mar-1999'     [     128.5836]
'19-Mar-1999'     [     115.1746]
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
```

```
    '10-Dec-1999'      [      33.8457]
    '17-Dec-1999'      [      33.3868]
    '24-Dec-1999'      [      32.7737]
    '31-Dec-1999'      [      28.5665]
```

Use `toquarterly` to obtain the quarterly aggregate for the `x0` times series.

```
x1 = toquarterly(x0)
```

```
x1 =

    desc:  TOQUARTERLY: Index
    freq:  Quarterly (4)

    'dates:  (4)'    'Metric:  (4)'
    '31-Mar-1999'    [    131.2854]
    '30-Jun-1999'    [     49.4032]
    '30-Sep-1999'    [     46.5143]
    '31-Dec-1999'    [     28.5665]
```

Transform Time Series Object from Weekly to Quarterly Values Including NYSE Nonbusiness Days and Holidays

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
```

```
'05-Feb-1999'     [     134.4882]
'12-Feb-1999'     [     117.5581]
'19-Feb-1999'     [     106.6683]
'26-Feb-1999'     [     118.2912]
'05-Mar-1999'     [     105.6835]
'12-Mar-1999'     [     128.5836]
'19-Mar-1999'     [     115.1746]
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
```

```
    '10-Dec-1999'      [      33.8457]
    '17-Dec-1999'      [      33.3868]
    '24-Dec-1999'      [      32.7737]
    '31-Dec-1999'      [      28.5665]
```

Use `toquarterly` with the optional `BusDays` argument set to `0` to obtain the quarterly cumulative sums for the `x0` times series that includes NYSE nonbusiness days and holidays.

```
x1 = toquarterly(x0,'CalcMethod','CumSum','Busdays',0)
```

```
x1 =

    desc:  TOQUARTERLY: Index
    freq:  Quarterly (4)

    'dates:  (4)'    'Metric:  (4)'
    '31-Mar-1999'    [  1.4763e+03]
    '30-Jun-1999'    [  1.0415e+03]
    '30-Sep-1999'    [    679.9459]
    '31-Dec-1999'    [    490.9659]
```

- “Data Transformation and Frequency Conversion” on page 12-12
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`convertto` | `fints` | `toannual` | `todaily` | `tomonthly` | `tosemi` | `toweekly`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

toquoted

Decimal to fractional conversion

Syntax

```
quote = toquoted(usddec,fracpart)
```

Description

`quote = toquoted(usddec,fracpart)` returns the fractional equivalent, `quote`, of the decimal figure, `usddec`, based on the fractional base (denominator), `fracpart`. The fractional bases are the ones used for quoting equity prices in the United States (denominator 2, 4, 8, 16, or 32). If `fracpart` is not entered, the denominator 32 is assumed.

Examples

A United States equity price in decimal form is 101.625. To convert this to fractional form in eighths of a dollar:

```
quote = toquoted(101.625, 8)
```

```
quote =
        101.05
```

The answer is interpreted as 101 5/8.

Note The convention of using `.` (period) as a substitute for `:` (colon) in the output is adopted from Excel software.

See Also

`todecimal`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

tosemi

Convert to semiannual

Syntax

```
newfts = tosemi(oldfts)

newfts = tosemi(oldfts,'ParameterName',ParameterValue, ...)
```

Arguments

`oldfts`	Financial time series object.

Description

`newfts = tosemi(oldfts)` converts a financial time series of any frequency to a semiannual frequency. The default semiannual days are the last business day of June and December. `tosemi` uses `holidays.m` to determine valid trading days.

Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as `00:00` for those days that did not previously exist in `oldfts`.

Empty (`[ ]`) passed as inputs for parameter pair values for `tosemi` triggers the use of the defaults.

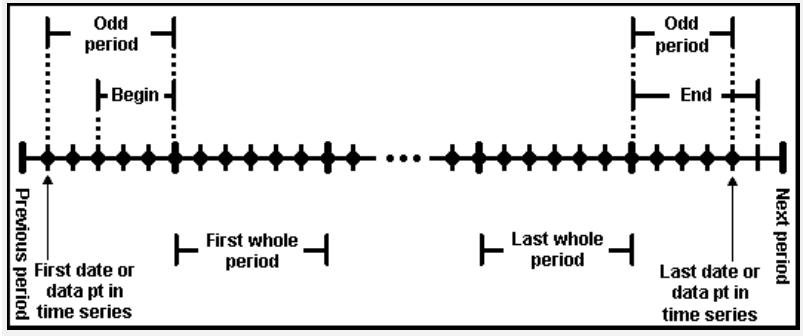
`newfts = tosemi(oldfts,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input, as specified in the following table.

Parameter Name	Parameter Value	Description
CalcMethod	CumSum	Returns the cumulative sum of the values within each semiannual period. Data for missing dates are given the value 0.
	Exact	Returns the exact value at the end-of-period date. No data manipulation occurs.
	Nearest	(Default) Returns the values located at the end-of-period date. If there is missing data, Nearest returns the nearest data point preceding the end-of-period date.
	SimpAvg	Returns an averaged semiannual value that only takes into account dates with data (non-NaN) 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 the following parameter name/parameter value pairs are not supported.	
BusDays	0	Generates a financial time series that ranges from (or between) the first date to the last date in oldfts (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a financial time series that ranges from the first date to the last date in oldfts (excludes NYSE nonbusiness days and holidays and weekends based on AltHolidays and Weekend). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ([]).

Parameter Name	Parameter Value	Description
`DateFilter`	`Absolute`	(Default) Returns all semiannual dates between the start and end dates of `oldfts`. Some dates may be disregarded if `BusDays = 1`. **Note** The default is to create a time series with every date at the specified periodicity, which is with `DateFilter` = `Absolute`. If you use `DateFilter` = `Relative`, the endpoint effects do not apply since only your data defines which dates appear in the output time series object.
	`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 (for example, `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 endpoints of the time series (before the first whole period and after the last whole period). Begin and End must be -1 or any positive integer greater than or equal to 0. A single value input for 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 endpoints for this function.



TimeSpec	First	Returns only the observation that occurs at the first (earliest) time for a specific date.

Parameter Name	Parameter Value	Description
	`Last`	(Default) Returns only the observation that occurs at the last (latest) time for a specific date.
`AltHolidays`		Vector of dates specifying an alternate set of market closure dates.
	`-1`	Excludes all holidays.
`Weekend`		Vector of length 7 containing 0's and 1's. The value `1` indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then `Weekend = [1 0 0 0 0 0 1]`.

Examples

Transform Time Series Object from Weekly to Semiannual Values

This example shows how to transform a time series object from weekly to semiannual values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a weekly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'w', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
```

```
'05-Feb-1999'     [     134.4882]
'12-Feb-1999'     [     117.5581]
'19-Feb-1999'     [     106.6683]
'26-Feb-1999'     [     118.2912]
'05-Mar-1999'     [     105.6835]
'12-Mar-1999'     [     128.5836]
'19-Mar-1999'     [     115.1746]
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
```

```
    '10-Dec-1999'     [      33.8457]
    '17-Dec-1999'     [      33.3868]
    '24-Dec-1999'     [      32.7737]
    '31-Dec-1999'     [      28.5665]
```

Use `tosemi` to obtain the semiannual aggregate for the `x0` times series.

```
x1 = tosemi(x0)


x1 =

    desc:  TOSEMI: Index
    freq:  Semiannual (5)

    'dates:  (2)'    'Metric:  (2)'
    '30-Jun-1999'    [     49.4032]
    '31-Dec-1999'    [     28.5665]
```

- "Data Transformation and Frequency Conversion" on page 12-12
- "Using Time Series to Predict Equity Return" on page 12-25

See Also

`convertto` | `fints` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `toweekly`

Topics

"Data Transformation and Frequency Conversion" on page 12-12
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

totalreturnprice

Total return price time series

Syntax

```
Return = totalreturnprice(Price,Action,Dividend)
```

Arguments

`Price`	`Price` can be a table or a `NUMOBS`-by-`2` matrix. If `Price` is a table, the dates can either be serial date numbers, date character vectors. If `Price` is a number of observations `NUMOBS`-by-`2` matrix of price data, column 1 contains MATLAB serial date numbers and column 2 contains price values.
`Action`	`Action` can be a table or a `NUMOBS`-by-`2` matrix. If `Action` is a table, the dates can either be serial date numbers, date character vectors. If `Action` is a `NUMOBS`-by-`2` matrix of price data, column 1 contains MATLAB serial date numbers and column 2 contains split ratios.
`Dividend`	`Dividend` can be a table or a `NUMOBS`-by-`2` matrix. If `Dividend` is a table, the dates can either be serial date numbers, date character vectors. If `Dividend` is a `NUMOBS`-by-`2` matrix of price data, column 1 contains MATLAB serial date numbers and column 2 contains dividend payouts.

The number of observations (`NUMOBS`) for the three input arguments 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.

If all three inputs are matrices, then `Return` is a `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.

However, if any inputs are tables, then `Return` will also be a table. The class of the first column depends on the classes used in the input tables. If any tables used datetimes for dates, then `Returns` will have datetimes in the first column. If there were no datetimes in the inputs, but if any inputs used date character vectors, then `Returns` will use date character vectors in the first column. For any other case, serial date numbers are used.

Examples

Compute Return Using datetime Input for Price and Action

Compute `Return` returned as a table using datetime input in tables for `Price` and `Action`.

```
act = [732313, 2; 732314 ,2];
div = [732313, 0.0800; 732314, 0.0800];
prc = [732313, 12; 732314, 13];

prcTableDateTime=table(datetime(prc(:,1),'ConvertFrom','datenum'),prc(:,2));
acttableString=table(datestr(act(:,1)),act(:,2));
divTableNum = array2table(div);
Return = totalreturnprice(prcTableDateTime,acttableString,divTableNum)
```

```
Return=2x2 table null
            Date             Return
    ____________________     ______

    01-Jan-2005 00:00:00          1
    02-Jan-2005 00:00:00     1.0833
```

- “Portfolio Construction Examples” on page 3-7

See Also

`datetime` | `periodicreturns`

Topics

“Portfolio Construction Examples” on page 3-7
“Portfolio Optimization Functions” on page 3-4

Introduced before R2006a

toweekly

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. `toweekly` uses `holidays.m` to determine valid trading days.

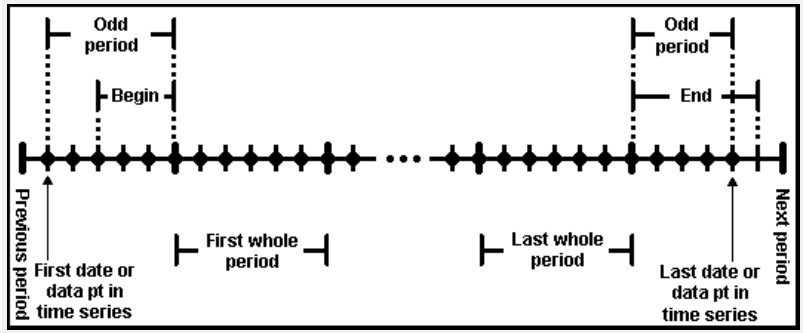
Note If `oldfts` contains time-of-day information, `newfts` displays the time-of-day as `00:00` for those days that did not previously exist in `oldfts`.

Empty (`[ ]`) passed as inputs for parameter pair values for `toweekly` triggers the use of the defaults.

`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.
	SimpAvg	Returns an averaged weekly value that only takes into account dates with data (non-NaN) 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 the following parameter name/parameter value pairs are not supported.	
BusDays	0	Generates a financial time series that ranges from (or between) the first date to the last date in oldfts (includes NYSE nonbusiness days and holidays).
	1	(Default) Generates a financial time series that ranges from the first date to the last date in oldfts (excludes NYSE nonbusiness days and holidays and weekends based on AltHolidays and Weekend). If an end-of-quarter date falls on a nonbusiness day or NYSE holiday, returns the last business day of the quarter. NYSE market closures, holidays, and weekends are observed if AltHolidays and Weekend are not supplied or empty ([]).

Parameter Name	Parameter Value	Description
`DateFilter`	`Absolute`	(Default) Returns all weekly dates between the start and end dates of `oldfts`. Some dates may be disregarded if `BusDays = 1`. **Note** The default is to create a time series with every date at the specified periodicity, which is with `DateFilter` = `Absolute`. If you use `DateFilter` = `Relative`, the endpoint effects do not apply since only your data defines which dates appear in the output time series object.
	`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 an odd week at the endpoints of the time series (before the first whole period and after the last whole period). `Begin` and `End` must be `-1` or any positive integer greater than or equal to `0`. A single value input for `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 endpoints 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.
`AltHolidays`		Vector of dates specifying an alternate set of market closure dates.
	`-1`	Excludes all holidays.

Parameter Name	Parameter Value	Description
`Weekend`		Vector of length 7 containing 0's and 1's. The value `1` indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days (default) then `Weekend = [1 0 0 0 0 0 1]`.

Examples

Transform Time Series Object from Quarterly to Weekly Values

This example shows how to transform a time series object from quarterly to weekly values.

Load the data from the file `predict_ret_data.mat` and use the `fints` function to create a time series object with a quarterly frequency.

```
load predict_ret_data.mat
x0 = fints(expdates, expdata, {'Metric'}, 'q', 'Index')
```

```
x0 =

    desc:  Index
    freq:  Quarterly (4)

    'dates:  (53)'    'Metric:  (53)'
    '01-Jan-1999'     [      97.8872]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
    '05-Feb-1999'     [     134.4882]
    '12-Feb-1999'     [     117.5581]
    '19-Feb-1999'     [     106.6683]
    '26-Feb-1999'     [     118.2912]
    '05-Mar-1999'     [     105.6835]
    '12-Mar-1999'     [     128.5836]
    '19-Mar-1999'     [     115.1746]
```

```
'26-Mar-1999'     [     131.2854]
'02-Apr-1999'     [     130.7116]
'09-Apr-1999'     [     123.1684]
'16-Apr-1999'     [     107.2975]
'23-Apr-1999'     [      91.5625]
'30-Apr-1999'     [      78.5738]
'07-May-1999'     [      65.2904]
'14-May-1999'     [      70.8581]
'21-May-1999'     [      72.4807]
'28-May-1999'     [      72.9190]
'04-Jun-1999'     [      64.3460]
'11-Jun-1999'     [      59.8743]
'18-Jun-1999'     [      55.0026]
'25-Jun-1999'     [      49.4032]
'02-Jul-1999'     [      49.9485]
'09-Jul-1999'     [      47.8061]
'16-Jul-1999'     [      61.0517]
'23-Jul-1999'     [      58.9313]
'30-Jul-1999'     [      53.9584]
'06-Aug-1999'     [      44.8472]
'13-Aug-1999'     [      45.0463]
'20-Aug-1999'     [      45.1088]
'27-Aug-1999'     [      56.4897]
'03-Sep-1999'     [      61.2449]
'10-Sep-1999'     [      58.1012]
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
'10-Dec-1999'     [      33.8457]
'17-Dec-1999'     [      33.3868]
'24-Dec-1999'     [      32.7737]
'31-Dec-1999'     [      28.5665]
```

Use `toweekly` to obtain the weekly aggregate for the `x0` times series.

```
x1 = toweekly(x0)
```

```
x1 =

    desc:  TOWEEKLY: Index
    freq:  Weekly (2)

    'dates:  (53)'    'Metric:  (53)'
    '31-Dec-1998'     [          NaN]
    '08-Jan-1999'     [      97.0847]
    '15-Jan-1999'     [     109.6312]
    '22-Jan-1999'     [     105.5743]
    '29-Jan-1999'     [     108.4028]
    '05-Feb-1999'     [     134.4882]
    '12-Feb-1999'     [     117.5581]
    '19-Feb-1999'     [     106.6683]
    '26-Feb-1999'     [     118.2912]
    '05-Mar-1999'     [     105.6835]
    '12-Mar-1999'     [     128.5836]
    '19-Mar-1999'     [     115.1746]
    '26-Mar-1999'     [     131.2854]
    '01-Apr-1999'     [          NaN]
    '09-Apr-1999'     [     123.1684]
    '16-Apr-1999'     [     107.2975]
    '23-Apr-1999'     [      91.5625]
    '30-Apr-1999'     [      78.5738]
    '07-May-1999'     [      65.2904]
    '14-May-1999'     [      70.8581]
    '21-May-1999'     [      72.4807]
    '28-May-1999'     [      72.9190]
    '04-Jun-1999'     [      64.3460]
    '11-Jun-1999'     [      59.8743]
    '18-Jun-1999'     [      55.0026]
    '25-Jun-1999'     [      49.4032]
    '02-Jul-1999'     [      49.9485]
    '09-Jul-1999'     [      47.8061]
    '16-Jul-1999'     [      61.0517]
    '23-Jul-1999'     [      58.9313]
    '30-Jul-1999'     [      53.9584]
    '06-Aug-1999'     [      44.8472]
    '13-Aug-1999'     [      45.0463]
    '20-Aug-1999'     [      45.1088]
    '27-Aug-1999'     [      56.4897]
    '03-Sep-1999'     [      61.2449]
    '10-Sep-1999'     [      58.1012]
```

```
'17-Sep-1999'     [      50.8974]
'24-Sep-1999'     [      46.5143]
'01-Oct-1999'     [      38.0806]
'08-Oct-1999'     [      33.6664]
'15-Oct-1999'     [      34.2992]
'22-Oct-1999'     [      33.4202]
'29-Oct-1999'     [      36.9287]
'05-Nov-1999'     [      35.1278]
'12-Nov-1999'     [      41.8128]
'19-Nov-1999'     [      35.8199]
'26-Nov-1999'     [      36.9495]
'03-Dec-1999'     [      36.2880]
'10-Dec-1999'     [      33.8457]
'17-Dec-1999'     [      33.3868]
'23-Dec-1999'     [          NaN]
'31-Dec-1999'     [      28.5665]
```

- “Data Transformation and Frequency Conversion” on page 12-12
- “Using Time Series to Predict Equity Return” on page 12-25

See Also

`convertto` | `fints` | `toannual` | `todaily` | `tomonthly` | `toquarterly` | `tosemi`

Topics

“Data Transformation and Frequency Conversion” on page 12-12
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

tr2bonds

Term-structure parameters given Treasury bond parameters

Syntax

```
[Bonds,Prices,Yields] = tr2bonds(TreasuryMatrix,Settle)
[Bonds,Prices,Yields] = tr2bonds(___,Settle)
```

Description

`[Bonds,Prices,Yields] = tr2bonds(TreasuryMatrix,Settle)` returns term-structure parameters (`Bonds`, `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,Prices,Yields] = tr2bonds(___,Settle)` adds an optional argument for `Settle`.

Examples

Return Term-Structure Parameters Given Treasury Bond Parameters

This example shows how to return term-structure parameters (bond information, prices, and yields) sorted by ascending maturity date, given Treasury bond market parameters for December 22, 1997.

```
Matrix =[0.0650 datenum('15-apr-1999')  101.03125 101.09375 0.0564
         0.05125 datenum('17-dec-1998')  99.4375   99.5     0.0563
         0.0625 datenum('30-jul-1998')  100.3125  100.375   0.0560
         0.06125 datenum('26-mar-1998') 100.09375 100.15625 0.0546];

[Bonds, Prices, Yields] = tr2bonds(Matrix)
```

```
Bonds =

   1.0e+05 *

    7.2984    0.0000    0.0010    0.0000         0    0.0000
    7.2997    0.0000    0.0010    0.0000         0    0.0000
    7.3011    0.0000    0.0010    0.0000         0    0.0000
    7.3022    0.0000    0.0010    0.0000         0    0.0000


Prices =

  100.1563
  100.3750
   99.5000
  101.0938


Yields =

    0.0546
    0.0560
    0.0563
    0.0564
```

Return Term-Structure Parameters Given Treasury Bond Parameters Using datetime Input

This example shows how to use `datetime` input to return term-structure parameters (bond information, prices, and yields) sorted by ascending maturity date, given Treasury bond market parameters for December 22, 1997.

```
Matrix =[0.0650 datenum('15-apr-1999')  101.03125 101.09375 0.0564
         0.05125 datenum('17-dec-1998')  99.4375   99.5     0.0563
         0.0625 datenum('30-jul-1998')  100.3125  100.375   0.0560
         0.06125 datenum('26-mar-1998') 100.09375 100.15625 0.0546];

t=array2table(Matrix);
t.Matrix2=datetime(t{:,2},'ConvertFrom','datenum','Locale','en_US');
[Bonds, Prices, Yields] = tr2bonds(t,datetime('1-Jan-1997','Locale','en_US'))

Bonds=4x6 table
          Maturity          CouponRate    Face    Period    Basis    EndMonthRule
```

```
    _____________________    __________    ____    ______    _____    ____________

    26-Mar-1998 00:00:00    0.06125       100     2         0        1
    30-Jul-1998 00:00:00     0.0625       100     2         0        1
    17-Dec-1998 00:00:00    0.05125       100     2         0        1
    15-Apr-1999 00:00:00      0.065       100     2         0        1


Prices =

  100.1563
  100.3750
   99.5000
  101.0938


Yields =

    0.0598
    0.0599
    0.0540
    0.0598
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`TreasuryMatrix` — Treasury bond parameters

table | matrix

Treasury bond parameters, specified as a 5-column table or a `NumBonds`-by-`5` matrix of bond information where the table columns or matrix columns contains:

- `CouponRate` (Required) Coupon rate of the Treasury bond, specified as a decimal indicating the coupon rates for each bond in the portfolio.
- `Maturity` (Required) Maturity date of the Treasury bond, specified as a serial date number when using a matrix. Use `datenum` to convert date character vectors to serial date numbers. If the input `TreasuryMatrix` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.

- `Bid` (Required) Bid prices, specified using an integer-decimal form for each bond in the portfolio.
- `Asked` (Required) Asked prices, specified using an integer-decimal form for each bond in the portfolio.
- `AskYield` (Required) Quoted ask yield, specified using a decimal form for each bond in the portfolio.

Data Types: `double` | `table`

`Settle` — Settlement date of Treasury bond
serial date number | date character vector | datetime

(Optional) Settlement date of the Treasury bond, specified as a scalar or a `NINST`-by-`1` vector of serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

Output Arguments

`Bonds` — Coupon bond information
numeric

Coupon bond information, returned as a table or matrix depending on the `TreasuryMatrix` input.

When `TreasuryMatrix` is a table, `Bonds` is also a table, and the variable type for the `Maturity` dates in `Bonds` (column 1) matches the variable type for `Maturity` in `TreasuryMatrix`.

When `TreasuryMatrix` input is a `n`-by-`5` matrix, then each row describes a bond.

The parameters or columns returned for `Bonds` are:

- `Maturity` (Column 1) Maturity date for each bond in the portfolio as a serial date number. The format of the dates matches the format used for `Maturity` in `TreasuryMatrix` (serial date number, date character vector, or datetime array).
- `CouponRate` (Column 2) Coupon rate for each bond in the portfolio in decimal form.

- `Face` (Column 3, Optional) Face or par value for each bond in the portfolio. The default is `100`.
- `Period` (Column 4, Optional) Number of coupon payments per year for each bond in the portfolio with allowed values: `1`, `2`, `3`, `4`, `6`, and `12`. The default is `2`, unless you are dealing with zero coupons, then `Period` is `0` instead of `2`.
- `Basis` (Column 5, Optional) Day-count basis for each bond in the portfolio with possible values:
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)
  - 11 = 30/360E (ICMA)
  - 12 = actual/365 (ISDA)
  - 13 = BUS/252
  - For more information, see **basis** on page Glossary-0 .
- `EndMonthRule` (Column 6, Optional) End-of-month rule flag for each bond in the portfolio. 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, meaning that a bond's coupon payment date is always the last actual day of the month. The default is `1`.

`Prices` — Bond prices

numeric

Bond prices, returned as 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` — Bond yields
numeric

Bond yields, returned as 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 the optional input argument `Settle` is used, `Yields` is computed as a semiannual yield to maturity. If the input `Settle` is not used, the quoted input yields are used.

See Also

`datetime` | `tbl2bond` | `zbtprice` | `zbtyield`

Topics

"Term Structure of Interest Rates" on page 2-45
"Treasury Bills Defined" on page 2-40

Introduced before R2006a

transprob

Estimate transition probabilities from credit ratings data

Syntax

```
[transMat,sampleTotals,idTotals] = transprob(data)
[transMat,sampleTotals,idTotals] = transprob(___,Name, Value)
```

Description

`[transMat,sampleTotals,idTotals] = transprob(data)` constructs a transition matrix from historical data of credit ratings.

`[transMat,sampleTotals,idTotals] = transprob(___,Name, Value)` adds optional name-value pair arguments.

Examples

Construct a Transition Matrix From a Table of Historical Data of Credit Ratings

Using the historical credit rating table as input data from `Data_TransProb.mat` display the first ten rows and compute the transition matrix:

```
load Data_TransProb
data(1:10,:)
```

```
ans=10x3 table
        ID            Date          Rating
    __________    ______________    ______

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '12-May-1986'    'B'
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'
```

```
    '00013326'    '09-Feb-1985'    'A'
    '00013326'    '24-Feb-1994'    'AA'
    '00013326'    '10-Nov-2000'    'BBB'
    '00014413'    '23-Dec-1982'    'B'
    '00014413'    '20-Apr-1988'    'BB'
    '00014413'    '16-Jan-1998'    'B'


% Estimate transition probabilities with default settings
transMat = transprob(data)

transMat =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Using the historical credit rating table input data from `Data_TransProb.mat`, compute the transition matrix using the `cohort` algorithm:

```
%Estimate transition probabilities with 'cohort' algorithm
transMatCoh = transprob(data,'algorithm','cohort')

transMatCoh =
```

```
Columns 1 through 7

   93.1345    5.9335    0.7456    0.1553    0.0311         0         0
    1.7359   92.9198    4.5446    0.6046    0.1560         0         0
    0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0
    0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421
    0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873
         0         0    0.0761    0.7230    7.9909   86.1872    2.7397
         0         0         0    0.3094    1.8561    4.5630   80.8971
         0         0         0         0         0         0         0

Column 8

         0
    0.0390
    0.0725
    0.2103
    0.6409
    2.2831
   12.3743
  100.0000
```

Using the historical credit rating data with ratings investment grade (`'IG'`), speculative grade (`'SG'`), and default (`'D'`), from `Data_TransProb.mat` display the first ten rows and compute the transition matrix:

```
dataIGSG(1:10,:)

ans=10x3 table
        ID             Date          Rating
    ___________    _____________    _______

    '00011253'    '04-Apr-1983'    'IG'
    '00012751'    '17-Feb-1985'    'SG'
    '00012751'    '19-May-1986'    'D'
    '00014690'    '17-Jan-1983'    'IG'
    '00012144'    '21-Nov-1984'    'IG'
    '00012144'    '25-Mar-1992'    'SG'
    '00012144'    '07-May-1994'    'IG'
    '00012144'    '23-Jan-2000'    'SG'
    '00012144'    '20-Aug-2001'    'IG'
    '00012937'    '07-Feb-1984'    'IG'
```

```
transMatIGSG = transprob(dataIGSG,'labels',{'IG','SG','D'})

transMatIGSG =

   98.6719    1.2020    0.1261
    3.5781   93.3318    3.0901
         0         0  100.0000
```

Using the historical credit rating data with numeric ratings for investment grade (1), speculative grade (2), and default (3), from `Data_TransProb.mat` display the first ten rows and compute the transition matrix:

```
dataIGSGnum(1:10,:)

ans=10x3 table
        ID             Date          Rating
    ___________    _____________    ______

    '00011253'    '04-Apr-1983'    1
    '00012751'    '17-Feb-1985'    2
    '00012751'    '19-May-1986'    3
    '00014690'    '17-Jan-1983'    1
    '00012144'    '21-Nov-1984'    1
    '00012144'    '25-Mar-1992'    2
    '00012144'    '07-May-1994'    1
    '00012144'    '23-Jan-2000'    2
    '00012144'    '20-Aug-2001'    1
    '00012937'    '07-Feb-1984'    1


transMatIGSGnum = transprob(dataIGSGnum,'labels',{1,2,3})

transMatIGSGnum =

   98.6719    1.2020    0.1261
    3.5781   93.3318    3.0901
         0         0  100.0000
```

Create a Transition Matrix Using a Cell Array for Historical Data of Credit Ratings

Using a MATLAB® table containing the historical credit rating cell array input data (`dataCellFormat`) from `Data_TransProb.mat`, estimate the transition probabilities with default settings.

```
load Data_TransProb
transMat = transprob(dataCellFormat)
```

```
transMat =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Using the historical credit rating cell array input data (`dataCellFormat`), compute the transition matrix using the `cohort` algorithm:

```
%Estimate transition probabilities with 'cohort' algorithm
transMatCoh = transprob(dataCellFormat,'algorithm','cohort')
```

```
transMatCoh =

  Columns 1 through 7

   93.1345    5.9335    0.7456    0.1553    0.0311         0         0
```

```
    1.7359   92.9198    4.5446    0.6046    0.1560         0         0
    0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0
    0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421
    0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873
         0         0    0.0761    0.7230    7.9909   86.1872    2.7397
         0         0         0    0.3094    1.8561    4.5630   80.8971
         0         0         0         0         0         0         0

  Column 8

         0
    0.0390
    0.0725
    0.2103
    0.6409
    2.2831
   12.3743
  100.0000
```

- "Estimation of Transition Probabilities" on page 8-2
- "Estimate Transition Probabilities for Different Rating Scales" on page 8-5

Input Arguments

`data` — Credit migration data

table | cell array of character vectors | preprocessed data structure

Using `transprob` to estimate transition probabilities given credit ratings historical data (that is, credit migration data), the `data` input can be one of the following:

- An `nRecords`-by-`3` MATLAB table containing the historical credit ratings data of the form:

```
    ID          Date          Rating
    _________   ___________   ______
    '00010283'  '10-Nov-1984'  'CCC'
    '00010283'  '12-May-1986'  'B'
    '00010283'  '29-Jun-1988'  'CCC'
    '00010283'  '12-Dec-1991'  'D'
    '00013326'  '09-Feb-1985'  'A'
```

```
'00013326'  '24-Feb-1994'  'AA'
'00013326'  '10-Nov-2000'  'BBB'
'00014413'  '23-Dec-1982'  'B'
```

where each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). Column 3 is the rating assigned to the corresponding ID on the corresponding date. All information corresponding to the same ID must be stored in contiguous rows. Sorting this information by date is not required, but recommended for efficiency. When using a MATLAB table input, the names of the columns are irrelevant, but the ID, date and rating information are assumed to be in the first, second, and third columns, respectively. Also, when using a table input, the first and third columns can be categorical arrays, and the second can be a datetime array. The following summarizes the supported data types for table input:

Data Input Type	ID (1st Column)	Date (2nd Column)	Rating (3rd Column)
Table	• Numeric array • Cell array of character vectors • Categorical array	• Numeric array • Cell array of character vectors • Datetime array	• Numeric array • Cell array of character vectors • Categorical array

- An `nRecords`-by-`3` cell array of character vectors containing the historical credit ratings data of the form:

```
'00010283'  '10-Nov-1984'  'CCC'
'00010283'  '12-May-1986'  'B'
'00010283'  '29-Jun-1988'  'CCC'
'00010283'  '12-Dec-1991'  'D'
'00013326'  '09-Feb-1985'  'A'
'00013326'  '24-Feb-1994'  'AA'
'00013326'  '10-Nov-2000'  'BBB'
'00014413'  '23-Dec-1982'  'B'
```

where each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). Column 3 is the rating assigned to the corresponding ID on the corresponding date. All information corresponding to the same ID must be stored in contiguous rows. Sorting this information by date is not required, but recommended for efficiency. IDs, dates, and ratings are stored in character vector format, but they can also be entered in numeric format. The following summarizes the supported data types for cell array input:

Data Input Type	ID (1st Column)	Date (2nd Column)	Rating (3rd Column)
Cell	• Numeric elements • Character vector elements	• Numeric elements • Character vector elements	• Numeric elements • Character vector elements

- A preprocessed data structure obtained using `transprobprep`. This data structure contains the fields `'idStart'`, `'numericDates'`, `'numericRatings'`, and `'ratingsLabels'`.

Data Types: `table` | `cell` | `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `transMat = transprob(data,'algorithm','cohort')`

`algorithm` — Estimation algorithm

`'duration'` (default) | character vector with values are `'duration'` or `'cohort'`

Estimation algorithm, specified as a character vector with a value of `'duration'` or `'cohort'`.

Data Types: `char`

`endDate` — End date of the estimation time window

latest date in `data` (default) | character vector | serial date number

End date of the estimation time window, specified as a date character vector or a serial date number. The `endDate` cannot be a date before the `startDate`.

Data Types: `char` | `double`

`labels` — Credit-rating scale

`{'AAA','AA','A','BBB','BB','B','CCC','D'}` (default) | cell array of character vectors

Credit-rating scale, specified as a `nRatings`-by-`1`, or `1`-by-`nRatings` cell array of character vectors.

`labels` must be consistent with the ratings labels used in the third column of `data`. Use a cell array of numbers for numeric ratings, and a cell array for character vectors for categorical ratings.

Note When the input argument `data` is a preprocessed data structure obtained from a previous call to `transprobprep`, this optional input for `'labels` is unused because the labels in the `'ratingsLabels'` field of `transprobprep` take priority.

Data Types: `cell`

`snapsPerYear` — Number of credit-rating snapshots per year
`1` (default) | numeric values are `1`, `2`, `3`, `4`, `6`, or `12`

Number of credit-rating snapshots per year to be considered for the estimation, specified as a numeric value of `1`, `2`, `3`, `4`, `6`, or `12`.

Note This parameter is only used with the `'cohort'` `algorithm`.

Data Types: `double`

`startDate` — Start date of the estimation time window
earliest date in `data` (default) | character vector | serial date number

Start date of the estimation time window, specified as a date character vector or a serial date number.

Data Types: `char` | `double`

`transInterval` — Length of the transition interval in years
`1` (one year transition probability) (default) | numeric

Length of the transition interval, in years, specified as a numeric value.

Data Types: `double`

Output Arguments

`transMat` — Matrix of transition probabilities in percent

matrix

Matrix of transition probabilities in percent, returned as a `nRatings`-by-`nRatings` transition matrix.

`sampleTotals` — Structure with sample totals

structure

Structure with sample totals, returned with fields:

- `totalsVec` — A vector of size `1`-by-`nRatings`.
- `totalsMat` — A matrix of size `nRatings`-by-`nRatings`.
- `algorithm` — A character vector with values `'duration'` or `'cohort'`.

For the `'duration'` algorithm, `totalsMat`(*i*,*j*) contains the total transitions observed out of rating *i* into rating*j* (all the diagonal elements are zero). The total time spent on rating *i* is stored in `totalsVec`(*i*). For example, if there are three rating categories, Investment Grade (`IG`), Speculative Grade (`SG`), and Default (`D`), and the following information:

```
Total time spent    IG       SG       D
in rating:       4859.09  1503.36  1162.05

Transitions             IG   SG    D
out of (row)        IG    0   89    7
into (column):      SG  202    0   32
                     D    0    0    0
```

Then

```
totals.totalsVec = [4859.09  1503.36  1162.05]
totals.totalsMat = [  0   89    7
                    202    0   32
                      0    0    0]
totals.algorithm = 'duration'
```

For the `'cohort'` algorithm, `totalsMat`(*i*,*j*) contains the total transitions observed from rating *i* to rating *j*, and `totalsVec`(*i*) is the initial count in rating *i*. For example, given the following information:

```
Initial count       IG     SG     D
in rating:        4808   1572   1145

Transitions         IG     SG     D
from (row)    IG  4721     80     7
to (column):  SG   193   1347    32
               D     0      0  1145
```

Then

```
totals.totalsVec = [4808   1572   1145]
totals.totalsMat = [4721     80      7
                    193   1347     32
                      0      0   1145
totals.algorithm = 'cohort'
```

`idTotals` — IDs totals

struct array

IDs totals, returned as a struct array of size `nIDs`-by-`1`, where *n*IDs is the number of distinct IDs in column 1 of `data` when this is a table or cell array or, equivalently, equal to the length of the `idStart` field minus 1 when `data` is a preprocessed data structure from `transprobprep`. For each ID in the sample, `idTotals` contains one structure with the following fields:

- `totalsVec` — A sparse vector of size `1`-by-`nRatings`.
- `totalsMat` — A sparse matrix of size `nRatings`-by-`nRatings`.
- `algorithm` — A character vector with values `'duration'` or `'cohort'`.

These fields contain the same information described for the output `sampleTotals`, but at an ID level. For example, for `'duration'`, `idTotals`(*k*).`totalsVec` contains the total time that the *k*-th company spent on each rating.

Definitions

Cohort Estimation

The cohort algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time.

If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates.

Duration Estimation

Unlike the cohort method, the duration algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur.

There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time.

Algorithms

Cohort Estimation

The algorithm first determines a sequence $t_0,...,t_K$ of snapshot dates. The elapsed time, in years, between two consecutive snapshot dates t_{k-1} and t_k is equal to $1 / ns$, where ns is the number of snapshots per year. These $K+1$ dates determine K transition periods.

The algorithm computes N_i^n, the number of transition periods in which obligor n starts at rating i. These are added up over all obligors to get N_i, the number of obligors in the sample that start a period at rating i. The number periods in which obligor n starts at rating i and ends at rating j, or migrates from i to j, denoted by N_{ij}^n, is also computed.

These are also added up to get N_{ij}, the total number of migrations from i to j in the sample.

The estimate of the transition probability from i to j in one period, denoted by P_{ij}, is given by:

$$P_{ij} = \frac{Nij}{Ni}$$

These probabilities are arranged in a one-period transition matrix P_0, where the i,j entry in P_0 is P_{ij}.

If the number of snapshots per year ns is 4 (quarterly snapshots), the probabilities in P_0 are 3-month (or 0.25-year) transition probabilities. You may, however, be interested in 1-year or 2-year transition probabilities. The latter time interval is called the transition interval, Δt , and it is used to convert P_0 into the final transition matrix, P, according to the formula:

$$P = P_0^{ns \triangle t}$$

For example, if ns = `4` and Δt = `2`, P contains the 2-year transition probabilities estimated from quarterly snapshots.

Note For the cohort algorithm, optional output arguments `idTotals` and `sampleTotals` from `transprob` contain the following information:

- `idTotals(n).totalsVec` = $(N_i^n) \forall i$
- `idTotals(n).totalsMat` = $(N_{i,j}^n) \forall ij$
- `idTotals(n).algorithm` = `'cohort'`
- `sampleTotals.totalsVec` = $(N_i) \forall i$
- `sampleTotals.totalsMat` = $(N_{i,j}) \forall ij$
- `sampleTotals.algorithm` = `'cohort'`

For efficiency, the vectors and matrices in `idTotals` are stored as sparse arrays.

Duration Estimation

The algorithm computes T_i^n , the total time that obligor n spends in rating i within the estimation time window. These quantities are added up over all obligors to get T_i , the total time spent in rating i, collectively, by all obligors in the sample. The algorithm also computes T_{ij}^n , the number times that obligor n migrates from rating i to rating j, with i

not equal to *j*, within the estimation time window. And it also adds them up to get T_{ij}, the total number of migrations, by all obligors in the sample, from the rating *i* to *j*, with *i* not equal to *j*.

To estimate the transition probabilities, the duration algorithm first computes a generator matrix Λ. Each off-diagonal entry of this matrix is an estimate of the transition rate out of rating *i* into rating *j*, and is given by:

$$\lambda_{ij} = \frac{T_{ij}}{T_i}, i \neq j$$

The diagonal entries are computed as:

$$\lambda_{ii} = -\sum_{j \neq i} \lambda_{ij}$$

With the generator matrix and the transition interval Δt (e.g., Δt = `2` corresponds to 2-year transition probabilities), the transition matrix is obtained as $P = \exp(\Delta t \Lambda)$, where *exp* denotes matrix exponentiation (`expm` in MATLAB).

Note For the duration algorithm, optional output arguments `idTotals` and `sampleTotals` from `transprob` contain the following information:

- `idTotals(n).totalsVec` = $(T_i^n) \forall i$
- `idTotals(n).totalsMat` = $(T_{i,j}^n) \forall ij$
- `idTotals(n).algorithm` = `'duration'`
- `sampleTotals.totalsVec` = $(T_i) \forall i$
- `sampleTotals.totalsMat` = $(T_{i,j}) \forall ij$
- `sampleTotals.algorithm` = `'duration'`

For efficiency, the vectors and matrices in `idTotals` are stored as sparse arrays.

References

[1] Hanson, S., T. Schuermann. "Confidence Intervals for Probabilities of Default." *Journal of Banking & Finance.* Vol. 30(8), Elsevier, August 2006, pp. 2281–2301.

[2] Löffler, G., P. N. Posch. *Credit Risk Modeling Using Excel and VBA.* West Sussex, England: Wiley Finance, 2007.

[3] Schuermann, T. "Credit Migration Matrices." in E. Melnick, B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment.* Wiley, 2008.

See Also

`table` | `transprobbytotals` | `transprobprep`

Topics

“Estimation of Transition Probabilities” on page 8-2
“Estimate Transition Probabilities for Different Rating Scales” on page 8-5

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Introduced in R2010b

transprobbytotals

Estimate transition probabilities using `totals` structure input

Syntax

```
[transMat,sampleTotals] = transprobbytotals(totals)
[transMat,sampleTotals] = transprobbytotals(___,Name, Value)
```

Description

`[transMat,sampleTotals] = transprobbytotals(totals)` estimates transition probabilities using a `totals` structure input. `transprobbytotals` is useful for removing outlier information, obtaining bootstrapped confidence intervals, or computing transition probability estimates for different periodicity parameters (1-year transitions, 2-year transitions, and so on) efficiently.

`[transMat,sampleTotals] = transprobbytotals(___,Name, Value)` adds optional name-value pair arguments.

Examples

Estimate Transition Probabilities Using a totals Structure Input

Use historical credit rating input data from `Data_TransProb.mat` and `transprob` to generate input for `transprobbytotals`:

```
load Data_TransProb

% Call TRANSPROB with three output arguments
[transMat, sampleTotals, idTotals] = transprob(data);
transMat
```

```
transMat =
```

```
  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Suppose companies 4 and 27 are outliers and you want to remove them from the pre-processed `idTotals` struct array and estimate the new transition probabilities.

```
idTotals([4 27]) = [];
[transMat1, sampleTotals1] = transprobbytotals(idTotals);
transMat1
```

```
transMat1 =

  Columns 1 through 7

   93.1172    5.8427    0.8231    0.1763    0.0377    0.0012    0.0001
    1.6213   93.1501    4.3584    0.6614    0.1631    0.0055    0.0004
    0.1239    2.9027   92.2297    4.0628    0.5367    0.0661    0.0028
    0.0236    0.2313    5.0070   90.1825    3.7986    0.4734    0.0642
    0.0216    0.1134    0.6357    5.7959   88.9866    3.4497    0.2920
    0.0010    0.0062    0.1081    0.8697    7.3367   86.7217    2.5171
    0.0002    0.0011    0.0120    0.2591    1.4340    4.3034   81.3027
         0         0         0         0         0         0         0

  Column 8
```

```
    0.0017
    0.0397
    0.0753
    0.2193
    0.7050
    2.4395
   12.6875
  100.0000
```

Obtain the 1-year, 2-year, 3-year, 4-year, and 5-year default probabilities, without the outlier information (i.e., using `sampleTotals1`).

```
DefProb = zeros(7,5);
for t = 1:5
    transMatTemp = transprobbytotals(sampleTotals1,'transInterval',t);
    DefProb(:,t) = transMatTemp(1:7,8);
end
DefProb
```

```
DefProb =

    0.0017    0.0070    0.0159    0.0285    0.0450
    0.0397    0.0828    0.1299    0.1813    0.2377
    0.0753    0.1606    0.2567    0.3640    0.4831
    0.2193    0.4675    0.7430    1.0445    1.3700
    0.7050    1.4668    2.2759    3.1232    4.0000
    2.4395    4.9282    7.4071    9.8351   12.1847
   12.6875   23.1184   31.7177   38.8282   44.7266
```

- “Estimation of Transition Probabilities” on page 8-2
- “Estimate Transition Probabilities for Different Rating Scales” on page 8-5

Input Arguments

`totals` — Total transitions observed

structure | struct array

Total transitions observed, specified as a structure, or a struct array of length nTotals, with fields:

- `totalsVec` — A sparse vector of size `1`-by-`nRatings1`.
- `totalsMat` — A sparse matrix of size `nRatings1`-by-`nRatings2` with `nRatings1` ≤ `nRatings2`.
- `algorithm` — A character vector with values `'duration'` or `'cohort'`.

For the `'duration'` algorithm, `totalsMat`(*i*,*j*) contains the total transitions observed out of rating *i* into rating *j* (all the diagonal elements are 0). The total time spent on rating *i* is stored in `totalsVec`(*i*). For example, you have three rating categories, Investment Grade (`IG`), Speculative Grade (`SG`), and Default (`D`), and the following information:

```
Total time spent    IG       SG       D
in rating:       4859.09  1503.36  1162.05

Transitions             IG   SG    D
out of (row)        IG    0   89    7
into (column):      SG  202    0   32
                     D    0    0    0
```

Then:

```
totals.totalsVec = [4859.09  1503.36  1162.05]
totals.totalsMat = [  0   89    7
                    202    0   32
                      0    0    0]
totals.algorithm = 'duration'
```

For the `'cohort'` algorithm, `totalsMat`(*i*,*j*) contains the total transitions observed from rating *i* to rating *j*, and `totalsVec`(*i*) is the initial count in rating *i*. For example, given the following information:

```
Initial count       IG     SG     D
in rating:        4808   1572   1145

Transitions         IG     SG     D
from (row)    IG  4721     80     7
to (column):  SG   193   1347    32
               D     0      0   1145
```

Then:

```
totals.totalsVec = [4808   1572   1145]
totals.totalsMat = [4721     80      7
```

```
                    193   1347     32
                      0      0   1145
totals.algorithm = 'cohort'
```

Common totals structures are the optional output arguments from `transprob`:

- `sampleTotals` — A single structure summarizing the totals information for the whole dataset.
- `idTotals` — A struct array with the totals information at the ID level.

Data Types: `struct | structure`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `transMat = transprobbytotals(Totals1,'transInterval',5)`

`snapsPerYear` — Number of credit-rating snapshots per year

`1` (default) | numeric values are `1`, `2`, `3`, `4`, `6`, or `12`

Number of credit-rating snapshots per year to be considered for the estimation, specified as a numeric value of `1`, `2`, `3`, `4`, `6`, or `12`.

Note This parameter is only used with the `'cohort'` `algorithm`.

Data Types: `double`

`transInterval` — Length of the transition interval in years

`1` (one year transition probability) (default) | numeric

Length of the transition interval, in years, specified as a numeric value.

Data Types: `double`

Output Arguments

`transMat` — Matrix of transition probabilities in percent
matrix

Matrix of transition probabilities in percent, returned as a `nRatings1`-by-`nRatings2` transition matrix.

`sampleTotals` — Structure with sample totals
structure

Structure with sample totals, returned with fields:

- `totalsVec` — A vector of size `1`-by-`nRatings1`.
- `totalsMat` — A matrix of size `nRatings1`-by-`nRatings2` with `nRatings1` ≤ `nRatings2`.
- `algorithm` — A character vector with values `'duration'` or `'cohort'`.

If `totals` is a struct array, `sampleTotals` contains the aggregated information. That is, `sampleTotals.totalsVec` is the sum of `totals`(*k*).`totalsVec` over all *k*, and similarly for `totalsMat`. When `totals` is itself a single structure, `sampleTotals` and `totals` are the same.

Definitions

Cohort Estimation

The cohort algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time.

If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates.

Duration Estimation

Unlike the cohort method, the duration algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur.

There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time.

References

[1] Hanson, S., T. Schuermann. "Confidence Intervals for Probabilities of Default." *Journal of Banking & Finance*. Vol. 30(8), Elsevier, August 2006, pp. 2281–2301.

[2] Löffler, G., P. N. Posch. *Credit Risk Modeling Using Excel and VBA*. West Sussex, England: Wiley Finance, 2007.

[3] Schuermann, T. "Credit Migration Matrices." in E. Melnick, B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment*. Wiley, 2008.

See Also

`transprob` | `transprobgrouptotals`

Topics

“Estimation of Transition Probabilities” on page 8-2
“Estimate Transition Probabilities for Different Rating Scales” on page 8-5

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Introduced in R2010b

transprobfromthresholds

Convert from credit quality thresholds to transition probabilities

Syntax

```
trans = transprobfromthresholds(thresh)
```

Description

`trans = transprobfromthresholds(thresh)` transforms credit quality thresholds into transition probabilities

Examples

Transform Credit Quality Thresholds Into Transition Probabilities

Use historical credit rating input data from `Data_TransProb.mat`, estimate transition probabilities with default settings.

```
load Data_TransProb

% Estimate transition probabilities with default settings
transMat = transprob(data)
```

```
transMat =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
```

```
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Obtain the credit quality thresholds.

```
thresh = transprobtothresholds(transMat)

thresh =

  Columns 1 through 7

       Inf   -1.4846   -2.3115   -2.8523   -3.3480   -4.0083   -4.1276
       Inf    2.1403   -1.6228   -2.3788   -2.8655   -3.3166   -3.3523
       Inf    3.0264    1.8773   -1.6690   -2.4673   -2.9800   -3.1631
       Inf    3.4963    2.8009    1.6201   -1.6897   -2.4291   -2.7663
       Inf    3.5195    2.9999    2.4225    1.5089   -1.7010   -2.3275
       Inf    4.2696    3.8015    3.0477    2.3320    1.3838   -1.6491
       Inf    4.6241    4.2097    3.6472    2.7803    2.1199    1.5556
       Inf       Inf       Inf       Inf       Inf       Inf       Inf

  Column 8

   -4.1413
   -3.3554
   -3.1736
   -2.8490
   -2.4547
   -1.9703
   -1.1399
       Inf
```

Recover the transition probabilities.

```
trans = transprobfromthresholds(thresh)

trans =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

- “Estimation of Transition Probabilities” on page 8-2
- “Estimate Transition Probabilities for Different Rating Scales” on page 8-5
- “Estimate Probabilities for Different Segments” on page 8-16

Input Arguments

`thresh` — Credit quality thresholds

matrix

Credit quality thresholds, specified as a `M`-by-`N` matrix of credit quality thresholds.

In each row, the first element must be `Inf` and the entries must satisfy the following monotonicity condition:

```
thresh(i,j) >= thresh(i,j+1), for 1<=j<N
```

The `M`-by-`N` input `thresh` and the `M`-by-`N` output `trans` are related as follows. The thresholds `thresh`(*i*,*j*) are critical values of a standard normal distribution *z*, such that:

```
trans(i,N) = P[z < thresh(i,N)],

trans(i,j) = P[z < thresh(i,j)] - P[z < thresh(i,j+1)], for 1<=j<N
```

Any given row in the output matrix `trans` determines a probability distribution over a discrete set of `N` ratings `'R1'`, . . . , `'RN'`, so that for any row *i* `trans`(*i*,*j*) is the probability of migrating into `'R`*j*`'`. `trans` can be a standard transition matrix, with `M` ≤ `N`, in which case row *i* contains the transition probabilities for issuers with rating `'R`*i*`'`. But `trans` does not have to be a standard transition matrix. `trans` can contain individual transition probabilities for a set of `M`-specific issuers, with `M` > `N`.

For example, suppose that there are only `N`=3 ratings, `'High'`, `'Low'`, and `'Default'`, with these credit quality thresholds:

```
        High     Low     Default
High     Inf   -2.0814   -3.1214
Low      Inf    2.4044   -1.7530
```

The matrix of transition probabilities is then:

```
       High    Low    Default
High  98.13    1.78    0.09
Low    0.81   95.21    3.98
```

This means the probability of default for `'High'` is equivalent to drawing a standard normal random number smaller than −3.1214, or 0.09%. The probability that a `'High'` ends up the period with a rating of `'Low'` or lower is equivalent to drawing a standard normal random number smaller than −2.0814, or 1.87%. From here, the probability of ending with a `'Low'` rating is:

```
P[z<-2.0814] - P[z<-3.1214] = 1.87% - 0.09% = 1.78%
```

And the probability of ending with a `'High'` rating is:

```
100%-1.87% = 98.13%
```

where 100% is the same as:

```
P[z<Inf]
```

Data Types: `double`

Output Arguments

`trans` — Matrix of transition probabilities in percent

matrix

Matrix of transition probabilities in percent, returned as a `M`-by-`N` matrix.

References

[1] Gupton, G. M., C. C. Finger, and M. Bhatia. *"CreditMetrics."* Technical Document, RiskMetrics Group, Inc., 2007.

See Also

`transprob` | `transprobbytotals` | `transprobtothresholds`

Topics

"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-5
"Estimate Probabilities for Different Segments" on page 8-16

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Introduced in R2011b

transprobgrouptotals

Aggregate credit ratings information into fewer rating categories

Syntax

```
totalsGrouped = transprobgrouptotals(totals,groupingEdges)
```

Description

`totalsGrouped = transprobgrouptotals(totals,groupingEdges)` aggregates the credit ratings information stored in the `totals` input into fewer ratings categories, which are defined by the `groupingEdges` argument.

Examples

Aggregate the Credit Ratings Information Stored in the totals Input

Use historical credit rating input data from `Data_TransProb.mat`. Load input data from file `Data_TransProb.mat`.

```
load Data_TransProb

% Call TRANSPROB with two output arguments
[transMat, sampleTotals] = transprob(data);
transMat
```

```
transMat =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
```

```
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Group into investment grade (ratings 1-4) and speculative grade (ratings 5-7); note, the default is the last rating (number 8).

```
edges = [4 7 8];
sampleTotalsGrp = transprobgrouptotals(sampleTotals,edges);

% Transition matrix at investment grade / speculative grade level
transMatIGSG = transprobbytotals(sampleTotalsGrp)

transMatIGSG =

   98.5336    1.3608    0.1056
    3.9155   92.9692    3.1153
         0         0  100.0000
```

Obtain the 1-year, 2-year, 3-year, 4-year, and 5-year default probabilities at investment grade and speculative grade level.

```
DefProb = zeros(2,5);
for t = 1:5
transMatTemp = transprobbytotals(sampleTotalsGrp,'transInterval',t);
DefProb(:,t) = transMatTemp(1:2,3);
end
DefProb

DefProb =
```

```
    0.1056    0.2521    0.4359    0.6537    0.9027
    3.1153    6.0157    8.7179   11.2373   13.5881
```

- "Group Credit Ratings" on page 8-11
- "Estimation of Transition Probabilities" on page 8-2
- "Estimate Transition Probabilities for Different Rating Scales" on page 8-5
- "Estimate Probabilities for Different Segments" on page 8-16

Input Arguments

`totals` — Total transitions observed

structure | struct array

Total transitions observed, specified as a structure, or a struct array of length nTotals, with fields:

- `totalsVec` — A sparse vector of size `1`-by-`nRatings1`.
- `totalsMat` — A sparse matrix of size `nRatings1`-by-`nRatings2` with `nRatings1` ≤ `nRatings2`.
- `algorithm` — A character vector with values `'duration'` or `'cohort'`.

For the `'duration'` algorithm, `totalsMat`(*i*,*j*) contains the total transitions observed out of rating *i* into rating *j* (all the diagonal elements are 0). The total time spent on rating *i* is stored in `totalsVec`(*i*). For example, you have three rating categories, Investment Grade (`IG`), Speculative Grade (`SG`), and Default (`D`), and the following information:

```
Total time spent    IG       SG       D
in rating:       4859.09  1503.36  1162.05

Transitions             IG   SG    D
out of (row)        IG    0   89    7
into (column):      SG  202    0   32
                     D    0    0    0
```

Then:

```
totals.totalsVec = [4859.09  1503.36  1162.05]
totals.totalsMat = [  0   89    7
```

```
                    202    0   32
                      0    0    0]
totals.algorithm = 'duration'
```

For the `'cohort'` algorithm, `totalsMat`(*i*,*j*) contains the total transitions observed from rating *i* to rating *j*, and `totalsVec`(*i*) is the initial count in rating *i*. For example, given the following information:

```
Initial count       IG     SG     D
in rating:        4808   1572   1145

Transitions         IG     SG     D
from (row)    IG  4721     80     7
to (column):  SG   193   1347    32
               D     0      0   1145
```

Then:

```
totals.totalsVec = [4808   1572   1145]
totals.totalsMat = [4721     80      7
                    193   1347     32
                      0      0   1145
totals.algorithm = 'cohort'
```

Common totals structures are the optional output arguments from `transprob`:

- `sampleTotals` — A single structure summarizing the totals information for the whole dataset.
- `idTotals` — A struct array with the totals information at the ID level.

Data Types: `struct` | `structure`

`groupingEdges` — Indicator for grouping credit ratings into categories

numeric array

Indicator for grouping credit ratings into categories, specified as a numeric array.

This table illustrates how to group a list of whole ratings into investment grade (`IG`) and speculative grade (`SG`) categories. Eight ratings are in the original list. Ratings `1` to `4` are `IG`, ratings `5` to `7` are `SG`, and rating `8` is a category of its own. In this example, the array of grouping edges is `[4 7 8]`.

```
Original ratings: 'AAA' 'AA'  'A'   'BBB' | 'BB'  'B'   'CCC' | 'D'
                                         |                   |
```

```
Relative ordering: (1)   (2)   (3)   (4)  |  (5)  (6)    (7)  | (8)
                                          |                   |
Grouped ratings:           'IG'           |      'SG'         | 'D'
                                          |                   |
Grouping edges:                      (4)  |              (7)  | (8)
```

In general, if `groupingEdges` has *K* elements `edge1` < `edge2` < . . . <`edge`*K*, ratings `1` to `edge1` (inclusive) are grouped in the first category, ratings `edge1`+`1` to `edge2` in the second category, and so forth.

Regarding the last element, `edge`*K*:

- If *n*`Ratings1` equals *n*`Ratings2`, then `edge`*K* must equal *n*`Ratings1`. This leads to *K* groups, and *n*`RatingsGrouped1` = *n*`RatingsGrouped2` = *K*.
- If *n*`Ratings1` < *n*`Ratings2`, then either:
  - `edge`*K* equals *n*`Ratings1`, in which case ratings `edge`*K*+`1`,...,`nRatings2` are treated as categories of their own. This results in *K*+(`nRatings2`-`edge`*K*) groups, with `nRatingsGrouped1` = *K* and `nRatingsGrouped2` = *K* + (`nRatings2` – `edge`*K*); or
  - `edge`*K* equals `nRatings2`, in which case there must be a *j*th edge element, `edge`*j*, such that `edge`*j* equals `nRatings1`. This leads to *K* groups, and `nRatingsGrouped1` = *j* and `nRatingsGrouped2` = *K*.

Data Types: `double`

Output Arguments

`totalsGrouped` — Aggregated information by categories

structure | struct array

Aggregated information by categories, returned as a structure, or a struct array of length `nTotals`, with fields:

- `totalsVec` — A vector of size `1`-by-`nRatingsGrouped1`.
- `totalsMat` — A matrix of size `nRatingsGrouped1`-by-`nRatingsGrouped2`.
- `algorithm` — A character vector, `'duration'` or `'cohort'`.

`nRatingsGrouped1` and `nRatingsGrouped2` are defined in the description of `groupingEdges`. Each structure contains aggregated information by categories, based

on the information provided in the corresponding structure in `totals`, according to the grouping of ratings defined by `groupingEdges` and consistent with the `algorithm` choice.

Following the examples in the description of the `totals` input, suppose `IG` and `SG` are grouped into a single `ND` (Not-Defaulted) category, using the edges `[2 3]`. For the `'cohort'` algorithm, the output is:

```
totalsGrouped.totalsVec = [6380   1145]
totalsGrouped.totalsMat = [6341     39
                              0   1145]
totalsGrouped.algorithm = 'cohort'
```

and for the `'duration'` algorithm:

```
totalsGrouped.totalsVec = [6362.45  1162.05]
totalsGrouped.totalsMat = [0  39
                           0   0]
totalsGrouped.algorithm = 'duration'
```

Definitions

Cohort Estimation

The `cohort` algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time.

If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates. For more information, see the Algorithms section of `transprob`.

Duration Estimation

Unlike the `cohort` algorithm, the `duration` algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur.

There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time. For more information, see the Algorithms section of `transprob`.

References

[1] Hanson, S., T. Schuermann. "Confidence Intervals for Probabilities of Default." *Journal of Banking & Finance*. Vol. 30(8), Elsevier, August 2006, pp. 2281–2301.

[2] Löffler, G., P. N. Posch. *Credit Risk Modeling Using Excel and VBA*. West Sussex, England: Wiley Finance, 2007.

[3] Schuermann, T. "Credit Migration Matrices." in E. Melnick, B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment*. Wiley, 2008.

See Also

`transprob` | `transprobbytotals`

Topics

“Group Credit Ratings” on page 8-11
“Estimation of Transition Probabilities” on page 8-2
“Estimate Transition Probabilities for Different Rating Scales” on page 8-5
“Estimate Probabilities for Different Segments” on page 8-16

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Introduced in R2011b

transprobprep

Preprocess credit ratings data to estimate transition probabilities

Syntax

```
[prepData] = transprobprep(data)
[prepData] = transprobprep(___,Name, Value)
```

Description

`[prepData] = transprobprep(data)` preprocesses credit ratings historical data (that is, credit migration data) for the subsequent estimation of transition probabilities.

`[prepData] = transprobprep(___,Name, Value)` adds optional name-value pair arguments.

Examples

Aggregate the Credit Ratings Information Stored in the totals Input

Load input data from the file `Data_TransProb.mat` and display the first ten rows. In this example, the inputs are provided in character vector format.

```
load Data_TransProb

% Preprocess credit ratings data.
prepData = transprobprep(data)
```

```
prepData = struct with fields:
           idStart: [1506x1 double]
      numericDates: [4315x1 double]
    numericRatings: [4315x1 double]
     ratingsLabels: {'AAA'  'AA'  'A'  'BBB'  'BB'  'B'  'CCC'  'D'}
```

Estimate transition probabilities with the default settings.

```
transMat = transprob(prepData)
```

```
transMat =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Estimate transition probabilities with the `'cohort'` algorithm.

```
transMatCoh = transprob(prepData,'algorithm','cohort')
```

```
transMatCoh =

  Columns 1 through 7

   93.1345    5.9335    0.7456    0.1553    0.0311         0         0
    1.7359   92.9198    4.5446    0.6046    0.1560         0         0
    0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0
    0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421
    0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873
         0         0    0.0761    0.7230    7.9909   86.1872    2.7397
         0         0         0    0.3094    1.8561    4.5630   80.8971
         0         0         0         0         0         0         0
```

```
Column 8

       0
  0.0390
  0.0725
  0.2103
  0.6409
  2.2831
 12.3743
100.0000
```

- “Group Credit Ratings” on page 8-11
- “Estimation of Transition Probabilities” on page 8-2
- “Estimate Transition Probabilities for Different Rating Scales” on page 8-5
- “Estimate Probabilities for Different Segments” on page 8-16

Input Arguments

`data` — Historical data for credit ratings

table | cell array of character vectors

Historical input data for credit ratings, specified as one of the following:

- A MATLAB table of size `nRecords`-by-`3` containing the credit ratings. Each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). The assigned credit rating corresponds to the associated ID on the associated date. All information corresponding to the same ID must be stored in contiguous rows. Sorting this information by date is not required, but recommended for efficiency. When using a MATLAB table input, the names of the columns are irrelevant, but the ID, date and rating information are assumed to be in the first, second, and third columns, respectively. Also, when using a table input, the first and third columns can be categorical arrays, and the second can be a datetime array. Here is an example with all the information in table format:

```
 ID            Date             Rating
_________    _____________    ______
'00010283'    '10-Nov-1984'    'CCC'
'00010283'    '12-May-1986'    'B'
```

```
'00010283'    '29-Jun-1988'    'CCC'
'00010283'    '12-Dec-1991'    'D'
'00013326'    '09-Feb-1985'    'A'
'00013326'    '24-Feb-1994'    'AA'
```

The following summarizes the supported data types for table input:

Data Input Type	ID (1st Column)	Date (2nd Column)	Rating (3rd Column)
Table	• Numeric array • Cell array of character vectors • Categorical array	• Numeric array • Cell array of character vectors • Datetime array	• Numeric array • Cell array of character vectors • Categorical array

- A cell array of size `nRecords`-by-`3` containing the credit ratings. Each row contains an ID (column 1), a date (column 2), and a credit rating (column 3). The assigned credit rating corresponds to the associated ID on the associated date. All information corresponding to the same ID must be stored in contiguous rows. Sorting this information by date is not required but is recommended. IDs, dates, and ratings are stored in character vector format, but they can also be entered in numeric format. Here is an example with all the information in character vector format:

  ```
  '00010283'    '10-Nov-1984'    'CCC'
  '00010283'    '12-May-1986'    'B'
  '00010283'    '29-Jun-1988'    'CCC'
  '00010283'    '12-Dec-1991'    'D'
  '00013326'    '09-Feb-1985'    'A'
  '00013326'    '24-Feb-1994'    'AA'
  ```

  The following summarizes the supported data types for cell array input:

Data Input Type	ID (1st Column)	Date (2nd Column)	Rating (3rd Column)
Cell	• Numeric elements • Character vector elements	• Numeric elements • Character vector elements	• Numeric elements • Character vector elements

Data Types: `table` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `prepData = transprobprep(data,'labels', {'AAA','AA','A','BBB','BB','B','CCC','F'})`

`labels` — Credit-rating scale

`{'AAA','AA','A','BBB','BB','B','CCC','D'}` (default) | cell array of character vectors

Credit-rating scale, specified as a `nRatings`-by-`1`, or `1`-by-`nRatings` cell array of character vectors.

`labels` must be consistent with the ratings labels used in the third column of `data`. Use a cell array of numbers for numeric ratings, and a cell array for character vectors for categorical ratings.

Data Types: `cell`

Output Arguments

`prepData` — Summary where credit ratings information corresponding to each company starts and ends

structure

Summary where the credit ratings information corresponding to each company starts and ends, returned as a structure with the following fields:

- `idStart` — Array of size (*n*IDs+1)-by-1, where *n*IDs is the number of distinct IDs in column 1 of `data`. This array summarizes where the credit ratings information corresponding to each company starts and ends. The dates and ratings corresponding to company *j* in `data` are stored from row `idStart`(*j*) to row `idStart`(*j*+1)−1 of `numericDates` and `numericRatings`.
- `numericDates` — Array of size *n*Records-by-`1`, containing the dates in column 2 of `data`, in numeric format.

- `numericRatings` — Array of size *n*Records-by-`1`, containing the ratings in column 3 of `data`, mapped into numeric format.
- `ratingsLabels` — Cell array of size`1`-by-`nRatings`, containing the credit rating scale.

See Also

`table` | `transprob` | `transprobbytotals`

Topics

"Group Credit Ratings" on page 8-11
"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-5
"Estimate Probabilities for Different Segments" on page 8-16

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Introduced in R2011b

transprobtothresholds

Convert from transition probabilities to credit quality thresholds

Syntax

```
thresh = transprobtothresholds(trans)
```

Description

`thresh = transprobtothresholds(trans)` transforms transition probabilities into credit quality thresholds.

Examples

Transform Transition Probabilities Into Credit Quality Thresholds

Use historical credit rating input data from `Data_TransProb.mat`. Load input data from file `Data_TransProb.mat`.

```
load Data_TransProb

% Estimate transition probabilities with default settings
transMat = transprob(data)
```

```
transMat =

  Columns 1 through 7

   93.1170    5.8428    0.8232    0.1763    0.0376    0.0012    0.0001
    1.6166   93.1518    4.3632    0.6602    0.1626    0.0055    0.0004
    0.1237    2.9003   92.2197    4.0756    0.5365    0.0661    0.0028
    0.0236    0.2312    5.0059   90.1846    3.7979    0.4733    0.0642
    0.0216    0.1134    0.6357    5.7960   88.9866    3.4497    0.2919
    0.0010    0.0062    0.1081    0.8697    7.3366   86.7215    2.5169
    0.0002    0.0011    0.0120    0.2582    1.4294    4.2898   81.2927
```

```
         0         0         0         0         0         0         0

  Column 8

    0.0017
    0.0396
    0.0753
    0.2193
    0.7050
    2.4399
   12.7167
  100.0000
```

Obtain the credit quality thresholds.

```
thresh = transprobtothresholds(transMat)

thresh =

  Columns 1 through 7

      Inf   -1.4846   -2.3115   -2.8523   -3.3480   -4.0083   -4.1276
      Inf    2.1403   -1.6228   -2.3788   -2.8655   -3.3166   -3.3523
      Inf    3.0264    1.8773   -1.6690   -2.4673   -2.9800   -3.1631
      Inf    3.4963    2.8009    1.6201   -1.6897   -2.4291   -2.7663
      Inf    3.5195    2.9999    2.4225    1.5089   -1.7010   -2.3275
      Inf    4.2696    3.8015    3.0477    2.3320    1.3838   -1.6491
      Inf    4.6241    4.2097    3.6472    2.7803    2.1199    1.5556
      Inf       Inf       Inf       Inf       Inf       Inf       Inf

  Column 8

   -4.1413
   -3.3554
   -3.1736
   -2.8490
   -2.4547
   -1.9703
   -1.1399
       Inf
```

- “Credit Quality Thresholds” on page 8-52

- “Group Credit Ratings” on page 8-11
- “Estimation of Transition Probabilities” on page 8-2
- “Estimate Transition Probabilities for Different Rating Scales” on page 8-5
- “Estimate Probabilities for Different Segments” on page 8-16

Input Arguments

`trans` — Transition probabilities in percent

matrix

Transition probabilities in percent, specified as a `M`-by-`N` matrix. Entries cannot be negative and cannot exceed 100, and all rows must add up to 100.

Any given row in the `M`-by-`N` input matrix `trans` determines a probability distribution over a discrete set of `N` ratings. If the ratings are `'R1',...,'RN'`, then for any row i `trans(i,j)` is the probability of migrating into `'Rj'`. If `trans` is a standard transition matrix, then $M \leqq N$ and row i contains the transition probabilities for issuers with rating `'Ri'`. But `trans` does not have to be a standard transition matrix. `trans` can contain individual transition probabilities for a set of `M`-specific issuers, with $M > N$.

The credit quality thresholds `thresh`(i,j) are critical values of a standard normal distribution z, such that:

```
trans(i,N) = P[z < thresh(i,N)],

trans(i,j) = P[z < thresh(i,j)] - P[z < thresh(i,j+1)], for 1<=j<N
```

This implies that `thresh`(i,1) = `Inf`, for all i. For example, suppose that there are only `N`=3 ratings, `'High'`, `'Low'`, and `'Default'`, with the following transition probabilities:

```
      High   Low   Default
High  98.13   1.78   0.09
Low    0.81  95.21   3.98
```

The matrix of credit quality thresholds is:

```
        High     Low     Default
High     Inf   -2.0814   -3.1214
Low      Inf    2.4044   -1.7530
```

This means the probability of default for `'High'` is equivalent to drawing a standard normal random number smaller than −3.1214, or 0.09%. The probability that a `'High'`

ends up the period with a rating of `'Low'` or lower is equivalent to drawing a standard normal random number smaller than −2.0814, or 1.87%. From here, the probability of ending with a `'Low'` rating is:

```
P[z<-2.0814] - P[z<-3.1214] = 1.87% - 0.09% = 1.78%
```

And the probability of ending with a `'High'` rating is:

```
100%-1.87% = 98.13%
```

where 100% is the same as:

```
P[z<Inf]
```

Data Types: `double`

Output Arguments

`thresh` — Credit quality thresholds
matrix

Credit quality thresholds, returned as a `M`-by-`N` matrix.

References

[1] Gupton, G. M., C. C. Finger, and M. Bhatia. *"CreditMetrics."* Technical Document, RiskMetrics Group, Inc., 2007.

See Also

`transprob` | `transprobbytotals` | `transprobfromthresholds`

Topics

"Credit Quality Thresholds" on page 8-52
"Group Credit Ratings" on page 8-11
"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-5
"Estimate Probabilities for Different Segments" on page 8-16

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

Introduced in R2011b

ts2func

Convert time series arrays to functions of time and state

Syntax

```
F = ts2func(Array)

F = ts2func(Array, 'Name1', Value1, 'Name2', Value2, ...)
```

Description

The `ts2func` function encapsulates a time series array associated with a vector of real-valued observation times within a MATLAB function suitable for Monte Carlo simulation of an `NVARS`-by-`1` state vector X_t.

Input Arguments

`Array`	Time series array to encapsulate within a callable function of time and state. `Array` may be a vector, 2-dimensional matrix, or three-dimensional array.

Optional Input Arguments

Specify optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character vector, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial character vector matches.

Valid parameter names are:

`Times`	Vector of monotonically increasing observation times associated with the time series input array `Array`. If you do not specify a value for this argument, `Times` is a zero-based, unit-increment vector of the same length as that of the dimension of `Array` associated with time (see `TimeDimension`).
`TimeDimension`	Positive scalar integer that specifies which dimension of the input time series array `Array` is associated with time. The value of this argument cannot be greater than the number of dimensions of `Array`. If you do not specify a value for this argument, the default is `1`, indicating that time is associated with the rows of `Array`.
`StateDimension`	Positive scalar integer that specifies which dimension of the input time series array `Array` is associated with the `NVARS` state variables. `StateDimension` cannot be greater than the number of dimensions of `Array`. If you do not specify a value for this argument, `ts2func` assigns `StateDimension` the first dimension of `Array` that is not already associated with time (the state vector X_t is associated with the first available dimension of `Array` *not* already assigned to `TimeDimension`).
`Deterministic`	A scalar, logical flag to indicate whether the output function is a deterministic function of time alone. If `Deterministic` is true, the output function `F` is a deterministic function of time, *F(t)*, and the only input it accepts is a scalar, real-valued time *t*. If `Deterministic` is false, the output function `F` accepts two inputs, a scalar, real-valued time *t* followed by an `NVARS`-by-`1` state vector*X(t)*. The default is false, and `F` is a callable function of time and state, *F(t,X)* .

Output Arguments

`F`	Callable function *F(t)* of a real-valued scalar observation time *t*. You can invoke `F` with a second input (such as an `NVARS`-by-`1` state vector *X*), which is a placeholder that `ts2func` ignores. For example, while *F(t)* and *F(t,X)* produce identical results, the latter directly supports SDE simulation methods. If the optional input argument `Deterministic` is true, `F` is a deterministic function of time, *F(t)*, and the only input it accepts is a scalar, real-valued time *t*. Otherwise, if `Deterministic` is false (the default), `F` accepts a scalar, real-valued time *t* followed by an `NVARS`-by-`1` state vector *X(t)*.

Algorithms

- When you specify `Array` as a trivial scalar or a vector (row or column), `ts2func` assumes that it represents a univariate time series.
- `F` returns an array with one fewer dimension than the input time series array `Array` with which `F` is associated. Thus, when `Array` is a vector, a 2-dimensional matrix, or a three-dimensional array, `F` returns a scalar, vector, or 2-dimensional matrix, respectively.
- When the scalar time *t* at which `ts2func` evaluates the function `F` does not coincide with an observation time in `Times`, `F` performs a zero-order-hold interpolation. The only exception is if *t* precedes the first element of `Times`, in which case *F(t)* = *F(Times(1))*.
- To support Monte Carlo simulation methods, the output function `F` returns an `NVARS`-by-`1` column vector or a 2-dimensional matrix with `NVARS` rows.
- The output function `F` is always a deterministic function of time, *F(t)*, and may always be called with a single input regardless of the `Deterministic` flag. The distinction is that when `Deterministic` is false, the function `F` may also be called with a second input, an `NVARS`-by-`1` state vector *X(t)*, which is a placeholder and ignored. While *F(t)* and *F(t,X)* produce identical results, the former specifically indicates that the function is a deterministic function of time, and may offer significant performance benefits in some situations.

References

Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*, Vol. 54, No. 4, August 1999.

Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York, Springer-Verlag, 2004.

Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `simulate`

Topics

"Simulating Equity Prices" on page 17-34
"Simulating Interest Rates" on page 17-59
"Stratified Sampling" on page 17-70
"Pricing American Basket Options by Monte Carlo Simulation" on page 17-84
"Base SDE Models" on page 17-16
"Drift and Diffusion Models" on page 17-19
"Linear Drift Models" on page 17-23
"Parametric Models" on page 17-25
"SDEs" on page 17-2
"SDE Models" on page 17-8
"SDE Class Hierarchy" on page 17-5
"Performance Considerations" on page 17-76

Introduced in R2008a

tsaccel

Acceleration between times

Syntax

```
acc = tsaccel(data,nTimes,datatype)

accts = tsaccel(tsobj,nTimes,datatype)
```

Arguments

`data`	Data series.
`nTimes`	(Optional) Number of times. 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,nTimes,datatype)` calculates the acceleration of a data series, essentially the difference of the current momentum with the momentum some number of periods ago. If `nTimes` is specified, `tsaccel` calculates the acceleration of a data series data with time distance of `nTimes` times.

`accts = tsaccel(tsobj,nTimes,datatype)` calculates the acceleration of the data series in the financial time series object `tsobj`, essentially the difference of the current momentum with the momentum some number of periods ago. Each data series in `tsobj`

is treated individually. `accts` is a financial time series object with similar dates and data series names as `tsobj`.

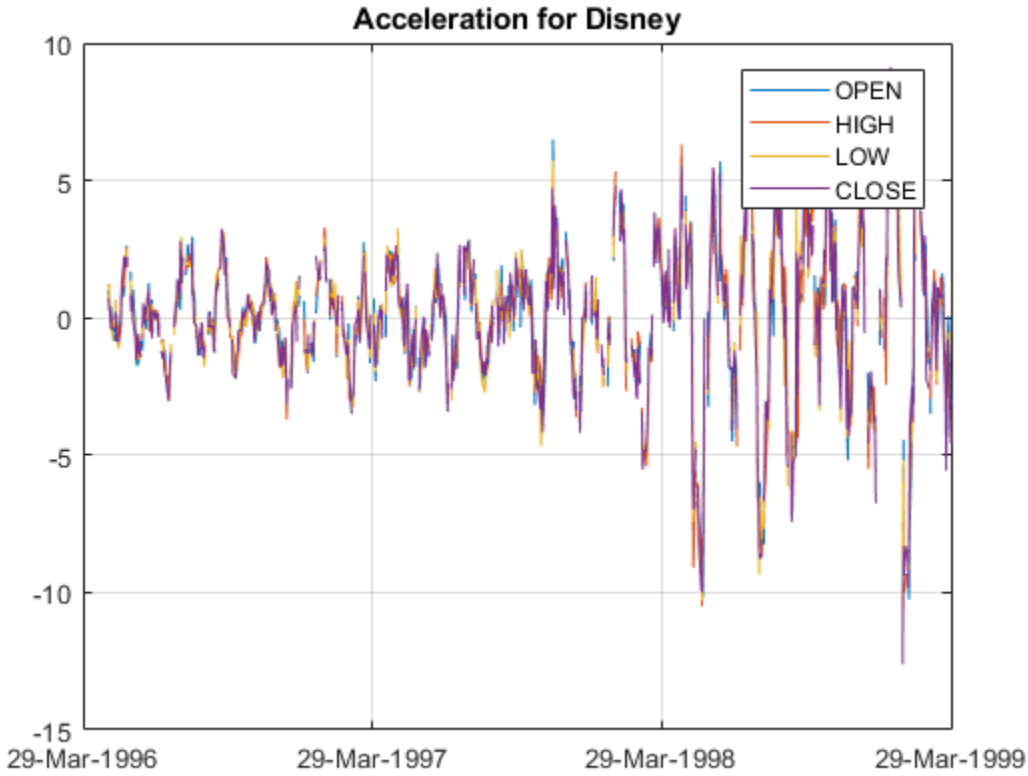
Note, to compute a quantity over `n` periods, you must specify `n+1` for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Calculate the Acceleration of a Data Series

This example shows how to calculate the acceleration of a data series for Disney stock and plot the results.

```
load disney.mat
dis = rmfield(dis,'VOLUME'); % remove VOLUME field
dis_Accel = tsaccel(dis);
plot(dis_Accel)
title('Acceleration for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Kaufman, P. J. *The New Commodity Trading Systems and Methods*. John Wiley & Sons, New York, 1987.

See Also

`tsmom`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

tsmom

Momentum between times

Syntax

```
mom = tsmom(data,nTimes)

momts = tsmom(tsobj,nTimes)
```

Arguments

`data`	Data series. Column-oriented vector or matrix.
`nTimes`	(Optional) Number of times. Default = `12`.
`tsobj`	Financial time series object.

Description

Momentum is the difference between two prices (data points) separated by a number of times.

`mom = tsmom(data,nTimes)` calculates the momentum of a data series data. If `nTimes` is specified, `tsmom` uses that value instead of the default 12.

`momts = tsmom(tsobj,nTimes)` calculates the momentum of all data series in the financial time series object `tsobj`. Each data series in `tsobj` is treated individually. `momts` is a financial time series object with similar dates and data series names as `tsobj`. If `nTimes` is specified, `tsmom` uses that value instead of the default `12`.

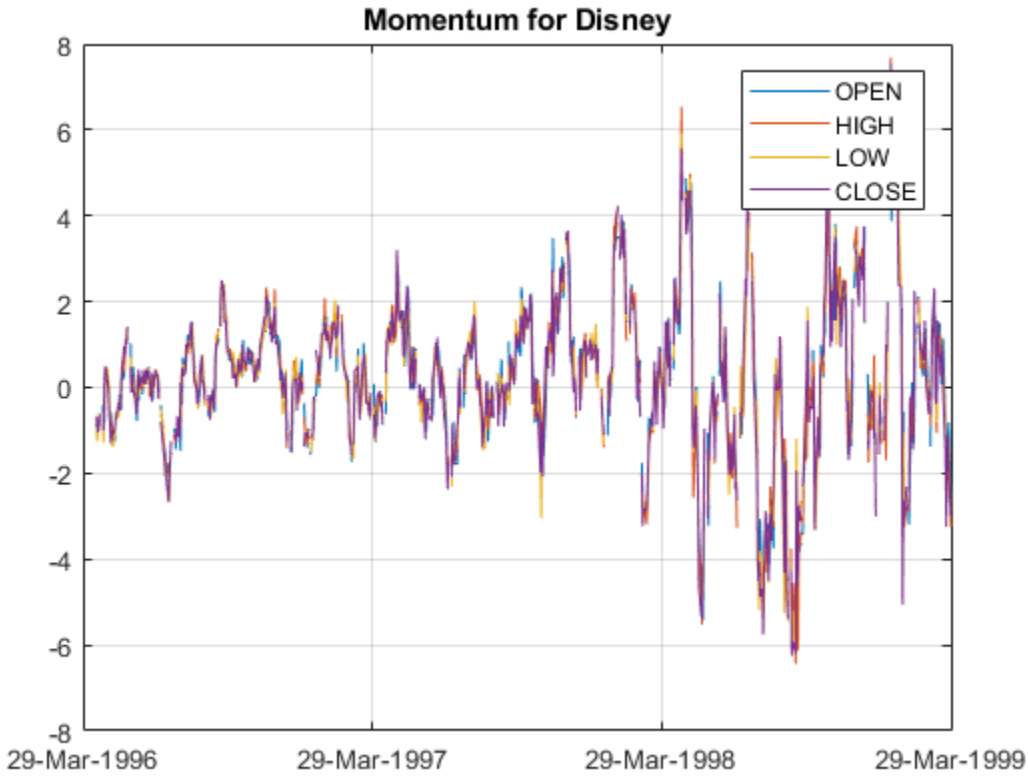
Note, to compute a quantity over `n` periods, you must specify `n+1` for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Calculate the Momentum of a Data Series

This example shows how to calculate the momentum of a data series for Disney stock and plot the results.

```
load disney.mat
dis = rmfield(dis,'VOLUME'); % remove VOLUME field
dis_Mom = tsmom(dis);
plot(dis_Mom)
title('Momentum for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

[1] Kaufman, P. J. *The New Commodity Trading Systems and Methods*. John Wiley and Sons, New York, 1987.

See Also

`tsaccel`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

tsmovavg

Moving average

`tsmovavg` calculates the simple, exponential, triangular, weighted, and modified moving average of a vector or `fints` object of data. For information on working with financial time series (`fints` objects) data, see "Working with Financial Time Series Objects" on page 12-3.

Syntax

```
output = tsmovavg(tsobj,'s',lag)
output = tsmovavg(vector,'s',lag,dim)

output = tsmovavg(tsobj,'e',timeperiod)
output = tsmovavg(vector,'e',timeperiod,dim)

output = tsmovavg(tsobj,'t',numperiod)
output = tsmovavg(vector,'t',numperiod,dim)

output = tsmovavg(tsobj,'w',weights)
output = tsmovavg(vector,'w',weights,dim)

output = tsmovavg(tsobj,'m',numperiod)
output = tsmovavg(vector,'m',numperiod,dim)
```

Description

`output = tsmovavg(tsobj,'s',lag)` returns the simple moving average by for financial time series object, `tsobj`. `lag` indicates the number of previous data points used with the current data point when calculating the moving average.

`output = tsmovavg(vector,'s',lag,dim)` returns the simple moving average for a vector. `lag` indicates the number of previous data points used with the current data point when calculating the moving average.

`output = tsmovavg(tsobj,'e',timeperiod)` returns the exponential weighted moving average for financial time series object, `tsobj`. The exponential moving average

is a weighted moving average, where `timeperiod` specifies the time period. Exponential moving averages reduce the lag by applying more weight to recent prices. For example, a 10-period exponential moving average weights the most recent price by 18.18%. `Exponential Percentage = 2/(TIMEPER + 1) or 2/(WINDOW_SIZE + 1).`

`output = tsmovavg(vector,'e',timeperiod,dim)` returns the exponential weighted moving average for a vector. The exponential moving average is a weighted moving average, where `timeperiod` specifies the time period. Exponential moving averages reduce the lag by applying more weight to recent prices. For example, a 10-period exponential moving average weights the most recent price by 18.18%. (`2/(timeperiod + 1)`).

`output = tsmovavg(tsobj,'t',numperiod)` returns the triangular moving average for financial time series object, `tsobj`. The triangular moving average double-smooths the data. `tsmovavg` calculates the first simple moving average with window width of `ceil(numperiod + 1)/2`. Then it calculates a second simple moving average on the first moving average with the same window size.

`output = tsmovavg(vector,'t',numperiod,dim)` returns the triangular moving average for a vector. The triangular moving average double-smooths the data. `tsmovavg` calculates the first simple moving average with window width of `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)` returns the weighted moving average for the financial time series object, `tsobj`, by supplying weights for each element in the moving window. The length of the weight vector determines the size of the window. If larger weight factors are used for more recent prices and smaller factors for previous prices, the trend is more responsive to recent changes.

`output = tsmovavg(vector,'w',weights,dim)` returns the weighted moving average for the vector by supplying weights for each element in the moving window. The length of the weight vector determines the size of the window. If larger weight factors are used for more recent prices and smaller factors for previous prices, the trend is more responsive to recent changes.

`output = tsmovavg(tsobj,'m',numperiod)` returns the modified moving average for the financial time series object, `tsobj`. The modified moving average is similar to the simple moving average. Consider the argument `numperiod` to be the lag of the simple moving average. The first modified moving average is calculated like a simple moving

average. Subsequent values are calculated by adding the new price and subtracting the last average from the resulting sum.

`output = tsmovavg(vector,'m',numperiod,dim)` returns the modified moving average for the vector. The modified moving average is similar to the simple moving average. Consider the argument `numperiod` to be the lag of the simple moving average. The first modified moving average is calculated like a simple moving average. Subsequent values are calculated by adding the new price and subtracting the last average from the resulting sum.

Examples

Compute Five Forms of Moving Averages Using a Financial Time Series Object

Load the financial time series object, `dis` for Disney stock and look at the weekly data for this time series.

```
load disney.mat
weekly = toweekly(dis);
dates = (weekly.dates);
price = fts2mat(weekly.CLOSE);
```

Set the |lag| input argument for the window size for the moving average.

```
window_size = 12;
```

Calculate the simple moving average.

```
simple = tsmovavg(price,'s',window_size,1);
```

Calculate the exponential weighted moving average moving average.

```
exp = tsmovavg(price,'e',window_size,1);
```

Calculate the triangular moving average moving average.

```
tri = tsmovavg(price,'t',window_size,1);
```

Calculate the weighted moving average moving average.

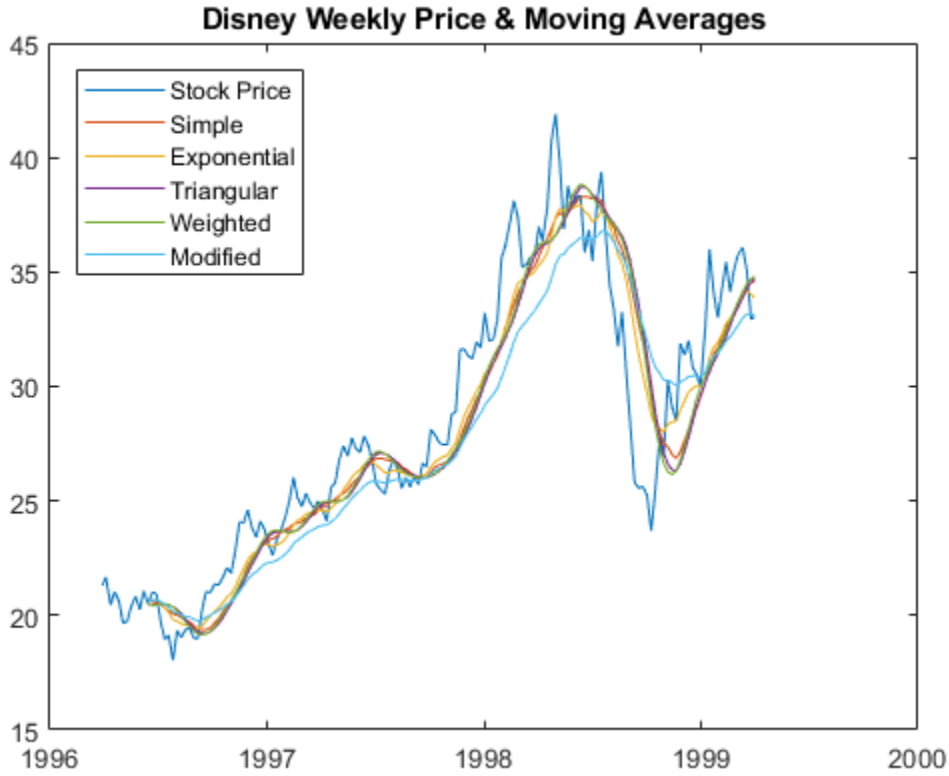
```
semi_gaussian = [0.026 0.045 0.071 0.1 0.12 0.138];
semi_gaussian = [semi_gaussian fliplr(semi_gaussian)];
weighted = tsmovavg(price,'w',semi_gaussian,1);
```

Calculate the modified moving average moving average.

```
modif = tsmovavg(price,'m',window_size,1);
```

Plot the results for the five moving average calculations for Disney stock.

```
plot(dates,price,...
    dates,simple,...
    dates,exp,...
    dates,tri,...
    dates,weighted,...
    dates,modif)
datetick
legend('Stock Price','Simple','Exponential','Triangular','Weighted',...
    'Modified','Location','NorthWest')
title('Disney Weekly Price & Moving Averages')
```



- "Using Time Series to Predict Equity Return" on page 12-25
- "Data Transformation and Frequency Conversion" on page 12-12
- "Working with Financial Time Series Objects" on page 12-3
- "Creating a Financial Time Series Object" on page 13-11
- "Indexing a Financial Time Series Object" on page 12-17
- "Financial Time Series Operations" on page 12-8
- "Using Time Series to Predict Equity Return" on page 12-25
- "Technical Analysis Examples" on page 16-4

Input Arguments

`tsobj` — Financial time series object
object

Financial time series object specified using a time series object created using `fints`.

`'s'` — Indicator for simple moving average
character vector

`lag` is the parameter indicating the number of previous data points to be used in conjunction with the current data point when calculating the simple moving average.

`lag` — Number of previous data points
nonnegative integer

Number of previous data points specified as a nonnegative integer. `Lag` indicates the window size or number of periods of the moving average.

`vector` — Set of observations
vector or matrix

Set of observations specified as a vector or matrix.

`dim` — dimension to operate along
positive integer with value `1` or `2`

Dimension to operate along, specified as a positive integer with a value of `1` or `2`. `dim` is an optional input argument, and if it is not included as an input, the default value `2` is assumed. The default of `dim` = `2` indicates a row-oriented matrix, where each row is a variable and each column is an observation.

If `dim` = `1`, the input is assumed to be a column vector or column-oriented matrix, where each column is a variable and each row an observation.

`'e'` — Indicator for exponential moving average
character vector

Exponential moving average is a weighted moving average, where `timeperiod` is the time period of the exponential moving average. 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%.

```
Exponential Percentage = 2/(TIMEPER + 1) or 2/(WINDOW_SIZE + 1)
```

`timeperiod` — Length of time period
nonnegative integer

Length of time period specified as a nonnegative integer.

`'t'` — Indicator for triangular moving average
character vector

Triangular moving average is a double-smoothing of the data. The first simple moving average is calculated with a window width of `ceil(numperiod + 1)/2`. Then a second simple moving average is calculated on the first moving average with the same window size.

`'m'` — Indicator for modified moving average
character vector

The modified moving average is similar to the simple moving average. Consider the argument `numperiod` to be the `lag` of the simple moving average. The first modified moving average is calculated like a simple moving average. Subsequent values are calculated by adding the new price and subtracting the last average from the resulting sum.

`numperiod` — Number of periods considered
nonnegative integer

Number of periods considered specified as a nonnegative integer.

`'w'` — Indicator for weighted moving average
character vector

A weighted moving average is calculated with a weight vector, `weights`. 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.

`weights` — Weights for each element in the moving window
vector of weights

Weights for each element in the window specified as a vector of weights.

Output Arguments

`output` — Moving average calculation

vector or matrix

Moving average calculation returned as a vector or matrix. The `output` returned from `tsmovavg` is identical in format to the input.

References

[1] Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 184–192.

See Also

`boxcox` | `convert2sur` | `convertto` | `diff` | `fillts` | `filter` | `lagts` | `leadts` | `mean` | `peravg` | `resamplets` | `smoothts`

Topics

“Using Time Series to Predict Equity Return” on page 12-25
“Data Transformation and Frequency Conversion” on page 12-12
“Working with Financial Time Series Objects” on page 12-3
“Creating a Financial Time Series Object” on page 13-11
“Indexing a Financial Time Series Object” on page 12-17
“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25
“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

typprice

Typical price

Syntax

```
tprc = typprice(highp,lowp,closep)

tprc = typprice([highp lowp closep])

tprcts = typprice(tsobj)

tprcts = typprice(tsobj,'ParameterName',ParameterValue, ...)
```

Arguments

`highp`	High price (vector).
`lowp`	Low price (vector).
`closep`	Closing price (vector).
`tsobj`	Financial time series object.

Description

`tprc = typprice(highp,lowp,closep)` calculates the typical prices `tprc` from the high (`highp`), low (`lowp`), and closing (`closep`) prices. The typical price is the average of the high, low, and closing prices for each period.

`tprc = typprice([highp lowp closep])` accepts a three-column matrix as the input rather than two individual vectors. The columns of the matrix represent the high, low, and closing prices, in that order.

`tprcts = typprice(tsobj)` calculates the typical prices from the stock data contained in the financial time series object `tsobj`. The object must contain, at least, the `High`, `Low`, and `Close` data series. The typical price is the average of the closing price

plus the high and low prices. `tprcts` is a financial time series object of the same dates as `tsobj` containing the data series `TypPrice`.

`tprcts = typprice(tsobj,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

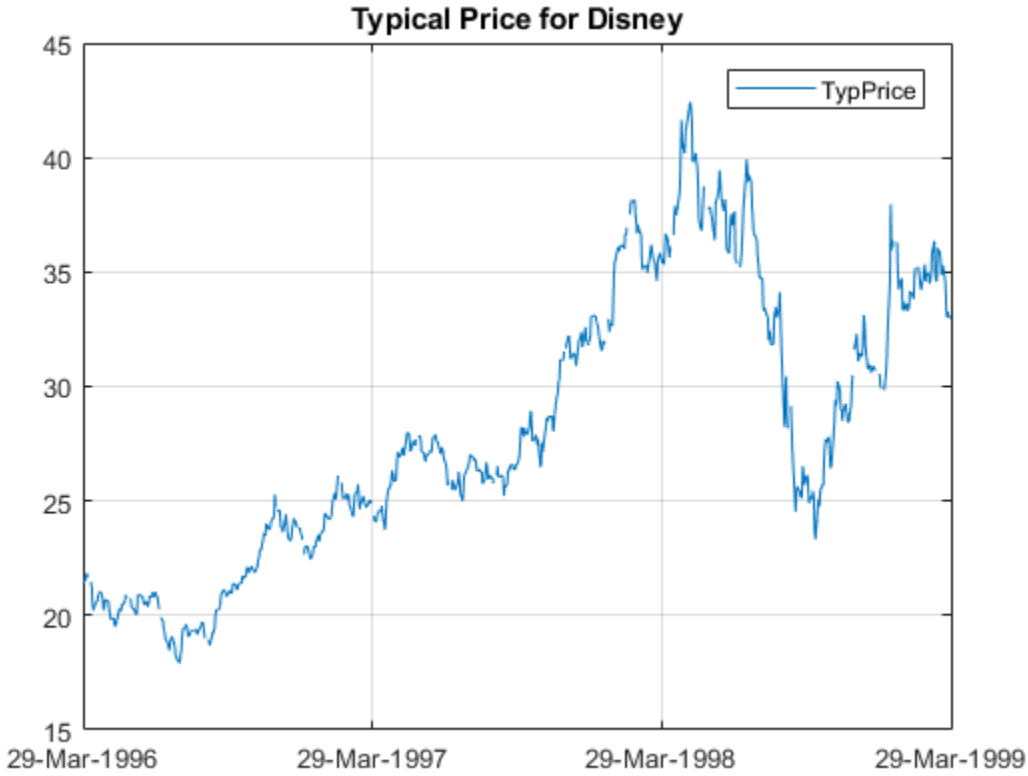
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Typical Price

This example shows how to compute the typical price for Disney stock and plot the results.

```
load disney.mat
dis_Typ = typprice(dis);
plot(dis_Typ)
title('Typical Price for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 291–292.

See Also

`medprice` | `wclose`

Topics

“Technical Analysis Examples” on page 16-4
“Technical Indicators” on page 16-2

Introduced before R2006a

uicalendar

Graphical calendar

Syntax

```
uicalendar(Name,Value)
```

Description

`uicalendar(Name,Value)` supports a customizable graphical calendar that interfaces with one or more `uicontrol`. `uicalendar` populates one or more `uicontrol` with user-selected dates.

Examples

Use `uicalendar` with an `uicontrol`

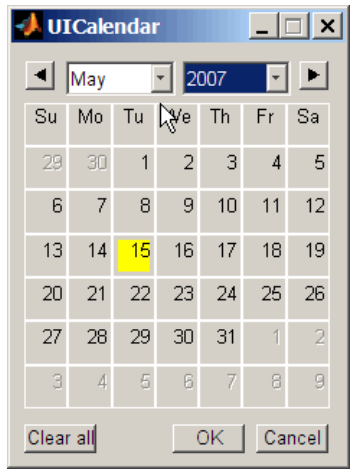
Create an `uicontrol`:

```
textH1 = uicontrol('style', 'edit', 'position', [10 10 100 20])
```

```
textH1 =

  UIControl with properties:

             Style: 'edit'
            String: ''
   BackgroundColor: [0.9400 0.9400 0.9400]
          Callback: ''
             Value: 0
          Position: [10 10 100 20]
             Units: 'pixels'
```

Call UICalendar:

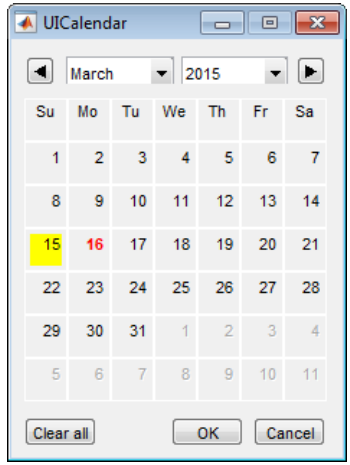
```
uicalendar('DestinationUI', {textH1, 'string'})
```



Select a date and click `OK`.

Alternatively, you can use datetime arrays for `InitDate` and `Holiday`.

```
uicalendar('InitDate',datetime('15-Mar-2015','Locale','en_US'),'Holiday',datetime('16-Mar-2015','Locale','en_US'))
```



Select a date and click `OK`. For more information on using `uicalendar` with an application, see "Example of Using UICalendar with an Application" on page 15-5.

- "Handle and Convert Dates" on page 2-2

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `uicalendar('InitDate',datetime('15-Mar-2015','Locale','en_US'),'Holiday',datetime('16-Mar-2015','Locale','en_US'))`

`BusDays` — Flag to indicate nonbusiness days

`0` (Standard calendar without nonbusiness day indicators) (default) | numeric values of `0` or `1`

Flag to indicate nonbusiness days, specified using numeric values of `0` or `1`. The values are:

- `0` — (Default) Standard calendar without nonbusiness day indicators.
- `1` — Marks NYSE nonbusiness days in red.

Data Types: `logical`

`BusDaySelect` — Flag to indicate whether business and nonbusiness days

`1` (Allows selections of business and nonbusiness days) (default) | numeric values of `0` or `1`

Flag to indicate whether business and nonbusiness days, specified using numeric values of `0` or `1`. The values are:

- 0 — Only allow selection of business days. Nonbusiness days are determined from the following parameters:

- 'BusDays'
- 'Holiday'
- 'Weekend'

- 1 — (Default) Allows selections of business and nonbusiness days.

Data Types: logical

DateBoxColor — Color of date squares
[date R G B]

Color of date squares, specified using [date R G B], where [R G B] is the color.

Data Types: double

DateStrColor — Color of numeric date number in the date square
[date R G B]

Color of numeric date number in the date square, specified using [date R G B], where [R G B] is the color.

Data Types: double

DestinationUI — Destination object's handles
'string' (default UI property populated with dates) (default) | values are H or {H, {Prop}}

Destination object's handles, specified with values H or {H, {Prop}}. The values are:

- H — Scalar or vector of the destination object's handles. The default UI property that is populated with the dates is a character vector.
- {H, {Prop}} — Cell array of handles and the destination object's UI properties. H must be a scalar or vector and Prop must be a single property character vector or a cell array of property character vectors.

Data Types: char | cell

Holiday — Holiday dates in calendar
serial date numbers | datetime arays

Holiday dates in calendar, specified using a scalar or vector of serial date numbers or datetime arrays. The corresponding date character vector of the holiday appears Red.

Data Types: `double` | `datetime`

`InitDate` — Initial start date when calendar is initialized

`TODAY` (default) | serial date number | datetime array | date character vector

Initial start date when calendar is initialized, specified with date values using a serial date number, datetime array, or date character vector. The values are:

- `Datenum` — Numeric or datetime array date value specifying the initial start date when the calendar is initialized. The default date is `TODAY`.
- `Datestr` — Date character vector value specifying the initial start date when the calendar is initialized. `Datestr` must include a Year, Month, and Day (for example, 01-Jan-2006).

Data Types: `double` | `char` | `datetime`

`InputDateFormat` — Format of initial start date

character vector

Format of initial start date (`InitDate`), specified using a character vector. See`datestr` for date format values.

Data Types: `double` | `datetime`

`OutputDateFormat` — Format of output date

character vector

Format of output date, specified using a character vector. See`datestr` for date format values.

Data Types: `double` | `datetime`

`OutputDateStyle` — Style for output date

`0` (default) | numeric value of `0`, `1`, `2`, or `3`

Style for output date, specified using a value of `0`, `1`, `2`, or `3`. The values are:

- `0` — (Default) Returns a single date character vector or a cell array (row) of date character vectors. For example, `{'01-Jan-2001, 02-Jan-2001, ...'}`.
- `1` — Returns a single date character vector or a cell (column) array of date character vectors. For example, `{'01-Jan-2001; 02-Jan-2001; ...'}`.

- `2` — Returns a character vector representation of a row vector of datenums. For example, `'[732758, 732759, 732760, 732761]'`.
- `3` — Returns a character vector representation of a column vector of datenums. For example, `'[732758; 732759; 732760; 732761]'`.

Data Types: `double`

`SelectionType` — Flag for date selection

`1` (default) | numeric value of `0` or `1`

Flag for date selection, specified with using a value of `0` or `1`. The values are:

- `0` — Allows multiple date selections.
- `1` — (Default) Allows only a single date selection.

Data Types: `logical`

`Weekend` — Define weekend days

`1` (default) | numeric values of `1` through `7`

Define weekend days, specified using a value of `1` through `7`. Weekend days are marked in red. `DayOfWeek` can be a vector containing the following numeric values:

- `1` — Sunday
- `2` — Monday
- `3` — Tuesday
- `4` — Wednesday
- `5` — Thursday
- `6` — Friday
- `7` — Saturday

Also this value can be a vector of length 7 containing 0's and 1's. The value 1 indicates a weekend day. The first element of this vector corresponds to Sunday. For example, when Saturday and Sunday are weekend days then `WEEKEND = [1 0 0 0 0 0 1]`.

Data Types: `double`

`WindowStyle` — Window figure properties

`Normal` (default) | character vector with value of `Normal` or `Model`

Window figure properties, specified with using a character vector with a value of `Normal` or `Model`. The values are:

- `Normal` — (Default) Standard figure properties.
- `Modal` — Modal figures remain stacked above all normal figures and the MATLAB Command Window.

Data Types: `char`

See Also

`datetime` | `holidays`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2
"UICalendar User Interface" on page 15-4

Introduced before R2006a

uminus

Unary minus of financial time series object

Syntax

```
uminus
```

Description

`uminus` implements unary minus for a financial time series object.

See Also

`uplus`

Topics

“Financial Time Series Operations” on page 12-8
“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

uplus

Unary plus of financial time series object

Syntax

```
uplus
```

Description

`uplus` implements unary plus for a financial time series object.

See Also

`uminus`

Topics

"Financial Time Series Operations" on page 12-8
"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

var

Variance

Syntax

```
y = var(X)

y = var(X,1)

y = var(X,W)

y = var(X,W,DIM)
```

Arguments

X	Financial times series object.
W	Weight vector used in calculating variance.
DIM	Dimension of X used in calculating variance.

Description

`var` supports financial time series objects based on the MATLAB `var` function. See `var`.

`y = var(X)`, if `X` is a financial time series object and returns the variance of each series.

`var` normalizes `y` by `N – 1` if `N > 1`, where `N` is the sample size. This is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples. For `N = 1`, `y` is normalized by `N`.

`y = var(X,1)` normalizes by `N` and produces the second moment of the sample about its mean. `var(X, 0)` is the same as `var(X)`.

`y = var(X,W)` computes the variance using the weight vector `W`. The length of `W` must equal the length of the dimension over which `var` operates, and its elements must be

nonnegative. `var` normalizes `W` to sum to `1`. Use a value of `0` for `W` to use the default normalization by `N – 1`, or use a value of `1` to use `N`.

`y = var(X,W,DIM)` takes the variance along the dimension `DIM` of `X`.

Examples

The variance is the square of the standard deviation. Consider if

```
f = fints((today:today+1)', [4 -2 1; 9  5 7])
```

then

```
var(f, 0, 1)
```

is

```
[12.5 24.5 18.0]
```

and

```
var(f, 0, 2)
```

is

```
[9.0; 4.0]
```

See Also

`corrcoef` | `cov` | `mean` | `std`

Topics

“Using Time Series to Predict Equity Return” on page 12-25

Introduced before R2006a

vertcat

Concatenate financial time series objects vertically

Syntax

```
vertcat
```

Description

`vertcat` implements vertical concatenation of financial time series objects. `vertcat` essentially adds data points to a time series object. Objects to be vertically concatenated must not have any duplicate dates and/or times or any overlapping dates and/or times. The description fields are concatenated as well. They are separated by `||`.

Examples

Create two financial time series objects with daily frequencies:

```
myfts   = fints((today:today+4)', (1:5)', 'DataSeries', 'd');
yourfts = fints((today+5:today+9)', (11:15)', 'DataSeries', 'd');
```

Use `vertcat` to concatenate them vertically:

```
newfts1 = [myfts; yourfts]
```

```
newfts1 =

    desc:   ||
    freq:  Daily (1)

    'dates:  (10)'    'DataSeries:  (10)'
    '11-Dec-2001'     [                1]
    '12-Dec-2001'     [                2]
    '13-Dec-2001'     [                3]
    '14-Dec-2001'     [                4]
    '15-Dec-2001'     [                5]
    '16-Dec-2001'     [               11]
```

```
    '17-Dec-2001'      [                12]
    '18-Dec-2001'      [                13]
    '19-Dec-2001'      [                14]
    '20-Dec-2001'      [                15]
```

Create two financial time series objects with different frequencies:

```
myfts   = fints((today:today+4)', (1:5)', 'DataSeries', 'd');
hisfts  = fints((today+5:7:today+34)', (11:15)', 'DataSeries',...
'w');
```

Concatenate these two objects vertically:

```
newfts2 = [myfts; hisfts]
```

```
 newfts2 =

    desc:   ||
    freq:  Unknown (0)

    'dates:  (10)'    'DataSeries:  (10)'
    '11-Dec-2001'      [                 1]
    '12-Dec-2001'      [                 2]
    '13-Dec-2001'      [                 3]
    '14-Dec-2001'      [                 4]
    '15-Dec-2001'      [                 5]
    '16-Dec-2001'      [                11]
    '23-Dec-2001'      [                12]
    '30-Dec-2001'      [                13]
    '06-Jan-2002'      [                14]
    '13-Jan-2002'      [                15]
```

If all frequency indicators are the same, the new object has the same frequency indicator. However, if one of the concatenated objects has a different `freq` from the other(s), the frequency of the resulting object is set to `Unknown (0)`. In these examples, `newfts1` has `Daily` frequency, while `newfts2` has `Unknown (0)` frequency.

See Also

`horzcat`

Topics

“Financial Time Series Operations” on page 12-8

"Using Time Series to Predict Equity Return" on page 12-25

Introduced before R2006a

volarea

Price and volume chart

Syntax

```
volarea(X)
```

Arguments

X	X can be a table or a M-by-3 matrix. If X is a table, the first column of dates can be serial date numbers, date character vectors, or datetime arrays. If X is a M-by-3 matrix, the first column contains date numbers, the second column is the asset price, and the third column is the volume.

Description

`volarea(X)` plots asset date, price, and volume on a single axis.

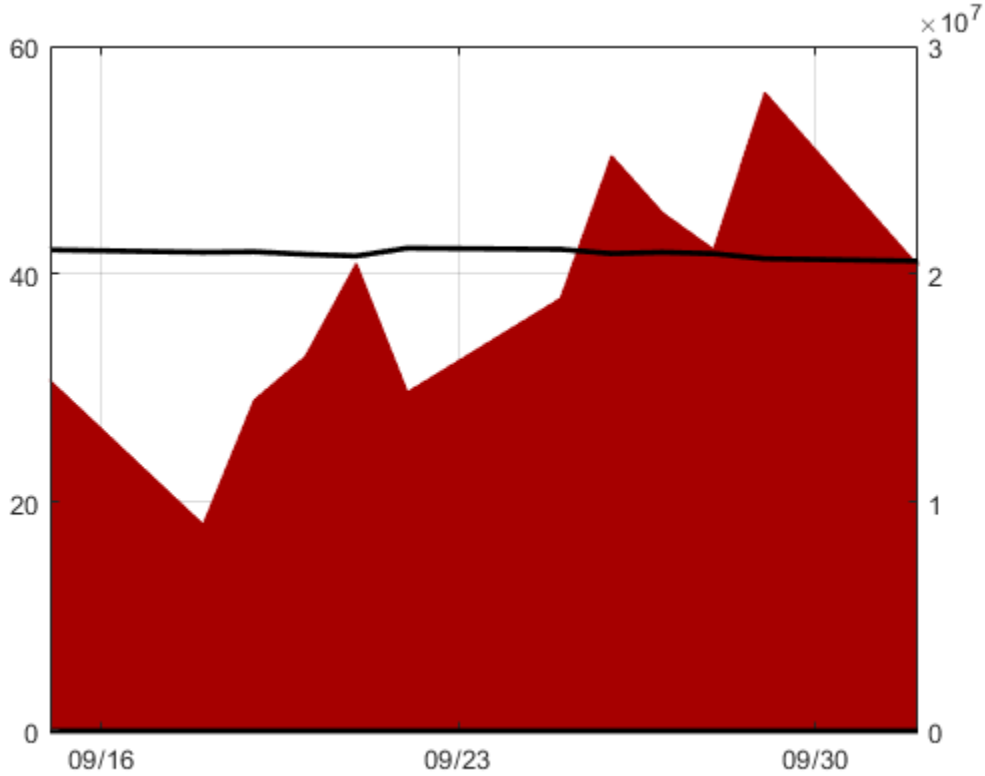
Examples

Plot Asset Date, Price, and Volume on a Single Axis

This example shows how to plot asset date, price, and volume on a single axis, given asset X as an M-by-3 matrix of date numbers, asset price, and volume.

```
X = [...

733299.00         41.99   15045445.00;...
733300.00         42.14   15346658.00;...
733303.00         41.93    9034397.00;...
733304.00         41.98   14486275.00;...
```

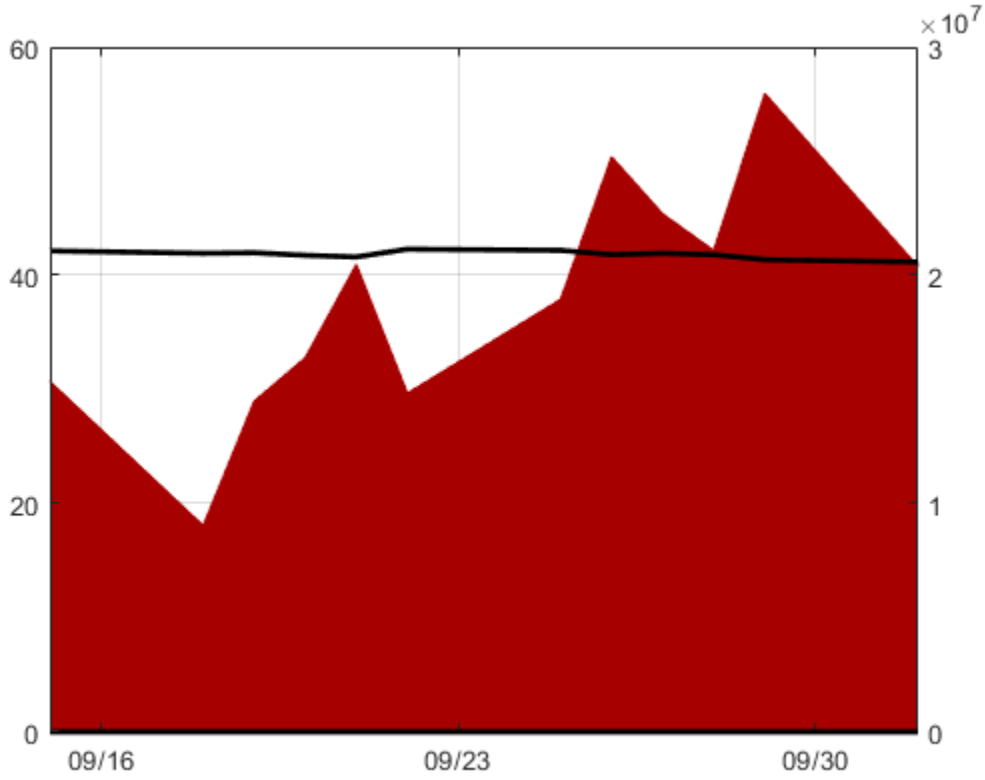
```
733305.00          41.75   16389872.00;...
733306.00          41.61   20475208.00;...
733307.00          42.29   14833200.00;...
733310.00          42.19   18945176.00;...
733311.00          41.82   25188101.00;...
733312.00          41.93   22689878.00;...
733313.00          41.81   21084723.00;...
733314.00          41.37   27963619.00;...
733317.00          41.17   20385033.00;...
733318.00          42.02   27783775.00];

volarea(X)
```



Plot Asset Date, Price, and Volume on a Single Axis Using datetime Input

This example shows how to use `datetime` input to plot asset date, price, and volume on a single axis, given asset `X` as an `M`-by-3 matrix of date numbers, asset price, and volume.

```
X = [...

733299.00          41.99   15045445.00;...
733300.00          42.14   15346658.00;...
733303.00          41.93    9034397.00;...
733304.00          41.98   14486275.00;...
733305.00          41.75   16389872.00;...
733306.00          41.61   20475208.00;...
733307.00          42.29   14833200.00;...
733310.00          42.19   18945176.00;...
733311.00          41.82   25188101.00;...
733312.00          41.93   22689878.00;...
733313.00          41.81   21084723.00;...
733314.00          41.37   27963619.00;...
733317.00          41.17   20385033.00;...
733318.00          42.02   27783775.00];

t=array2table(X);
t.X1 = datetime(t.X1,'ConvertFrom','datenum','Locale','en_US');
volarea(t);
```



- “Charting Financial Data” on page 2-14

See Also

`bolling` | `candle` | `datetime` | `highlow` | `kagi` | `linebreak` | `movavg` | `pointfig` | `priceandvol` | `renko`

Topics

“Charting Financial Data” on page 2-14

Introduced in R2008a

volroc

Volume rate of change

Syntax

```
vroc = volroc(tvolume,nTimes)

vrocts = volroc(tsobj,nTimes)

vrocts = volroc(tsobj,nTimes,'ParameterName',ParameterValue, ...)
```

Arguments

`tvolume`	Volume traded.
`nTimes`	(Optional) Time difference. Default = `12`.
`tsobj`	Financial time series object.

Description

`vroc = volroc(tvolume,nTimes)` calculates the volume rate of change, `vroc`, from the volume traded data `tvolume`. If `nTimes` is specified, the volume rate of change is calculated between the current volume and the volume `nTimes` ago.

`vrocts = volroc(tsobj,nTimes)` calculates the volume rate of change, `vrocts`, from the financial time series object `tsobj`. The `vrocts` output is a financial time series object with similar dates as `tsobj` and a data series named `VolumeROC`. If `nTimes` is specified, the volume rate of change is calculated between the current volume and the volume `nTimes` ago.

`vrocts = volroc(tsobj,nTimes,'ParameterName',ParameterValue, ...)` specifies the name for the required data series when it is different from the default name. The valid parameter name is

- `VolumeName`: volume traded series name

The parameter value is a character vector that represents the valid parameter name.

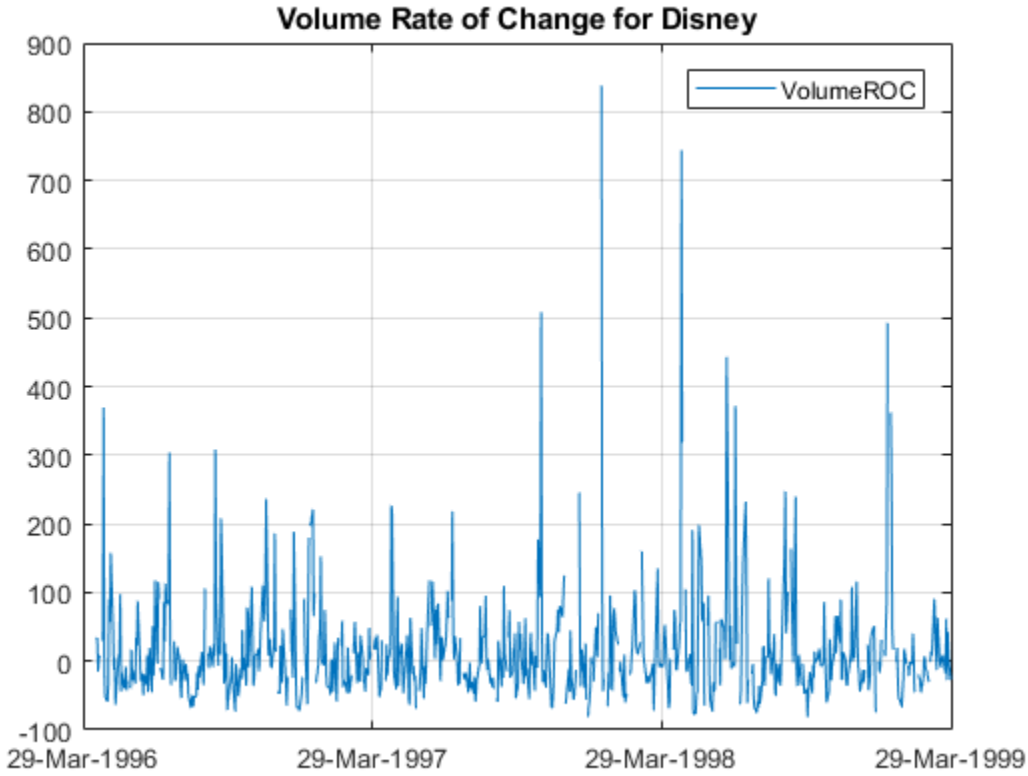
Note, to compute a quantity over `n` periods, you must specify `n+1` for `nTimes`. If you specify `nTimes = 0`, the function returns your original time series.

Examples

Compute the Volume Rate of Change

This example shows how to compute the volume rate of change for Disney stock and plot the results.

```
load disney.mat
dis_VolRoc = volroc(dis);
plot(dis_VolRoc)
title('Volume Rate of Change for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 310–311.

See Also

`prcroc`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

wclose

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

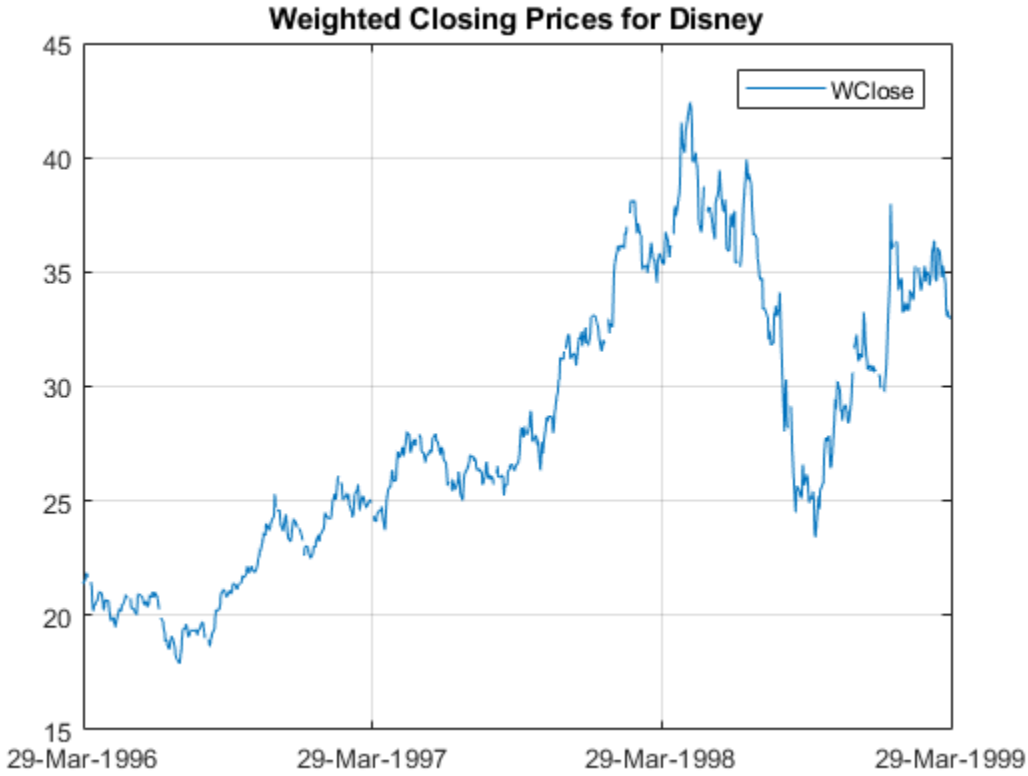
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Weighted Closing Prices

This example shows how to compute the weighted closing prices for Disney stock and plot the results.

```
load disney.mat
dis_Wclose = wclose(dis);
plot(dis_Wclose)
title('Weighted Closing Prices for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 312–313.

See Also

`medprice` | `typprice`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

weeknum

Week in year

Syntax

```
[N] = weeknum(D)
[N] = weeknum(___,W,E)
```

Description

`[N] = weeknum(D)` returns the week in year. The `weeknum` function considers the week containing January 1 to be the first week of the year.

`[N] = weeknum(___,W,E)` returns the week in year using the optional input arguments for `W` and `E`. The `weeknum` function considers the week containing January 1 to be the first week of the year.

Examples

Determine the Week of the Year

Determine the week of the year using a serial date number.

```
N = weeknum(728647)
```

```
N = 52
```

Determine the week of the year using a character vector.

```
N = weeknum('19-Dec-1994')
```

```
N = 52
```

Determine the week of the year using a datetime array.

```
N = weeknum(datetime('19-Dec-1994','Locale','en_US'))
```

```
N = 52
```

The first week of the year must have at least four days in it. For example, January 8, 2004 was a Thursday. The European standard is used because the first week of the year is the first week longer than three days.

```
weeknum('08-jan-2004',1,1)
```

```
ans = 1
```

You can also use `weeknum` with `datenum`.

```
weeknum(datenum('01-Jan-2004'):datenum('08-Jan-2004'))
```

```
ans =

     1     1     1     2     2     2     2     2
```

The default start day of the week is Sunday. Every day after, and including the first Sunday of the year (04-Jan-2004), returns `2` denoting the second week. In this case, the first of week of the year started before January 1, 2004. You can also use `weeknum` with `datenum` and specify a `D` value of `5` to indicate that the weeks start on Thursday.

```
weeknum(datenum('01-Jan-2004'):datenum('08-Jan-2004'),5)
```

```
ans =

     1     1     1     1     1     1     1     2
```

The first week of the year that has four or more days, based on the specified start day, is considered week one (even if this is not the first week in the calendar). Any day falling in (or before) this week is given a week number of 1.

- “Handle and Convert Dates” on page 2-2

Input Arguments

D — Date to determine week in year

serial date number | data character vector | datetime array

Date to determine week in year, specified as a serial date number, date character vector, or datetime array.

Serial date numbers can be a matrix. Date character vectors can be specified as a one-dimensional cell array of character vectors. All the date character vectors must have the same format.

Use the function `datestr` to convert serial date numbers to formatted date character vectors.

Data Types: `single` | `double` | `char` | `datetime`

W — Day a week begins

`1` (default) | integer with value `1` through `7` | vector of integers with values `1` through `7`

Day a week begins, specified as an integer or a vector of integers from `1` through `7`.

- `1` — Sunday (default)
- `2` — Monday
- `3` — Tuesday
- `4` — Wednesday
- `5` — Thursday
- `6` — Friday
- `7` — Saturday

The `weeknum` function considers the week containing January 1 to be the first week of the year.

Data Types: `single` | `double`

E — Flag indicates if week of year display is European standard

`0` (default) | numeric with values `1` or `0`

Flag indicates if week of year display is European standard, specified as `1` (to use the European standard) or `0` (not to use the European standard).

The European standard considers first week of year to be first week longer than three days, offset by the given week's start day.

Data Types: `logical`

Output Arguments

`N` — Week number of the year, given `D`

numeric | column vector

Week number of the year, given `D`, returned as a numeric value, given `D`, a serial date number, date character vector, or datetime array. If `D` is a one-dimensional cell array of character vectors, then `weeknum` returns a column vector of *M* week numbers, where *M* is the number of character vectors in `D`.

If the optional input arguments `W` and `E` are defined, the week of the year is in the European standard.

See Also

`datenum` | `datestr` | `datetime` | `datevec` | `day`

Topics

"Handle and Convert Dates" on page 2-2

Introduced before R2006a

weights2holdings

Portfolio values and weights into holdings

Syntax

```
Holdings = weights2holdings(Values,Weights,Prices)
```

Arguments

`Values`	Scalar or number of portfolios (`NPORTS`) vector containing portfolio values.
`Weights`	`NPORTS` by number of assets (`NASSETS`) matrix with portfolio weights. The weights sum to the value of a `Budget` constraint, which is usually 1. (See `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`

Topics

“Data Transformation and Frequency Conversion” on page 12-12

Introduced before R2006a

willad

Williams Accumulation/Distribution line

Syntax

```
wadl = willad(highp,lowp,closep)

wadl = willad([highp lowp closep])

wadlts = willad(tsobj)

wadlts = willad(tsobj,'ParameterName',ParameterValue, ...)
```

Arguments

`highp`	High price (vector)
`lowp`	Low price (vector)
`closep`	Closing price (vector)
`tsobj`	Time series object

Description

`wadl = willad(highp,lowp,closep)` computes the Williams Accumulation/Distribution line for a set of stock price data. The prices needed for this function are the high (`highp`), low (`lowp`), and closing (`closep`) prices. All three are required.

`wadl = willad([highp lowp closep])` accepts a three-column matrix of prices as input. The first column contains the high prices, the second contains the low prices, and the third contains the closing prices.

`wadlts = willad(tsobj)` computes the Williams Accumulation/Distribution line for a set of stock price data contained in the financial time series object `tsobj`. The object must contain the high, low, and closing prices needed for this function. The function

assumes that the series are named `High`, `Low`, and `Close`. All three are required. `wadlts` is a financial time series object with the same dates as `tsobj` and a single data series named `WillAD`.

`wadlts = willad(tsobj,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

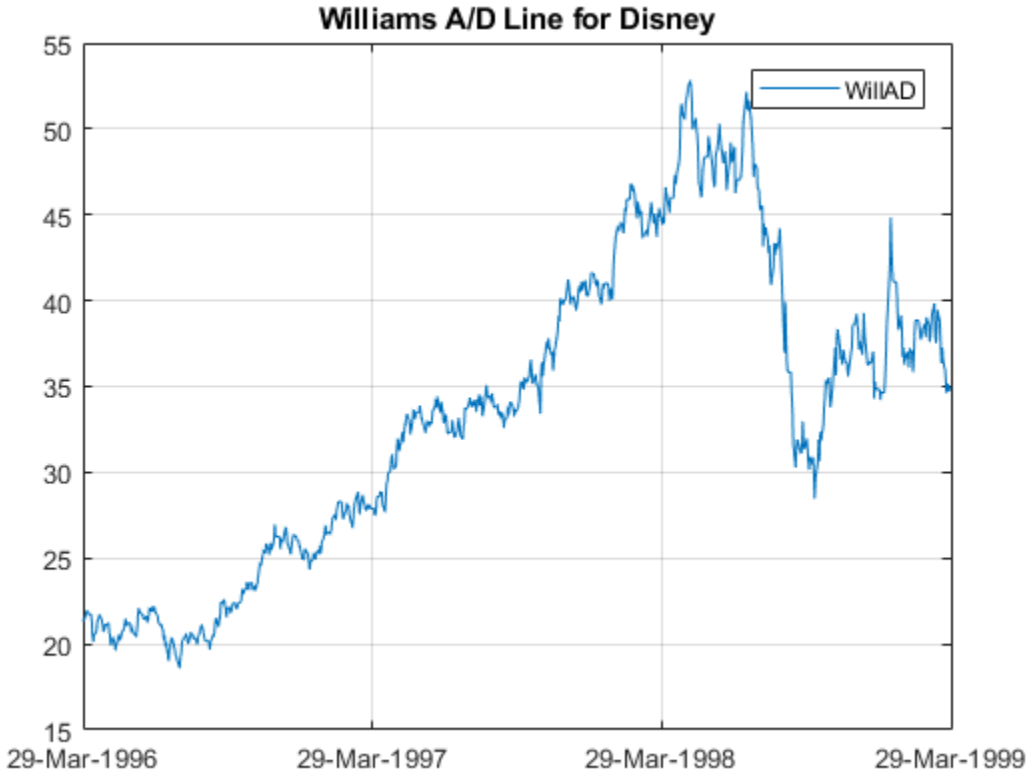
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Williams A/D Line

This example shows how to compute the Williams A/D line for Disney stock and plot the results.

```
load disney.mat
dis_Willad = willad(dis);
plot(dis_Willad)
title('Williams A/D Line for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 314–315.

See Also

`adline` | `adosc` | `willpctr`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

willpctr

Williams %R

Syntax

```
wpctr = willpctr(highp,lowp,closep,nperiods)

wpctr = willpctr([highp, lowp, closep],nperiods)

wpctrts = willpctr(tsobj)

wpctrts = willpctr(tsobj,nperiods)

wpctrts = willpctr(tsobj,nperiods,'ParameterName',ParameterValue,... )
```

Arguments

`highp`	High price (vector)
`lowp`	Low price (vector)
`closep`	Closing price (vector)
`nperiods`	Number of periods (scalar). Default = `14`.
`tsobj`	Financial time series object

Description

`wpctr = willpctr(highp,lowp,closep,nperiods)` calculates the Williams %R values for the given set of stock prices for a specified number of periods `nperiods`. The stock prices needed are the high (`highp`), low (`lowp`), and closing (`closep`) prices. `wpctr` is a vector that represents the Williams %R values from the stock data.

`wpctr = willpctr([highp, lowp, closep],nperiods)` accepts the price input as a three-column matrix representing the high, low, and closing prices, in that order.

`wpctrts = willpctr(tsobj)` calculates the Williams %R values for the financial time series object `tsobj`. The object must contain at least three data series named `High` (high prices), `Low` (low prices), and `Close` (closing prices). `wpctrts` is a financial time series object with the same dates as `tsobj` and a single data series named `WillPctR`.

`wpctrts = willpctr(tsobj,nperiods)` calculates the Williams %R values for the financial time series object `tsobj` for `nperiods` periods.

`wpctrts = willpctr(tsobj,nperiods,'ParameterName',ParameterValue, ...)` accepts parameter name/parameter value pairs as input. These pairs specify the name(s) for the required data series if it is different from the expected default name(s). Valid parameter names are

- `HighName`: high prices series name
- `LowName`: low prices series name
- `CloseName`: closing prices series name

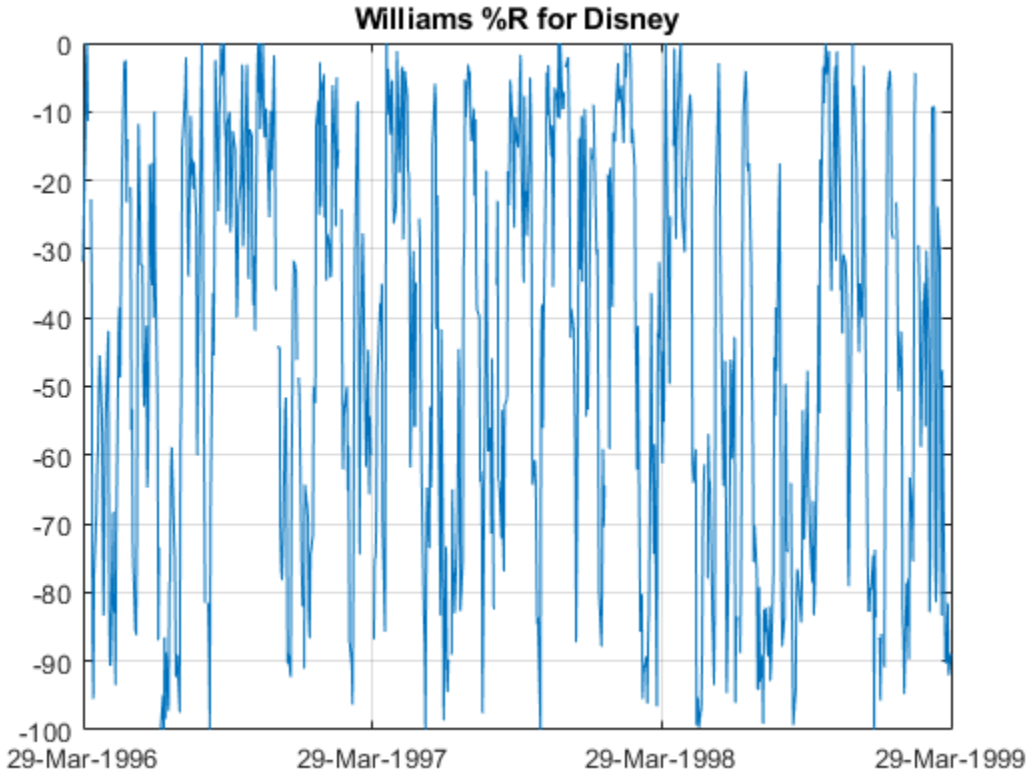
Parameter values are the character vectors that represent the valid parameter names.

Examples

Compute the Williams %R Values

This example shows how to compute the Williams %R values for Disney stock and plot the results.

```
load disney.mat
dis_Wpctr = willpctr(dis);
plot(dis_Wpctr)
legend('off');
title('Williams %R for Disney')
```



- “Technical Analysis Examples” on page 16-4

References

Achelis, Steven B. *Technical Analysis from A to Z*. Second Edition. McGraw-Hill, 1995, pp. 316–317.

See Also

`stochosc` | `willad`

Topics

"Technical Analysis Examples" on page 16-4
"Technical Indicators" on page 16-2

Introduced before R2006a

wrkdydif

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` inclusive. `Holidays` is the number of holidays between the given dates, an integer.

Examples

Determine the Number of Working Days Between a `StartDate` and `EndDate`

Determine `Days` using date character vectors for `StartDate` and `EndDate`.

```
Days = wrkdydif('9/1/2000', '9/11/2000', 1)
```

```
Days = 6
```

Determine `Days` using serial date numbers for `StartDate` and `EndDate`.

```
Days = wrkdydif(730730, 730740, 1)
```

```
Days = 6
```

Determine `Days` using a datetime array for `EndDate`.

```
Days = wrkdydif('9/1/2000', datetime('11-Sep-2000','Locale','en_US'), 1)
```

```
Days = 6
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Holidays` — Holidays between `StartDate` and `EndDate`

vector of integers

Holidays between the `StartDate` and `EndDate`, specified as an `N`-by-`1` or `1`-by-`N` vector of integers.

Data Types: `single` | `double`

Output Arguments

`Days` — Number of working days between dates `StartDate` and `EndDate` inclusive

integer

Number of working days between dates `StartDate` and `EndDate` inclusive, returned an `N`-by-`1` or `1`-by-`N` vector of integers.

See Also

`busdate` | `datetime` | `datewrkdy` | `days365` | `daysact` | `daysdif` | `holidays` | `yearfrac`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2
"UICalendar User Interface" on page 15-4

Introduced before R2006a

x2mdate

Excel serial date number to MATLAB serial date number or datetime format

Syntax

```
MATLABDate = x2mdate(ExcelDateNumber,Convention)
MATLABDate = x2mdate(___,outputType)
```

Description

`MATLABDate = x2mdate(ExcelDateNumber,Convention)` converts Excel serial date numbers to MATLAB serial date numbers or datetime format.

MATLAB date numbers start with `1` = January 1, 0000 A.D., hence there is a difference of 693960 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with Spreadsheet Link software.

`MATLABDate = x2mdate(___,outputType)` converts Excel serial date numbers to MATLAB serial date numbers or datetime format using an optional input argument for `outputType`.

The type of output is determined by an optional `outputType` input. If `outputType` is `'datenum'`, then `MATLABDate` is a serial date number. If `outputType` is `'datetime'`, then `MATLABDate` is a datetime array. By default, `outputType` is `'datenum'`.

Examples

Convert Excel Serial Date Numbers to MATLAB Dates

Given Excel® date numbers in the 1904 system, convert them to MATLAB® serial date numbers, and then to date character vectors.

```
ExDates = [35423  35788  36153];
MATLABDate = x2mdate(ExDates, 1)
```

```
MATLABDate =

      730845      731210      731575


datestr(MATLABDate)

ans = 3x11 char array
    '25-Dec-2000'
    '25-Dec-2001'
    '25-Dec-2002'
```

Alternatively, use the optional input `outputType` to specify `'datetime'` to return datetime format.

```
ExDates = [35423  35788  36153];
MATLABDate = x2mdate(ExDates, 1,'datetime')

MATLABDate = 1x3 datetime array
   25-Dec-2000   25-Dec-2001   25-Dec-2002
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`ExcelDateNumber` — Excel serial date number

serial date number

Excel serial date number, specified as a scalar or vector ofExcel serial date numbers.

Data Types: `double`

`Convention` — Flag for Excel date system

`0` (Excel 1900 date system is in effect) (default) | numeric with value `0` or `1`

Flag for Excel date system, specified as a scalar or vector as a numeric with a value `0` or `1`. `Convention` must be either a scalar or else must be the same size as `ExcelDateNumber`.

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.

Due to a software limitation in Excel software, the year 1900 is considered a leap year. As a result, all DATEVALUE's reported by Excel software between Jan. 1, 1900 and Feb. 28, 1900 (inclusive) differs from the values reported by 1. For example:

- In Excel software, Jan. 1, 1900 = `1`
- In MATLAB, Jan. 1, 1900 – `693960` (for 1900 date system) = `2`

```
datenum('Jan 1, 1900') - 693960

ans =

     2
```

Data Types: `logical`

`outputType` — Output date format
`'datenum'` (default) | character vector with values `'datenum'` or `'datetime'`

Output date format, specified as a character vector with values `'datenum'` or `'datetime'`. The output `MATLABDate` is in serial date format if `'datenum'` is specified or datetime format if `'datetime'` is specified. By default the output is in serial date format.

Data Types: `char`

Output Arguments

`MATLABDate` — MATLAB date
serial date numbers | datetime format

MATLAB date, returned as serial date numbers or datetime format.

The type of output is determined by an optional `outputType` input argument. If `outputType` is `'datenum'`, then `MATLABDate` is a serial date number. If `outputType` is

`'datetime'`, then `MATLABDate` is a datetime array. By default, `outputType` is `'datenum'`.

See Also

`datenum` | `datestr` | `datetime` | `m2xdate`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

xirr

Internal rate of return for nonperiodic cash flow

Syntax

```
Return = xirr(CashFlow, CashFlowDates)
Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations,Basis)
```

Description

`Return = xirr(CashFlow, CashFlowDates)` returns the internal rate of return for a schedule of nonperiodic cash flows.

`Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations,Basis)` returns the internal rate of return for a schedule of nonperiodic cash flows with optional inputs.

Input Arguments

`CashFlow`

A vector or matrix of cash flows. If `CashFlow` is a matrix, each column represents a separate stream of cash flows whose internal rate of return is calculated. The first cash flow of each stream is the initial investment, entered as a negative number.

`CashFlowDates`

(Required) `CashFlowDates` is specified as serial date numbers, date character vectors, or datetime arrays. The size of the input date numbers for `CashFlowDates` must the same size as `CashFlow`, Each column of `CashFlowDate` represents the dates of the corresponding column of `CashFlow`.

Guess

The initial estimate of the internal rate of return. `Guess` is a scalar applied to all streams, or a vector the same length as the number of streams.

Default: `0.1` (10%)

MaxIterations

The positive integer number of iterations used by Newton's method to solve the internal rate of return. `MaxIterations` is a scalar applied to all streams, or a vector the same length as the number of streams.

Default: `50`

Basis

Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Default: `0`

Output Arguments

`Return`

Vector of the annualized internal rate of return of each cash flow stream. A `NaN` indicates that a solution is not found.

Examples

Find the internal rate of return for an investment of $10,000 that returns the following nonperiodic cash flow. The original investment is the first cash flow and is a negative number.

Cash Flow	Dates
($10000)	January 12, 2007
$2500	February 14, 2008
$2000	March 3, 2008
$3000	June 14, 2008
$4000	December 1, 2008

Calculate the internal rate of return for this nonperiodic cash flow:

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
CashFlowDates = ['01/12/2007'
                 '02/14/2008'
                 '03/03/2008'
                 '06/14/2008'
                 '12/01/2008'];
Return = xirr(CashFlow, CashFlowDates)
```

This returns:

```
Return =
         0.1006 (or 10.0644% per annum)
```

Alternatively, you can use `datetime` input to calculate the internal rate of return for this nonperiodic cash flow:

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
CashFlowDates = ['01/12/2007'
```

```
                  '02/14/2008'
                  '03/03/2008'
                  '06/14/2008'
                  '12/01/2008'];
CashFlowDates = datetime(CashFlowDates,'Locale','en_US')';
Return = xirr(CashFlow, CashFlowDates)
```

This returns:

```
Return =
         0.1006 (or 10.0644% per annum)
```

References

Brealey and Myers. *Principles of Corporate Finance.* McGraw-Hill Higher Education, Chapter 5, 2003.

Sharpe, William F., and Gordon J. Alexander. *Investments.* Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990.

See Also

`datetime` | `fvvar` | `irr` | `mirr` | `pvvar`

Topics

“Analyzing and Computing Cash Flows” on page 2-21

Introduced before R2006a

year

Year of date

Syntax

```
Year = year(Date)
Year = year(___,F)
```

Description

`Year = year(Date)` returns the year of date given a serial date number or a date character vector.

`Year = year(___,F)` returns the year of date given a serial date number or a date character vector, `Date`, using format defined by the optional input `F`. `Date` can be a character array where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`.

Examples

Determine the Year of the Date for Various Dates

Find the year for `Date` using a serial date number.

```
Year = year(731798.776)
```

```
Year = 2003
```

Find the year for `Date` using a date character vector format.

```
Year = year('05-Aug-2003')
```

```
Year = 2003
```

Use the optional `F` argument to designate a country-specific date format for a given `Date`.

```
Year = year('1999/05/09','yyyy/dd/mm')
```

```
Year = 1999
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Date` — Date to determine year

serial date number | date character vector | cell array of date character vectors

Date to determine year, specified as a serial date number or date character vector.

`Date` can be an array of date character vectors, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. All the character vectors in `Date` must have the same format `F`. `F` must designate a supported date format symbol. For more information on supported date formats, see `datestr`

Data Types: `single` | `double` | `char` | `cell`

`F` — Date format symbol

character vector designating date format

Date format symbol, specified as a character vector to designate the date format symbol for input argument `Date`. For more information on supported date character vector formats, see `datestr`. Note, formats with `'Q'` are not accepted.

Data Types: `char`

Output Arguments

`Year` — Numeric representation of the year

nonnegative integer

Numeric representation of the year, returned as a nonnegative integer.

See Also

`datevec` | `day` | `month` | `yeardays`

Topics

“Handle and Convert Dates” on page 2-2

Introduced before R2006a

yeardays

Number of days in year

Syntax

```
Days = yeardays(Year)
Days = yeardays(___,Basis)
```

Description

`Days = yeardays(Year)` returns the number of days in the given `Year`.

`Days = yeardays(___,Basis)` returns the number of days in the given `Year`, based on the optional argument `Basis` for day-count.

Examples

Determine the Number of Days in a Given Year

Find the number of days in a given `Year`.

```
Days = yeardays(2000)
```

```
Days = 366
```

Find the number of days in a given `Year` using the optional argument `Basis`.

```
Days = yeardays(2000, 1)
```

```
Days = 360
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`Year` — Year to determine days

4 digit integer

Data Types: `single` | `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | vector of integers with values `0,1,2,3,4,5,6,7,8,9,10,11,12,13`

Day-count basis, specified as a vector of integers with values `0,1,2,3,4,5,6,7,8,9,10,11,12,13`.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

Output Arguments

`Days` — Number of days in given `Year`

nonnegative integer

Number of days in given `Year`, returned as a nonnegative integer.

See Also

`days360` | `days365` | `daysact` | `year` | `yearfrac`

Topics

"Handle and Convert Dates" on page 2-2

Introduced before R2006a

yearfrac

Fraction of year between dates

Syntax

```
YearFraction = yearfrac(StartDate,EndDate,Basis)
```

Description

`YearFraction = yearfrac(StartDate,EndDate,Basis)` returns a fraction, in years, based on the number of days between dates `StartDate` and `EndDate` using the given day-count `Basis`.

The number of days in a year (365 or 366) is equal to the number of days in the calendar year after the `StartDate`. If `EndDate` is earlier than `StartDate`, `YearFraction` is negative.

All specified arguments must be number of instruments (`NUMINST`-by-`1`) or (`1`-by-`NUMINST`) conforming vectors or scalar arguments.

Examples

Compute yearfrac When the Calendar Year After the StartDate is Not a Leap Year

Given a `Basis` of `0` and a `Basis` of `1`, compute `yearfrac`.

Define the `StartDate` and `EndDate` using a `Basis` of `0`.

```
YearFraction = yearfrac('14 mar 01', '14 sep 01', 0)
```

```
YearFraction = 0.5041
```

Define the `StartDate` and `EndDate` using a `Basis` of `1`.

```
YearFraction = yearfrac('14 mar 01', '14 sep 01', 1)

YearFraction = 0.5000
```

Compute yearfrac When the Calendar Year After the StartDate is a Leap Year

Given a `Basis` of `0`, compute `yearfrac` when the calendar after `StartDate` is in a leap year.

Define the `StartDate` and `EndDate` using a `Basis` of `0`.

```
yearFraction = yearfrac(' 14 mar 03', '14 sep 03', 0)

yearFraction = 0.5027
```

There are 184 days between March 14 and September 14, and the calendar year after the `StartDate` is a leap year, so `yearfrac` returns 184/366 = 0.5027.

Compute the Fraction of a Year Using an actual/actual Basis

To get the fraction of a year between '31-Jul-2015' and '30-Sep-2015' using the actual/actual basis:

```
yearfrac('31-Jul-2015', '30-Sep-2015', 0)*2


ans =

    0.3333
```

For the actual/actual basis, the fraction of a year is calculated as:

```
(Actual Days between Start Date and End Date)/(Actual Days between
Start Date and exactly one year after Start Date)
```

There are 61 days between 31-Jul-2015 and 30-Sep-2015. Since the next year is a leap year, there are 366 days between 31-Jul-2015 and 31-Jul-2016. So, there is 61/366 which

is exactly 1/6. So given this, exactly 2/6 is the expected result for the fraction of the six-month period.

Compute `yearfrac` When Specifying datetime Arrays

Given a `Basis` of `9`, compute `yearfrac` when the `StartDate` and `EndDate` are specified using datetime arrays.

```
yearfrac(datetime('1-Jan-2000','Locale','en_US'), '1/1/2001', 9)
```

```
ans = 1.0167
```

- “Handle and Convert Dates” on page 2-2

Input Arguments

`StartDate` — Start date

serial date number | date character vector | datetime object

Start date, specified as an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`EndDate` — End date

serial date number | date character vector | datetime object

End date, specified as an `N`-by-`1` or `1`-by-`N` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Basis` — Day-count basis for each set of dates

`0` (actual/actual) (default) | vector of numerics with values `0` through `13`

Day-count basis for each set of dates, specified as an `N`-by-`1` or `1`-by-`N` vector of integers with values of `0` through `13`.

- 0 = actual/actual (default)

- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `single` | `double`

Output Arguments

`YearFraction` — Real numbers identifying interval, in years, between `StartDate` and `EndDate`

vector

Real numbers identifying the interval, in years, between `StartDate` and `EndDate`, returned an `N`-by-`1` or `1`-by-`N` vector.

Definitions

Difference Between `yearfrac` and `date2time`

The difference between `yearfrac` and `date2time` is that `date2time` counts full periods as a whole integer, even if the number of actual days in the periods are different. `yearfrac` does not count full periods.

For example,

```
yearfrac('1/1/2000', '1/1/2001', 9)

ans =

    1.0167
```

`yearfrac` for `Basis 9` (ACT/360 ICMA) calculates 366/360 = 1.0167. So, even if the dates have the same month and date, with a difference of 1 in the year, the returned value may not be exactly 1. On the other hand, `date2time` calculates one full year period:

```
date2time('1/1/2000', '1/1/2001', 1, 9)

ans =

     1
```

See Also

`date2time` | `datetime` | `days360` | `days365` | `daysact` | `daysdif` | `months` | `wrkdydif` | `year` | `year` | `yeardays` | `yearfrac`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 15-2
"UICalendar User Interface" on page 15-4

Introduced before R2006a

yldddisc

Yield of discounted security

Syntax

```
Yield = ylddisc(Settle,Maturity,Face,Price)
Yield = ylddisc(___,Basis)
```

Description

`Yield = ylddisc(Settle,Maturity,Face,Price)` returns the yield of a discounted security.

`Yield = ylddisc(___,Basis)` adds optional an argument for `Basis`.

Examples

Find the Yield of a Discounted Security

This example shows how to find the yield of the following discounted security.

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Price = 96.28;
Basis = 2;

Yield = ylddisc(Settle, Maturity, Face, Price, Basis)

Yield = 0.0903
```

Find the Yield of a Discounted Security Using datetime Inputs

This example shows how to use `datetime` inputs to find the yield of the following discounted security.

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Price = 96.28;
Basis = 2;

Settle = datetime(Settle,'Locale','en_US');
Maturity = datetime(Maturity,'Locale','en_US');

Yield = ylddisc(Settle, Maturity, Face, Price, Basis)

Yield = 0.0903
```

- “Yield Functions” on page 2-35

Input Arguments

`Settle` — Settlement date of security

serial date number | date character vector | datetime

Settlement date of the security, specified as serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of security

serial date number | date character vector | datetime

Maturity date of the security, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value of security

numeric

Redemption value (par value) of the security, specified as a numeric value.

Data Types: `double`

`Price` — Discount price of security
numeric

Discount price of the security, specified as a numeric value.

Data Types: `double`

`Basis` — (Optional) Day-count basis
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

Day-count basis for the security, specified using the following values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`Yield` — Yield of discounted security

numeric

Yield of discounted security, returned as a numeric value.

References

[1] Mayle, J. *Standard Securities Calculation Methods*. Volumes I-II, 3rd edition. Formula 1.

See Also

`acrudisc` | `bndprice` | `bndyield` | `datetime` | `prdisc` | `yldmat` | `yldtbill`

Topics

“Yield Functions” on page 2-35
“Yield Conventions” on page 2-34

Introduced before R2006a

yldmat

Yield with interest at maturity

Syntax

```
Yield = yldmat(Settle,Maturity,Issue,Face,Price,CouponRate)
Yield = yldmat( ___ ,Basis)
```

Description

`Yield = yldmat(Settle,Maturity,Issue,Face,Price,CouponRate)` returns the yield of a security paying interest at maturity.

`Yield = yldmat( ___ ,Basis)` adds an optional argument for `Basis`.

Examples

Find the Yield of a Security Paying Interest at Maturity

This example shows how to find the yield of a security paying interest at maturity for the following.

```
Settle = '02/07/2000';
Maturity = '04/13/2000';
Issue = '10/11/1999';
Face = 100;
Price = 99.98;
CouponRate = 0.0608;
Basis = 1;

Yield = yldmat(Settle, Maturity, Issue, Face, Price,...
CouponRate, Basis)

Yield = 0.0607
```

Find the Yield of a Security Paying Interest at Maturity Using datetime Inputs

This example shows how to use `datetime` inputs find the yield of a security paying interest at maturity for the following:

```
Settle = '7-Feb-2000';
Maturity = '13-Apr-2000';
Issue = '11-Oct-1999';
Face = 100;
Price = 99.98;
CouponRate = 0.0608;
Basis = 1;

Settle = datetime(Settle,'Locale','en_US');
Maturity = datetime(Maturity,'Locale','en_US');
Issue = datetime(Issue,'Locale','en_US');

Yield = yldmat(Settle, Maturity, Issue, Face, Price,...
CouponRate, Basis)

Yield = 0.0607
```

- “Yield Functions” on page 2-35

Input Arguments

`Settle` — Settlement date of security

serial date number | date character vector | datetime

Settlement date of the security, specified as serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of security

serial date number | date character vector | datetime

Maturity date of the security, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Issue` — Issue date of security
serial date number | date character vector | datetime

Issue date of the security, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value of security
numeric

Redemption value (par value) of the security, specified as a numeric value.

Data Types: `double`

`Price` — Price of security
numeric

Price of the security, specified as a numeric value.

Data Types: `double`

`CouponRate` — Coupon rate of security
decimal fraction

Coupon rate of the security, specified as a decimal fraction.

Data Types: `double`

`Basis` — Day-count basis
`0` (actual/actual) (default) | integers of the set `[0...13]` | vector of integers of the set `[0...13]`

(Optional) Day-count basis for the security, specified using the following values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)

- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`Yield` — Yield of a security paying interest at maturity

numeric

Yield of a security paying interest at maturity, returned as a numeric value.

References

[1] Mayle, J. *Standard Securities Calculation Methods*. Volumes I-II, 3rd edition. Formula 3.

See Also

`acrudisc` | `bndprice` | `bndyield` | `datetime` | `prmat` | `ylddisc` | `yldtbill`

Topics

“Yield Functions” on page 2-35
“Yield Conventions” on page 2-34

Introduced before R2006a

yldtbill

Yield of Treasury bill

Syntax

```
Yield = yldtbill(Settle,Maturity,Face,Price)
```

Description

`Yield = yldtbill(Settle,Maturity,Face,Price)` returns the yield for a Treasury bill.

Examples

Find the Yield for a Treasury Bill

This example shows how to return the yield for a Treasury bill, given the settlement date of a Treasury bill is February 10, 2000, the maturity date is August 6, 2000, the par value is $1000, and the price is $981.36.

```
Yield = yldtbill('2/10/2000', '8/6/2000', 1000, 981.36)
```

```
Yield = 0.0384
```

Find the Yield for a Treasury Bill Using datetime Inputs

This example shows how to use `datetime` inputs to return the yield for a Treasury bill, given the settlement date of a Treasury bill is February 10, 2000, the maturity date is August 6, 2000, the par value is $1000, and the price is $981.36.

```
Yield = yldtbill(datetime('10-Feb-2000','Locale','en_US'), datetime('6-Aug-2000','Local
```

```
Yield = 0.0384
```

- “Computing Treasury Bill Price and Yield” on page 2-41
- “Yield Functions” on page 2-35

Input Arguments

`Settle` — Settlement date of Treasury bill

serial date number | date character vector | datetime

Settlement date of the Treasury bill, specified as serial date numbers, date character vectors, or datetime arrays. The `Settle` date must be before the `Maturity` date.

Data Types: `double` | `char` | `datetime`

`Maturity` — Maturity date of Treasury bill

serial date number | date character vector | datetime

Maturity date of the Treasury bill, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `char` | `datetime`

`Face` — Redemption value of Treasury bill

numeric

Redemption value (par value) of the Treasury bill, specified as a numeric value.

Data Types: `double`

`Price` — Price of Treasury bill

numeric

Price of the Treasury bill, specified as a numeric value.

Data Types: `double`

Output Arguments

`Yield` — Yield for Treasury bill
numeric

Yield for Treasury bill, returned as a numeric value.

References

[1] Bodie, Kane, and Marcus. *Investments*. McGraw-Hill Education, 2013.

See Also

`beytbill` | `bndyield` | `datetime` | `prtbill` | `yldmat`

Topics

“Computing Treasury Bill Price and Yield” on page 2-41
“Yield Functions” on page 2-35
“Treasury Bills Defined” on page 2-40
“Yield Conventions” on page 2-34

Introduced before R2006a

zbtprice

Zero curve bootstrapping from coupon bond data given price

Syntax

```
[ZeroRates,CurveDates] = zbtprice(Bonds,Prices,Settle)
ZeroRates,CurveDates = zbtprice(___,OutputCompounding)
```

Description

`[ZeroRates,CurveDates] = zbtprice(Bonds,Prices,Settle)` uses the bootstrap method to return a zero curve given a portfolio of coupon bonds and their prices.

A zero curve consists of the yields to maturity for a portfolio of theoretical zero-coupon bonds that are derived from the input `Bonds` portfolio. The bootstrap method that this function uses does *not* require alignment among the cash-flow dates of the bonds in the input portfolio. It uses theoretical par bond arbitrage and yield interpolation to derive all zero rates; specifically, the interest rates for cash flows are determined using linear interpolation. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

`ZeroRates,CurveDates = zbtprice(___,OutputCompounding)` adds an optional argument for `OutputCompounding`.

Examples

Compute a Zero Curve Given a Portfolio of Coupon Bonds and Their Prices

Given data and prices for 12 coupon bonds, two with the same maturity date, and given the common settlement date.

```
Bonds = [datenum('6/1/1998')   0.0475   100  2  0  0;
         datenum('7/1/2000')   0.06     100  2  0  0;
```

```
          datenum('7/1/2000')   0.09375  100  6  1  0;
          datenum('6/30/2001')  0.05125  100  1  3  1;
          datenum('4/15/2002')  0.07125  100  4  1  0;
          datenum('1/15/2000')  0.065    100  2  0  0;
          datenum('9/1/1999')   0.08     100  3  3  0;
          datenum('4/30/2001')  0.05875  100  2  0  0;
          datenum('11/15/1999') 0.07125  100  2  0  0;
          datenum('6/30/2000')  0.07     100  2  3  1;
          datenum('7/1/2001')   0.0525   100  2  3  0;
          datenum('4/30/2002')  0.07     100  2  0  0];

Prices = [99.375;
           99.875;
          105.75 ;
           96.875;
          103.625;
          101.125;
          103.125;
           99.375;
          101.0  ;
          101.25 ;
           96.375;
          102.75 ];

Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve.

```
OutputCompounding = 2;
```

Execute the function `zbtprice` which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date.

```
[ZeroRates, CurveDates] = zbtprice(Bonds, Prices, Settle,...
OutputCompounding)

ZeroRates =

    0.0616
    0.0609
    0.0658
    0.0590
    0.0647
    0.0655
    0.0606
```

```
    0.0601
    0.0642
    0.0621


CurveDates =

      729907
      730364
      730439
      730500
      730667
      730668
      730971
      731032
      731033
      731321
```

Compute a Zero Curve Given a Portfolio of Coupon Bonds and Their Prices Using datetime Inputs

Given data and prices for 12 coupon bonds, two with the same maturity date, and given the common settlement date, use `datetime` inputs to compute a zero curve.

```
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');
OutputCompounding = 2;

t=array2table(Bonds);
t.Bonds1=datetime(t.Bonds1,'ConvertFrom','datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
[ZeroRates, CurveDates] = zbtprice(t, Prices, Settle,...
OutputCompounding)

ZeroRates =

    0.0616
    0.0609
    0.0658
    0.0590
    0.0647
    0.0655
    0.0606
    0.0601
    0.0642
    0.0621


CurveDates = 11x1 datetime array
   01-Jun-1998 00:00:00
   01-Sep-1999 00:00:00
   15-Nov-1999 00:00:00
   15-Jan-2000 00:00:00
   30-Jun-2000 00:00:00
   01-Jul-2000 00:00:00
   30-Apr-2001 00:00:00
   30-Jun-2001 00:00:00
   01-Jul-2001 00:00:00
   15-Apr-2002 00:00:00
   30-Apr-2002 00:00:00
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`Bonds` — Coupon bond information to generate zero curve

table | matrix

Coupon bond information to generate zero curve, specified as a 6-column table or a `n`-by-`2` to `n`-by-`6` matrix of bond information, where the table columns or matrix columns contains:

- `Maturity` (Column 1, Required) Maturity date of the bond, as a serial date number. Use `datenum` to convert date character vectors to serial date numbers. If the input `Bonds` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- `CouponRate` (Column 2, Required) Decimal fraction indicating the coupon rate of the bond.
- `Face` (Column 3, Optional) Redemption or face value of the bond. Default = `100`.
- `Period` (Column 4, Optional) Coupons per year of the bond. Allowed values are `0`, `1`, `2` (default), `3`, `4`, `6`, and `12`.
- `Basis` (Column 5, Optional) Day-count basis of the bond. A vector of integers.
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252
- For more information, see **basis** on page Glossary-0 .

- `EndMonthRule` (Column 6, 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

:

Note

- If `Bonds` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- If `Bonds` is a matrix, is an `n`-by-`2` to `n`-by-`6` matrix where each row describes a bond, the first two columns (`Maturity` and `CouponRate`) are required. The remainder of the columns are optional but must be added in order. All rows in `Bonds` must have the same number of columns.

.

Data Types: `double` | `table`

`Prices` — Clean price (price without accrued interest) of each bond in `Bonds`
numeric

Clean price (price without accrued interest) of each bond in `Bonds`, specified as a `N`-by-`1` column vector. The number of rows (*n*) must match the number of rows in `Bonds`.

Data Types: `double`

`Settle` — Settlement date representing time zero in derivation of zero curve
serial date number | date character vector | datetime

Settlement date representing time zero in derivation of zero curve, specified as serial date number, date character vector, or datetime array. `Settle` represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.

Data Types: `double` | `char` | `datetime`

`OutputCompounding` — Compounding frequency of output `ZeroRates`
`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

(Optional) Compounding frequency of output `ZeroRates`, specified using the allowed values:

- `0` — Simple interest (no compounding)
- `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

Data Types: `double`

Output Arguments

`ZeroRates` — Implied zero rates for each point along the investment horizon defined by maturity date
decimal fractions

Implied zero rates for each point along the investment horizon defined by a maturity date, returned as a `m`-by-`1` vector of decimal fractions where `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`. Any rates before the first `Maturity` are assumed to be equal to the rate at the first `Maturity`, that is, the curve is assumed to be flat before the first `Maturity`.

`CurveDates` — Maturity dates that correspond to `ZeroRates`
serial date number | date character vector | datetime

Maturity dates that correspond to the `ZeroRates`, returned as a `m`-by-`1` vector of unique maturity dates, where `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 in the `Bonds` table or matrix.

If either inputs for `Bonds` or `Settle` have datetime values, then `CurveDates` is datetimes. Otherwise `CurveDates` is serial date numbers.

References

[1] 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.

[2] McEnally, Richard W. and James V. Jordan. "The Term Structure of Interest Rates." in Ch. 37 in Fabozzi and Fabozzi, *ibid*

[3] Das, Satyajit. "Calculating Zero Coupon Rates." in *Swap and Derivative Financing.* Appendix to Ch. 8, pp. 219–225. New York, Irwin Professional Publishing, 1994.

See Also

`datetime` | `zbtyield`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

zbtyield

Zero curve bootstrapping from coupon bond data given yield

Syntax

```
[ZeroRates,CurveDates] = zbtyield(Bonds,YieldsSettle)
ZeroRates,CurveDates = zbtyield(___,OutputCompounding)
```

Description

`[ZeroRates,CurveDates] = zbtyield(Bonds,YieldsSettle)` uses the bootstrap method to return a zero curve given a portfolio of coupon bonds and their yields.

A zero curve consists of the yields to maturity for a portfolio of theoretical zero-coupon bonds that are derived from the input `Bonds` portfolio. The bootstrap method that this function uses does *not* require alignment among the cash-flow dates of the bonds in the input portfolio. It uses theoretical par bond arbitrage and yield interpolation to derive all zero rates; specifically, the interest rates for cash flows are determined using linear interpolation. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

`ZeroRates,CurveDates = zbtyield(___,OutputCompounding)` adds an optional argument for `OutputCompounding`.

Examples

Compute a Zero Curve Given a Portfolio of Coupon Bonds and Their Yields

Given data and yields to maturity for 12 coupon bonds, two with the same maturity date; and given the common settlement date.

```
Bonds = [datenum('6/1/1998')   0.0475   100  2  0  0;
         datenum('7/1/2000')   0.06     100  2  0  0;
```

```
          datenum('7/1/2000')    0.09375  100   6   1   0;
          datenum('6/30/2001')   0.05125  100   1   3   1;
          datenum('4/15/2002')   0.07125  100   4   1   0;
          datenum('1/15/2000')   0.065    100   2   0   0;
          datenum('9/1/1999')    0.08     100   3   3   0;
          datenum('4/30/2001')   0.05875  100   2   0   0;
          datenum('11/15/1999') 0.07125  100   2   0   0;
          datenum('6/30/2000')   0.07     100   2   3   1;
          datenum('7/1/2001')    0.0525   100   2   3   0;
          datenum('4/30/2002')   0.07     100   2   0   0];

Yields = [0.0616
          0.0605
          0.0687
          0.0612
          0.0615
          0.0591
          0.0603
          0.0608
          0.0655
          0.0646
          0.0641
          0.0627];

Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve.

```
OutputCompounding = 2;
```

Execute the function `zbtyield` which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date.

```
[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle,...
OutputCompounding)

ZeroRates =

    0.0616
    0.0603
    0.0657
    0.0590
    0.0649
    0.0650
    0.0606
```

```
    0.0611
    0.0643
    0.0614


CurveDates =

      729907
      730364
      730439
      730500
      730667
      730668
      730971
      731032
      731033
      731321
```

Compute a Zero Curve Given a Portfolio of Coupon Bonds and Their Yields Using datetime Inputs

Given data and yields to maturity for 12 coupon bonds, two with the same maturity date; and given the common settlement date, compute the zero curve using `datetime` inputs.

```
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');
OutputCompounding = 2;
t = array2table(Bonds);
t.Bonds1 = datetime(t.Bonds1,'ConvertFrom','datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
[ZeroRates, CurveDates] = zbtyield(t, Yields, Settle,...
OutputCompounding)

ZeroRates =

    0.0616
    0.0603
    0.0657
    0.0590
    0.0649
    0.0650
    0.0606
    0.0611
    0.0643
    0.0614


CurveDates = 11x1 datetime array
   01-Jun-1998 00:00:00
   01-Sep-1999 00:00:00
   15-Nov-1999 00:00:00
   15-Jan-2000 00:00:00
   30-Jun-2000 00:00:00
   01-Jul-2000 00:00:00
   30-Apr-2001 00:00:00
   30-Jun-2001 00:00:00
   01-Jul-2001 00:00:00
   15-Apr-2002 00:00:00
   30-Apr-2002 00:00:00
```

- "Term Structure of Interest Rates" on page 2-45

Input Arguments

`Bonds` — Coupon bond information to generate zero curve

table | matrix

Coupon bond information to generate zero curve, specified as a 6-column table or a `n`-by-`2` to `n`-by-`6` matrix of bond information, where the table columns or matrix columns contains:

- `Maturity` (Column 1, Required) Maturity date of the bond, as a serial date number. Use `datenum` to convert date character vectors to serial date numbers. If the input `Bonds` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- `CouponRate` (Column 2, Required) Decimal fraction indicating the coupon rate of the bond.
- `Face` (Column 3, Optional) Redemption or face value of the bond. Default = `100`.
- `Period` (Column 4, Optional) Coupons per year of the bond. Allowed values are `0`, `1`, `2` (default), `3`, `4`, `6`, and `12`.
- `Basis` (Column 5, Optional) Day-count basis of the bond. A vector of integers.
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252
- For more information, see **basis** on page Glossary-0 .

- `EndMonthRule` (Column 6, 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

:

Note

- If `Bonds` is a table, the `Maturity` dates can be serial date numbers, date character vectors, or datetime arrays.
- If `Bonds` is a matrix, is an `n`-by-`2` to `n`-by-`6` matrix where each row describes a bond, the first two columns (`Maturity` and `CouponRate`) are required. The remainder of the columns are optional but must be added in order. All rows in `Bonds` must have the same number of columns.

.

Data Types: `double` | `table`

`Yields` — Yield to maturity of each bond in `Bonds`

numeric

Yield to maturity of each bond in `Bonds`, specified as a `N`-by-`1` column vector. The number of rows (*n*) must match the number of rows in `Bonds`.

Note Yield to maturity must be compounded semiannually.

Data Types: `double`

`Settle` — Settlement date representing time zero in derivation of zero curve
serial date number | date character vector | datetime

Settlement date representing time zero in derivation of zero curve, specified as serial date number, date character vector, or datetime array. `Settle` represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.

Data Types: `double` | `char` | `datetime`

`OutputCompounding` — Compounding frequency of output `ZeroRates`
`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

(Optional) Compounding frequency of output `ZeroRates`, specified using the allowed values:

- `0` — Simple interest (no compounding)
- `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

Data Types: `double`

Output Arguments

`ZeroRates` — Implied zero rates for each point along the investment horizon defined by maturity date
decimal fractions

Implied zero rates for each point along the investment horizon defined by a maturity date, returned as a `m`-by-`1` vector of decimal fractions where `m` is the number of bonds with unique 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`. Any rates before the first `Maturity` are assumed to be equal to the rate at the first `Maturity`, that is, the curve is assumed to be flat before the first `Maturity`.

`CurveDates` — Maturity dates that correspond to `ZeroRates`
serial date number | date character vector | datetime

Maturity dates that correspond to the `ZeroRates`, returned as a `m`-by-`1` vector of unique maturity dates, where `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 in the `Bonds` table or matrix.

If either inputs for `Bonds` or `Settle` have datetime values, then `CurveDatesCurveDates` is datetimes. Otherwise `CurveDates` is serial date numbers.

References

[1] 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.

[2] McEnally, Richard W. and James V. Jordan. "The Term Structure of Interest Rates." in Ch. 37 in Fabozzi and Fabozzi, *ibid*

[3] Das, Satyajit. "Calculating Zero Coupon Rates." in *Swap and Derivative Financing.* Appendix to Ch. 8, pp. 219–225. New York, Irwin Professional Publishing, 1994.

See Also

`datetime` | `zbtprice`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

zero2disc

Discount curve given zero curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the optional name-value pair inputs: `Compounding` and `Basis`.

Syntax

```
[DiscRates,CurveDates] = zero2disc(ZeroRates,CurveDates,Settle)
[DiscRates,CurveDates] = zero2disc(___,Name,Value)
```

Description

`[DiscRates,CurveDates] = zero2disc(ZeroRates,CurveDates,Settle)` returns a discount curve given a zero curve and its maturity dates. If either inputs for or are`CurveDatesSettle` a datetime array, `CurveDates` is returned as a datetime array. Otherwise, `CurveDates` is returned as a serial date number. The `DiscRates` output is the same for any of these input data types.

`[DiscRates,CurveDates] = zero2disc(___,Name,Value)` adds optional name-value pair arguments

Examples

Compute a Discount Curve Given a Zero Curve and Maturity Dates

Given a zero curve over a set of maturity dates and a settlement date.

```
ZeroRates = [0.0464
             0.0509
             0.0524
             0.0525
```

```
              0.0531
              0.0525
              0.0530
              0.0531
              0.0549
              0.0536];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
```

The zero curve is compounded daily on an actual/365 basis.

```
Compounding = 365;
Basis = 3;
```

Execute the function `zero2disc` which returns the discount curve `DiscRates` at the maturity dates `CurveDates`.

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates,...
Settle, Compounding, Basis)

DiscRates =

    0.9996
    0.9947
    0.9896
    0.9866
    0.9826
    0.9787
    0.9745
    0.9665
    0.9552
    0.9466
```

```
CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `ZeroRates` and `DiscRates` are shown here only to the basis point. However, MATLAB® software computed them at full precision. If you enter `ZeroRates` as shown, `DiscRates` may differ due to rounding.

Compute a Discount Curve Given a Zero Curve and Maturity Dates Using datetime Inputs

Given a zero curve over a set of maturity dates and a settlement date, compute a discount curve using `datetime` inputs.

```
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');
Compounding = 365;
Basis = 3;

CurveDates = datetime(CurveDates, 'ConvertFrom', 'datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates,...
Settle, Compounding, Basis)

DiscRates =

    0.9996
    0.9947
    0.9896
    0.9866
    0.9826
    0.9787
    0.9745
    0.9665
    0.9552
    0.9466


CurveDates = 10x1 datetime array
   06-Nov-2000 00:00:00
   11-Dec-2000 00:00:00
   15-Jan-2001 00:00:00
   05-Feb-2001 00:00:00
   04-Mar-2001 00:00:00
   02-Apr-2001 00:00:00
   30-Apr-2001 00:00:00
   25-Jun-2001 00:00:00
   04-Sep-2001 00:00:00
   12-Nov-2001 00:00:00
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`ZeroRates` — Annualized zero rates

decimal fraction

Annualized zero rates, specified as a `NUMBONDS`-by-`1` vector using decimal fractions. In aggregate, the zero rates constitute an implied zero curve for the investment horizon represented by `CurveDates`.

Data Types: `double`

`CurveDates` — Maturity dates

serial date number | date character vector | datetime

Maturity dates that correspond to the input `ZeroRates`, specified as `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`Settle` — Common settlement date for `ZeroRates`

serial date number | date character vector | datetime

Common settlement date for `ZeroRates`, specified as serial date numbers, date character vectors, or datetime arrays. `Settle` is the settlement date for the bonds from which the zero curve was bootstrapped.

Data Types: `double` | `datetime` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[DiscRates,CurveDates] = zero2disc(ZeroRates,CurveDates,Settle,'Compounding',4,'Basis',6)`

`Compounding` — Rate at which input `ZeroRates`zero rates are compounded when annualized

`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Rate at which the input `ZeroRates` are compounded when annualized, specified as using allowed numeric values:

- `0` — Simple interest (no compounding)
- `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

Data Types: `double`

`Basis` — Day-count basis used for annualizing input `ZeroRates`

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day-count basis used for annualizing the input `ZeroRates`, specified using allowed numeric values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)

- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Data Types: `double`

Output Arguments

`DiscRates` — Discount factors

decimal

Discount factors, returned as a `NUMBONDS`-by-`1` vector of decimal fractions. In aggregate, the discount factors constitute a discount curve for the investment horizon represented by `CurveDates`.

`CurveDates` — Maturity dates that correspond to `DiscRates`

serial date number | date character vector | datetime

Maturity dates that correspond to the `DiscRates`, returned as a `NUMBONDS`-by-`1` vector of maturity dates that correspond to the discount factors. This vector is the same as the input vector `CurveDates`, but is sorted by ascending maturity.

If either inputs for `CurveDates` or `Settle` are a datetime array, `CurveDates` is returned as a datetime array. Otherwise, `CurveDates` is returned as a serial date number.

See Also

`datetime` | `disc2zero` | `fwd2zero` | `prbyzero` | `pyld2zero` | `zbtprice` | `zbtyield` | `zero2fwd` | `zero2pyld`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

zero2fwd

Forward curve given zero curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the new optional name-value pair inputs: `InputCompounding`, `InputBasis`, `OutputCompounding`, and `OutputBasis`.

Syntax

```
[ForwardRates,CurveDates] = zero2fwd(ZeroRates,CurveDates,Settle)
[ForwardRates,CurveDates] = zero2fwd(___,Name,Value)
```

Description

`[ForwardRates,CurveDates] = zero2fwd(ZeroRates,CurveDates,Settle)` returns an implied forward rate curve given a zero curve and its maturity dates. If either input for `CurveDates` or `Settle` is a datetime array, `CurveDates` is returned as a datetime array. Otherwise, `CurveDates` is returned as a serial date number. `ForwardRates` is the same for any of these input data types.

`[ForwardRates,CurveDates] = zero2fwd(___,Name,Value)` adds optional name-value pair arguments

Examples

Compute an Implied Forward Rate Curve Given a Zero Curve and Maturity Dates

Given a zero curve over a set of maturity dates, a settlement date, and a compounding rate, compute the forward rate curve.

```
ZeroRates = [0.0458
             0.0502
```

```
              0.0518
              0.0519
              0.0524
              0.0519
              0.0523
              0.0525
              0.0541
              0.0529];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
InputCompounding = 1;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;
```

Execute the function `zero2fwd` to return the forward rate curve `ForwardRates` at the maturity dates `CurveDates`.

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,...
Settle, 'InputCompounding',1,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)
```

```
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
```

Compute an Implied Forward Rate Curve Given a Zero Curve and Maturity Dates Using datetime Inputs

Given a zero curve over a set of maturity dates, a settlement date, and a compounding rate, use `datetime` 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');
InputCompounding = 1;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;

CurveDates = datetime(CurveDates, 'ConvertFrom', 'datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,...
Settle,'InputCompounding',1,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)

ForwardRates =

    0.0458
    0.0506
    0.0535
    0.0522
    0.0541
    0.0498
    0.0544
    0.0531
    0.0594
    0.0476


CurveDates = 10x1 datetime array
   06-Nov-2000 00:00:00
   11-Dec-2000 00:00:00
   15-Jan-2001 00:00:00
   05-Feb-2001 00:00:00
   04-Mar-2001 00:00:00
   02-Apr-2001 00:00:00
   30-Apr-2001 00:00:00
   25-Jun-2001 00:00:00
   04-Sep-2001 00:00:00
   12-Nov-2001 00:00:00
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`ZeroRates` — Annualized zero rates

decimal fraction

Annualized zero rates, specified as a `NUMBONDS`-by-`1` vector using 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.

Data Types: `double`

`CurveDates` — Maturity dates

serial date number | date character vector | datetime

Maturity dates, specified as a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays, that correspond to the `ZeroRates`.

Data Types: `double` | `datetime` | `char`

`Settle` — Common settlement date for `ZeroRates`

serial date number | date character vector | datetime

Common settlement date for input `ZeroRates`, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
[ForwardRates,CurveDates] =
zero2fwd(ZeroRates,CurveDates,Settle,'InputCompounding',
3,'InputBasis',5,'OutputCompounding',4,'OutputBasis',5)
```

`InputCompounding` — Compounding frequency of input zero rates

`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of input zero rates, specified using allowed values:

- `0` — Simple interest (no compounding)
- `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

Note If `InputCompounding` is not specified, then `InputCompounding` is assigned the value specified for `OutputCompounding`. If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`InputBasis` — Day-count basis of input zero rates

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of input zero rates, specified using allowed values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `InputBasis` is not specified, then `InputBasis` is assigned the value specified for `OutputBasis`. If either `InputBasis` or `Outputbasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

`OutputCompounding` — Compounding frequency of output forward rates
`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of output forward rates, specified using the allowed values:

- `0` — Simple interest (no compounding)
- `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

Note If `OutputCompounding` is not specified, then `OutputCompounding` is assigned the value specified for `InputCompounding`. If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`OutputBasis` — Day-count basis of output forward rates
`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of output forward rates, specified using allowed values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `OutputBasis` is not specified, then `OutputBasis` is assigned the value specified for `InputBasis`. If either `InputBasis` or `OutputBasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

Output Arguments

`ForwardRates` — Forward curve for investment horizon represented by `CurveDates`
numeric

Forward curve for the investment horizon represented by `CurveDates`, returned as a `NUMBONDS`-by-`1` vector of decimal fractions. In aggregate, the rates in `ForwardRates`

constitute a forward curve over the dates in `CurveDates`. `ForwardRates` are ordered by ascending maturity.

`CurveDates` — Maturity dates that correspond to `ForwardRates`
serial date number | date character vector | datetime

Maturity dates that correspond to the `ForwardRates`, returned as a `NUMBONDS`-by-`1` vector of maturity dates that correspond to the `ForwardRates`.

`ForwardRates` are expressed as serial date numbers (default) or datetimes (if `CurveDates` or `Settle` are datetime arrays), representing the maturity dates for each rate in `ForwardRates`. These dates are the same dates as those associated with the input `ZeroRates`, but are ordered by ascending maturity.

See Also

`datetime` | `disc2zero` | `fwd2zero` | `getForwardRates` | `pyld2zero` | `zbtprice` | `zbtyield` | `zero2disc` | `zero2pyld`

Topics

“Term Structure of Interest Rates” on page 2-45
“Fixed-Income Terminology” on page 2-25

Introduced before R2006a

zero2pyld

Par yield curve given zero curve

Note In R2017b, the specification of optional input arguments has changed. While the previous ordered inputs syntax is still supported, it may no longer be supported in a future release. Use the new optional name-value pair inputs: `InputCompounding`, `InputBasis`, `OutputCompounding`, and `OutputBasis`.

Syntax

```
[ParRates,CurveDates] = zero2pyld(ZeroRates,CurveDates,Settle)
[ParRates,CurveDates] = zero2pyld(___,Name,Value)
```

Description

`[ParRates,CurveDates] = zero2pyld(ZeroRates,CurveDates,Settle)` returns a par yield curve given a zero curve and its maturity dates. If either input for `CurveDates` or `Settle` is a datetime array, `CurveDates` is returned as a datetime array. Otherwise, `CurveDates` is returned as a serial date number. `ParRates` is the same for any of these input data types.

`[ParRates,CurveDates] = zero2pyld(___,Name,Value)` adds optional name-value pair arguments

Examples

Compute Par Yield Curve Given a Zero Curve and Maturity Dates

Given a zero curve over a set of maturity dates, a settlement date, and annual compounding for the input zero curve and monthly compounding for the output par rates, compute a par yield curve.

```
ZeroRates = [0.0457
             0.0487
             0.0506
             0.0507
             0.0505
             0.0504
             0.0506
             0.0516
             0.0539
             0.0530];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
InputCompounding = 12;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;

[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates,...
Settle, 'InputCompounding',1,'InputBasis',1,'OutputCompounding',12,'OutputBasis',1)

ParRates =

    0.0448
    0.0477
    0.0495
    0.0496
    0.0494
    0.0493
    0.0495
    0.0504
    0.0526
    0.0517
```

```
CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

Compute Par Yield Curve Given a Zero Curve and Maturity Dates Using datetime Inputs

Given a zero curve over a set of maturity dates, a settlement date, and annual compounding for the input zero curve and monthly compounding for the output par rates, use `datetime` inputs to compute a par yield curve.

```
ZeroRates = [0.0457
0.0487
0.0506
0.0507
0.0505
0.0504
0.0506
0.0516
0.0539
0.0530];
CurveDates = [datenum('06-Nov-2000')
datenum('11-Dec-2000')
datenum('15-Jan-2001')
datenum('05-Feb-2001')
datenum('04-Mar-2001')
datenum('02-Apr-2001')
datenum('30-Apr-2001')
datenum('25-Jun-2001')
datenum('04-Sep-2001')
datenum('12-Nov-2001')];
Settle = datenum('03-Nov-2000');
InputCompounding = 12;
```

```
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;

CurveDates = datetime(CurveDates, 'ConvertFrom', 'datenum','Locale','en_US');
Settle = datetime(Settle,'ConvertFrom','datenum','Locale','en_US');
[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates,...
Settle, 'InputCompounding',12,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)

ParRates =

   -0.0436
    0.0611
    0.0579
    0.0567
    0.0550
    0.0543
    0.0541
    0.0546
    0.0565
    0.0561


CurveDates = 10x1 datetime array
   06-Nov-2000 00:00:00
   11-Dec-2000 00:00:00
   15-Jan-2001 00:00:00
   05-Feb-2001 00:00:00
   04-Mar-2001 00:00:00
   02-Apr-2001 00:00:00
   30-Apr-2001 00:00:00
   25-Jun-2001 00:00:00
   04-Sep-2001 00:00:00
   12-Nov-2001 00:00:00
```

Demonstrate a Roundtrip From `zero2pyld` to `pyld2zero`

Given the following zero curve and its maturity dates, return the `ParRates`.

```
Settle = datenum('01-Feb-2013');
```

```
CurveDates = [datenum('01-Feb-2014')
    datenum('01-Feb-2015')
    datenum('01-Feb-2016')
    datenum('01-Feb-2018')
    datenum('01-Feb-2020')
    datenum('01-Feb-2023')
    datenum('01-Feb-2033')
    datenum('01-Feb-2043')];

OriginalZeroRates = [.11 0.30 0.64 1.44 2.07 2.61 3.29 3.55]'/100;

OutputCompounding = 1;
OutputBasis = 0;
InputCompounding = 1;
InputBasis = 0;

ParRates = zero2pyld(OriginalZeroRates, CurveDates, Settle, ...
'OutputCompounding', OutputCompounding, 'OutputBasis', OutputBasis, ...
'InputCompounding', InputCompounding, 'InputBasis', InputBasis)

ParRates =

    0.0011
    0.0030
    0.0064
    0.0142
    0.0202
    0.0251
    0.0310
    0.0331
```

For the `ParRates`, use the `pyld2zero` function to return the `ZeroRatesOut` and determine the roundtrip error.

```
ZeroRatesOut = pyld2zero(ParRates, CurveDates, Settle, ...
'OutputCompounding', OutputCompounding, 'OutputBasis', OutputBasis, ...
'InputCompounding', InputCompounding, 'InputBasis', InputBasis)

ZeroRatesOut =

    0.0011
    0.0030
    0.0064
    0.0144
```

```
    0.0207
    0.0261
    0.0329
    0.0355
```

```
max(abs(OriginalZeroRates - ZeroRatesOut)) % Roundtrip error

ans = 1.4919e-16
```

- “Term Structure of Interest Rates” on page 2-45

Input Arguments

`ZeroRates` — Annualized zero rates

decimal fraction

Annualized zero rates, specified as a `NUMBONDS`-by-`1` vector using decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by `CurveDates`.

Data Types: `double`

`CurveDates` — Maturity dates

serial date number | date character vector | datetime

Maturity dates which correspond to the input `ZeroRates`, specified as a `NUMBONDS`-by-`1` vector using serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

`Settle` — Common settlement date for `ZeroRates`

serial date number | date character vector | datetime

Common settlement date for input `ZeroRates`, specified as serial date numbers, date character vectors, or datetime arrays.

Data Types: `double` | `datetime` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[ParRates,CurveDates] = zero2pyld(ZeroRates,CurveDates, Settle,'OutputCompounding',3,'OutputBasis',5,'InputCompounding', 4,'InputBasis',5)`

`OutputCompounding` — Compounding frequency of output `ParRates`

`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of output `ParRates`, specified using the allowed values:

- `1` — Annual compounding
- `2` — Semiannual compounding (default)
- `3` — Compounding three times per year
- `4` — Quarterly compounding
- `6` — Bimonthly compounding
- `12` — Monthly compounding

Note

- If `InputCompounding` is `1`, `2`, `3`, `4`, `6`, or `12` and `OutputCompounding` is not specified, the value of `InputCompounding` is used.
- If `InputCompounding` is `0` (simple), `-1` (continuous), or `365` (daily), a valid `OutputCompounding` value must also be specified.
- If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`OutputBasis` — Day-count basis of output `ParRates`

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of output `ParRates`, specified using allowed values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `OutputBasis` is not specified, then `OutputBasis` is assigned the value specified for `InputBasis`. If either `InputBasis` or `OutputBasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

`InputCompounding` — Compounding frequency of input `ZeroRates`

`2` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `12`, `365`, `-1`

Compounding frequency of input `ZeroRates`, specified using allowed values:

- `0` — Simple interest (no compounding)
- `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

Note

- If `InputCompounding` is set to `0` (simple), `-1` (continuous), or `365` (daily), the `OutputCompounding` must also be specified using a valid value.
- If `InputCompounding` is not specified, then `InputCompounding` is assigned the value specified for `OutputCompounding`.
- If either `InputCompounding` or `OutputCompounding` are not specified, the default is `2` (semiannual) for both.

Data Types: `double`

`InputBasis` — Day-count basis of input `ZeroRates`

`0` (default) | numeric values: `0`,`1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, `10`, `11`, `12`, `13`

Day count basis of the input `ZeroRates`, specified using allowed values:

- 0 = actual/actual
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (PSA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)

- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/365 (ISDA)
- 13 = BUS/252

For more information, see **basis** on page Glossary-0 .

Note If `InputBasis` is not specified, then `InputBasis` is assigned the value specified for `OutputBasis`. If either `InputBasis` or `Outputbasis` are not specified, the default is `0` (actual/actual) for both.

Data Types: `double`

Output Arguments

`ParRates` — Par bond coupon rates
numeric

Par bond coupon rates, returned as a `NUMBONDS`-by-`1` numeric vector. `ParRates` are ordered by ascending maturity.

`CurveDates` — Maturity dates that correspond to `ParRates`
serial date number | date character vector | datetime

Maturity dates that correspond to the `ParRates`, returned as a `NUMBONDS`-by-`1` vector of maturity dates that correspond to each par rate contained in `ParRates`.

`ParRates` are expressed as serial date numbers (default) or datetimes (if `CurveDates` or `Settle` are datetime arrays). `CurveDates` are ordered by ascending maturity.

See Also

`datetime` | `datetime` | `disc2zero` | `fwd2zero` | `getForwardRates` | `pyld2zero` | `zbtprice` | `zbtyield` | `zero2disc` | `zero2fwd`

Topics

"Term Structure of Interest Rates" on page 2-45
"Fixed-Income Terminology" on page 2-25

Introduced before R2006a

score

Compute credit scores for given data

Syntax

```
Scores = score(sc)
Scores = score(sc, data)

[Scores,Points] = score(sc)
[Scores,Points] = score(sc,data)
```

Description

`Scores = score(sc)` computes the credit scores for the `creditscorecard` object's training data. This data can be a "training" or a "live" dataset. If the `data` input argument is not explicitly provided, the `score` function determines scores for the existing `creditscorecard` object's data.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are reported separately or spread across predictors. Missing data translates into `NaN` values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

`Scores = score(sc, data)` computes the credit scores for the given input `data`. This data can be a "training" or a "live" dataset.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are reported separately or spread across predictors. Missing data translates into `NaN` values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

`[Scores,Points] = score(sc)` computes the credit scores and points for the given data. If the `data` input argument is not explicitly provided, the `score` function determines scores for the existing `creditscorecard` object's data.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are reported separately or spread across predictors. Missing data translates into `NaN` values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

`[Scores,Points] = score(sc,data)` computes the credit scores and points for the given input `data`. This data can be a “training” or a “live” dataset.

`formatpoints` supports multiple alternatives to modify the scaling of the scores and can also be used to control the rounding of points and scores, and whether the base points are reported separately or spread across predictors. Missing data translates into `NaN` values for the corresponding points, and therefore for the total score. Use `formatpoints` to modify the score behavior for rows with missing data.

Examples

Obtain Scores for Training Data

This example shows how to use `score` to obtain scores for the training data.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
```

```
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    _________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Score training data using the `score` function without an optional input for `data`. By default, it returns unscaled scores. For brevity, only the first ten scores are displayed.

```
Scores = score(sc);
disp(Scores(1:10))
```

```
    1.0968
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
   -0.0872
    0.9207
    1.0399
    0.8252
```

Scale scores and display both points and scores for each individual in the training data (for brevity, only the first ten rows are displayed). For other scaling methods, and other options for formatting points and scores, use the `formatpoints` function.

```
sc = formatpoints(sc,'WorstAndBestScores',[300 850]);
[Scores,Points] = score(sc);
disp(Scores(1:10))
```

```
  602.0394
  648.1988
  560.5569
  662.4189
  646.8109
  702.1398
  453.4572
  579.9475
  594.9064
  567.9533
```

```
disp(Points(1:10,:))
```

CustAge	ResStatus	EmpStatus	CustIncome	TmWBank	OtherCC	AMBalance
95.256	62.421	56.765	121.18	116.05	86.224	64.15
126.46	82.276	105.81	121.18	62.107	86.224	64.15
93.256	62.421	105.81	76.585	116.05	42.287	64.15
95.256	82.276	105.81	121.18	60.719	86.224	110.96
126.46	82.276	105.81	121.18	60.719	86.224	64.15
126.46	82.276	105.81	121.18	116.05	86.224	64.15
48.727	82.276	56.765	53.208	62.107	86.224	64.15
95.256	113.58	105.81	121.18	62.107	42.287	39.729
95.256	62.421	56.765	121.18	62.107	86.224	110.96
95.256	82.276	56.765	121.18	62.107	86.224	64.15

Score a New Dataset

This example shows how to use `score` to obtain scores for a new dataset (for example, a validation or a test dataset) using the optional `'data'` input in the `score` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in

`creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                    Estimate       SE       tStat       pValue
                    ________    ________    _______    __________

    (Intercept)     0.70239     0.064001    10.975     5.0538e-28
    CustAge         0.60833      0.24932      2.44       0.014687
    ResStatus         1.377      0.65272    2.1097       0.034888
    EmpStatus       0.88565        0.293    3.0227      0.0025055
    CustIncome      0.70164      0.21844    3.2121      0.0013179
    TmWBank          1.1074      0.23271    4.7589     1.9464e-06
    OtherCC          1.0883      0.52912    2.0569       0.039696
    AMBalance         1.045      0.32214    3.2439      0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

For the purpose of illustration, suppose that a few rows from the original data are our "new" data. Use the optional `data` input argument in the `score` function to obtain the scores for the `newdata`.

```
newdata = data(10:20,:);
Scores = score(sc,newdata)

Scores =

    0.8252
    0.6553
    1.2443
    0.9478
    0.5690
    1.6192
    0.4899
    0.3824
    0.2945
    1.4401
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`data` — Dataset to be scored

table

(Optional) Dataset to be scored, specified as a MATLAB table where each row corresponds to individual observations. The `data` must contain columns for each of the predictors in the `creditscorecard` object.

Output Arguments

`Scores` — Scores for each observation
vector

Scores for each observation, returned as a vector.

`Points` — Points per predictor for each observation
table

Points per predictor for each observation, returned as a table.

Algorithms

The score of an individual *i* is given by the formula

```
Score(i) = Shift + Slope*(b0 + b1*WOE1(i) + b2*WOE2(i)+ ... +bp*WOEp(i))
```

where *bj* is the coefficient of the *j*-th variable in the model, and WOE*j*(*i*) is the Weight of Evidence (WOE) value for the *i*-th individual corresponding to the *j*-th model variable. `Shift` and `Slope` are scaling constants that can be controlled with `formatpoints`.

If the data for individual *i* is in the *i*-th row of a given dataset, to compute a score, the data(*i*,*j*) is binned using existing binning maps, and converted into a corresponding Weight of Evidence value `WOE`*j*(*i*). Using the model coefficients, the unscaled score is computed as

```
s = b0 + b1*WOE1(i) + ... +bp*WOEp(i).
```

For simplicity, assume in the description above that the *j*-th variable in the model is the *j*-th column in the data input, although, in general, the order of variables in a given dataset does not have to match the order of variables in the model, and the dataset could have additional variables that are not used in the model.

The formatting options can be controlled using `formatpoints`.

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `setmodel` | `table` | `validatemodel`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57

Introduced in R2014b

formatpoints

Format scorecard points and scaling

Syntax

```
sc = formatpoints(sc,Name,Value)
```

Description

`sc = formatpoints(sc,Name,Value)` modifies the scorecard points and scaling using optional name-value pair arguments. For example, use optional name-value pair arguments to change the scaling of the scores or the rounding of the points.

Examples

Scale Points Using Worst and Best Scores

This example shows how to use `formatpoints` to scale by providing the `Worst` and `Best` score values. By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors           Bin           Points
    _____________    _______________    _________

    'CustAge'        '[-Inf,33)'          -0.15894
    'CustAge'        '[33,37)'            -0.14036
```

```
    'CustAge'       '[37,40)'          -0.060323
    'CustAge'       '[40,46)'           0.046408
    'CustAge'       '[46,48)'             0.21445
    'CustAge'       '[48,58)'             0.23039
    'CustAge'       '[58,Inf]'              0.479
    'ResStatus'     'Tenant'           -0.031252
    'ResStatus'     'Home Owner'          0.12696
    'ResStatus'     'Other'               0.37641
    'EmpStatus'     'Unknown'          -0.076317
    'EmpStatus'     'Employed'            0.31449
    'CustIncome'    '[-Inf,29000)'      -0.45716
    'CustIncome'    '[29000,33000)'     -0.10466
    'CustIncome'    '[33000,35000)'      0.052329
    'CustIncome'    '[35000,40000)'      0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

Scale by providing the `'Worst'` and `'Best'` score values. The range provided below is a common score range. Display the points information again to verify that they are now scaled and also display the scaled minimum and maximum scores.

```
sc = formatpoints(sc,'WorstAndBestScores',[300 850]);
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)

PointsInfo=30x3 table
     Predictors           Bin           Points
    _____________    _______________    _______

    'CustAge'       '[-Inf,33)'         46.396
    'CustAge'       '[33,37)'           48.727
    'CustAge'       '[37,40)'           58.772
    'CustAge'       '[40,46)'           72.167
    'CustAge'       '[46,48)'           93.256
    'CustAge'       '[48,58)'           95.256
    'CustAge'       '[58,Inf]'          126.46
    'ResStatus'     'Tenant'            62.421
    'ResStatus'     'Home Owner'        82.276
    'ResStatus'     'Other'             113.58
    'EmpStatus'     'Unknown'           56.765
    'EmpStatus'     'Employed'          105.81
    'CustIncome'    '[-Inf,29000)'      8.9706
    'CustIncome'    '[29000,33000)'     53.208
```

```
    'CustIncome'    '[33000,35000)'     72.91
    'CustIncome'    '[35000,40000)'    76.585

MinScore = 300

MaxScore = 850.0000
```

As expected, the values of `MinScore` and `MaxScore` correspond to the desired worst and best scores.

Scale Points Using Shift and Slope

This example shows how to use `formatpoints` to scale by providing the `Shift` and `Slope` values. By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points by the `formatpoints` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);

1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
```

```
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    _______    __________

    (Intercept)    0.70239     0.064001    10.975     5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors            Bin             Points
    _____________    _________________    __________

    'CustAge'        '[-Inf,33)'           -0.15894
    'CustAge'        '[33,37)'             -0.14036
    'CustAge'        '[37,40)'            -0.060323
    'CustAge'        '[40,46)'             0.046408
    'CustAge'        '[46,48)'              0.21445
    'CustAge'        '[48,58)'              0.23039
    'CustAge'        '[58,Inf]'               0.479
    'ResStatus'      'Tenant'             -0.031252
    'ResStatus'      'Home Owner'           0.12696
    'ResStatus'      'Other'                0.37641
    'EmpStatus'      'Unknown'            -0.076317
    'EmpStatus'      'Employed'             0.31449
```

```
    'CustIncome'    '[-Inf,29000)'      -0.45716
    'CustIncome'    '[29000,33000)'     -0.10466
    'CustIncome'    '[33000,35000)'     0.052329
    'CustIncome'    '[35000,40000)'     0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

Scale by providing the `'Shift'` and `'Slope'` values. In this example, there is an arbitrary choice of shift and slope. Display the points information again to verify that they are now scaled and also display the scaled minimum and maximum scores.

```
sc = formatpoints(sc,'ShiftAndSlope',[300 6]);
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)

PointsInfo=30x3 table
     Predictors            Bin           Points
    _____________    _________________    _______

    'CustAge'        '[-Inf,33)'          41.904
    'CustAge'        '[33,37)'            42.015
    'CustAge'        '[37,40)'            42.495
    'CustAge'        '[40,46)'            43.136
    'CustAge'        '[46,48)'            44.144
    'CustAge'        '[48,58)'            44.239
    'CustAge'        '[58,Inf]'           45.731
    'ResStatus'      'Tenant'              42.67
    'ResStatus'      'Home Owner'         43.619
    'ResStatus'      'Other'              45.116
    'EmpStatus'      'Unknown'            42.399
    'EmpStatus'      'Employed'           44.744
    'CustIncome'     '[-Inf,29000)'       40.114
    'CustIncome'     '[29000,33000)'      42.229
    'CustIncome'     '[33000,35000)'      43.171
    'CustIncome'     '[35000,40000)'      43.347


MinScore = 292.1401

MaxScore = 318.4355
```

Scale Points Using Points, Odds Levels, and PDO

This example shows how to use `formatpoints` to scale by providing the points, odds levels, and PDO (points to double the odds). By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points by the `formatpoints` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
```

```
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors           Bin            Points
    _____________    _______________    _________

    'CustAge'        '[-Inf,33)'          -0.15894
    'CustAge'        '[33,37)'            -0.14036
    'CustAge'        '[37,40)'           -0.060323
    'CustAge'        '[40,46)'            0.046408
    'CustAge'        '[46,48)'             0.21445
    'CustAge'        '[48,58)'             0.23039
    'CustAge'        '[58,Inf]'              0.479
    'ResStatus'      'Tenant'            -0.031252
    'ResStatus'      'Home Owner'          0.12696
    'ResStatus'      'Other'               0.37641
    'EmpStatus'      'Unknown'           -0.076317
    'EmpStatus'      'Employed'            0.31449
    'CustIncome'     '[-Inf,29000)'       -0.45716
    'CustIncome'     '[29000,33000)'      -0.10466
    'CustIncome'     '[33000,35000)'      0.052329
    'CustIncome'     '[35000,40000)'      0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

Scale by providing the points, odds levels, and PDO (points to double the odds). Suppose that you want a score of 500 points to have odds of 2 (twice as likely to be good than to be bad) and that the odds double every 50 points (so that 550 points would have odds of 4).

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors            Bin           Points
    _____________    _________________    ______

    'CustAge'        '[-Inf,33)'          52.821
    'CustAge'        '[33,37)'            54.161
    'CustAge'        '[37,40)'            59.934
    'CustAge'        '[40,46)'            67.633
    'CustAge'        '[46,48)'            79.755
    'CustAge'        '[48,58)'            80.905
    'CustAge'        '[58,Inf]'           98.838
    'ResStatus'      'Tenant'             62.031
    'ResStatus'      'Home Owner'         73.444
    'ResStatus'      'Other'              91.438
    'EmpStatus'      'Unknown'            58.781
    'EmpStatus'      'Employed'           86.971
    'CustIncome'     '[-Inf,29000)'       31.309
    'CustIncome'     '[29000,33000)'      56.736
    'CustIncome'     '[33000,35000)'       68.06
    'CustIncome'     '[35000,40000)'      70.173


MinScore = 355.5051

MaxScore = 671.6403
```

Report Base Points Separately

This example shows how to use `formatpoints` to separate the base points from the rest of the points assigned to each predictor variable. The `formatpoints` name-value pair argument `'BasePoints'` serves this purpose.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors           Bin           Points
```

```
    ____________    _______________    _________

    'CustAge'       '[-Inf,33)'         -0.15894
    'CustAge'       '[33,37)'           -0.14036
    'CustAge'       '[37,40)'          -0.060323
    'CustAge'       '[40,46)'           0.046408
    'CustAge'       '[46,48)'            0.21445
    'CustAge'       '[48,58)'            0.23039
    'CustAge'       '[58,Inf]'             0.479
    'ResStatus'     'Tenant'           -0.031252
    'ResStatus'     'Home Owner'         0.12696
    'ResStatus'     'Other'              0.37641
    'EmpStatus'     'Unknown'          -0.076317
    'EmpStatus'     'Employed'           0.31449
    'CustIncome'    '[-Inf,29000)'      -0.45716
    'CustIncome'    '[29000,33000)'     -0.10466
    'CustIncome'    '[33000,35000)'     0.052329
    'CustIncome'    '[35000,40000)'     0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

By setting the name-value pair argument `BasePoints` to true, the points information table reports the base points separately in the first row. The minimum and maximum possible scores are not affected by this option.

```
sc = formatpoints(sc,'BasePoints',true);
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)

PointsInfo=31x3 table
     Predictors           Bin           Points
    ____________    _______________    _________

    'BasePoints'    'BasePoints'         0.70239
    'CustAge'       '[-Inf,33)'         -0.25928
    'CustAge'       '[33,37)'           -0.24071
    'CustAge'       '[37,40)'           -0.16066
    'CustAge'       '[40,46)'          -0.053933
    'CustAge'       '[46,48)'            0.11411
    'CustAge'       '[48,58)'            0.13005
    'CustAge'       '[58,Inf]'           0.37866
    'ResStatus'     'Tenant'            -0.13159
    'ResStatus'     'Home Owner'         0.026616
```

```
    'ResStatus'     'Other'              0.27607
    'EmpStatus'     'Unknown'           -0.17666
    'EmpStatus'     'Employed'           0.21415
    'CustIncome'    '[-Inf,29000)'       -0.5575
    'CustIncome'    '[29000,33000)'       -0.205
    'CustIncome'    '[33000,35000)'    -0.048013

MinScore = -1.3100

MaxScore = 3.0726
```

Round Points

This example shows how to use `formatpoints` to round points. Rounding is usually applied after scaling, otherwise, if the points for a particular predictor are all in a small range, rounding could cause the rounded points for different bins to be the same. Also, rounding all the points may slightly change the minimum and maximum total points.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769
```

```
Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the fitting model and display the minimum and maximum possible unscaled scores.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors            Bin             Points
    _____________    _________________    _________

    'CustAge'        '[-Inf,33)'           -0.15894
    'CustAge'        '[33,37)'             -0.14036
    'CustAge'        '[37,40)'            -0.060323
    'CustAge'        '[40,46)'             0.046408
    'CustAge'        '[46,48)'              0.21445
    'CustAge'        '[48,58)'              0.23039
    'CustAge'        '[58,Inf]'               0.479
    'ResStatus'      'Tenant'             -0.031252
    'ResStatus'      'Home Owner'           0.12696
    'ResStatus'      'Other'                0.37641
    'EmpStatus'      'Unknown'            -0.076317
    'EmpStatus'      'Employed'             0.31449
    'CustIncome'     '[-Inf,29000)'       -0.45716
```

```
    'CustIncome'    '[29000,33000)'     -0.10466
    'CustIncome'    '[33000,35000)'     0.052329
    'CustIncome'    '[35000,40000)'     0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

Scale points, and display the points information. By default, no rounding is applied.

```
sc = formatpoints(sc,'WorstAndBestScores',[300 850]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors           Bin           Points
    _____________    _______________    ______

    'CustAge'        '[-Inf,33)'        46.396
    'CustAge'        '[33,37)'          48.727
    'CustAge'        '[37,40)'          58.772
    'CustAge'        '[40,46)'          72.167
    'CustAge'        '[46,48)'          93.256
    'CustAge'        '[48,58)'          95.256
    'CustAge'        '[58,Inf]'         126.46
    'ResStatus'      'Tenant'           62.421
    'ResStatus'      'Home Owner'       82.276
    'ResStatus'      'Other'            113.58
    'EmpStatus'      'Unknown'          56.765
    'EmpStatus'      'Employed'         105.81
    'CustIncome'     '[-Inf,29000)'     8.9706
    'CustIncome'     '[29000,33000)'    53.208
    'CustIncome'     '[33000,35000)'     72.91
    'CustIncome'     '[35000,40000)'    76.585
```

Use the name-value pair argument `Round` to apply rounding for all points and then display the points information again.

```
sc = formatpoints(sc,'Round','AllPoints');
PointsInfo = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors           Bin           Points
    _____________    _______________    ______
```

```
'CustAge'       '[-Inf,33)'         46
'CustAge'       '[33,37)'           49
'CustAge'       '[37,40)'           59
'CustAge'       '[40,46)'           72
'CustAge'       '[46,48)'           93
'CustAge'       '[48,58)'           95
'CustAge'       '[58,Inf]'         126
'ResStatus'     'Tenant'            62
'ResStatus'     'Home Owner'        82
'ResStatus'     'Other'            114
'EmpStatus'     'Unknown'           57
'EmpStatus'     'Employed'         106
'CustIncome'    '[-Inf,29000)'       9
'CustIncome'    '[29000,33000)'     53
'CustIncome'    '[33000,35000)'     73
'CustIncome'    '[35000,40000)'     77
```

Scores for Missing or Out-of-Range Data

This example shows how to use `formatpoints` to score missing or out-of-range data. When data is scored, some observations can be either missing (`NaN`, or `undefined`) or out of range. You will need to decide whether or not points are assigned to these cases. Use the name-value pair argument `Missing` to do so.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `creditscorecard` to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Suppose missing observations are added to the data that you want to score. Notice that by default, the points and score assigned to the missing value is `NaN`.

```
newdata = data(1:10,:);
newdata.CustAge(1) = NaN;
[Scores,Points] = score(sc,newdata)

Scores =

       NaN
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
```

```
   -0.0872
    0.9207
    1.0399
    0.8252


Points=10x7 table
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBala
    ________    _________    _________    __________    _________    ________    ______

         NaN    -0.031252    -0.076317      0.43693       0.39607     0.15842    -0.017
       0.479      0.12696      0.31449      0.43693      -0.033752    0.15842    -0.017
     0.21445    -0.031252      0.31449     0.081611       0.39607    -0.19168    -0.017
     0.23039      0.12696      0.31449      0.43693      -0.044811    0.15842      0.35
       0.479      0.12696      0.31449      0.43693      -0.044811    0.15842    -0.017
       0.479      0.12696      0.31449      0.43693       0.39607     0.15842    -0.017
    -0.14036      0.12696    -0.076317     -0.10466      -0.033752    0.15842    -0.017
     0.23039      0.37641      0.31449      0.43693      -0.033752    -0.19168    -0.21
     0.23039    -0.031252    -0.076317      0.43693      -0.033752    0.15842      0.35
     0.23039      0.12696    -0.076317      0.43693      -0.033752    0.15842    -0.017
```

Use the name-value pair argument `Missing` to replace `NaN` with zero.

```
sc = formatpoints(sc,'Missing','ZeroPoints');
[Scores,Points] = score(sc,newdata)

Scores =

    0.9667
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
   -0.0872
    0.9207
    1.0399
    0.8252


Points=10x7 table
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBala
    ________    _________    _________    __________    _________    ________    ______
```

```
    0.10034    -0.031252    -0.076317     0.43693       0.39607     0.15842    -0.017
      0.479      0.12696      0.31449     0.43693      -0.033752    0.15842    -0.017
    0.21445    -0.031252      0.31449    0.081611       0.39607    -0.19168    -0.017
    0.23039      0.12696      0.31449     0.43693      -0.044811    0.15842      0.35
      0.479      0.12696      0.31449     0.43693      -0.044811    0.15842    -0.017
      0.479      0.12696      0.31449     0.43693       0.39607     0.15842    -0.017
   -0.14036      0.12696    -0.076317    -0.10466      -0.033752    0.15842    -0.017
    0.23039      0.37641      0.31449     0.43693      -0.033752    -0.19168    -0.21
    0.23039    -0.031252    -0.076317     0.43693      -0.033752    0.15842      0.35
    0.23039      0.12696    -0.076317     0.43693      -0.033752    0.15842    -0.017
```

Use the name-value pair argument `Missing` to replace the missing value with the minimum points for the predictor that has the missing values, `'CustAge'`.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,newdata)
```

```
Scores =

    0.7074
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
   -0.0872
    0.9207
    1.0399
    0.8252


Points=10x7 table
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBala
    ________    _________    _________    __________    ________    ________    ______

    -0.15894    -0.031252    -0.076317     0.43693       0.39607     0.15842    -0.017
       0.479      0.12696      0.31449     0.43693      -0.033752    0.15842    -0.017
     0.21445    -0.031252      0.31449    0.081611       0.39607    -0.19168    -0.017
     0.23039      0.12696      0.31449     0.43693      -0.044811    0.15842      0.35
       0.479      0.12696      0.31449     0.43693      -0.044811    0.15842    -0.017
       0.479      0.12696      0.31449     0.43693       0.39607     0.15842    -0.017
    -0.14036      0.12696    -0.076317    -0.10466      -0.033752    0.15842    -0.017
     0.23039      0.37641      0.31449     0.43693      -0.033752    -0.19168    -0.21
     0.23039    -0.031252    -0.076317     0.43693      -0.033752    0.15842      0.35
```

```
    0.23039      0.12696    -0.076317     0.43693     -0.033752     0.15842    -0.017
```

Use the name-value pair argument `Missing` to replace the missing value with the maximum points for the predictor that has the missing values, `'CustAge'`.

```
sc = formatpoints(sc,'Missing','MaxPoints');
[Scores,Points] = score(sc,newdata)
```

```
Scores =

    1.3454
    1.4646
    0.7662
    1.5779
    1.4535
    1.8944
   -0.0872
    0.9207
    1.0399
    0.8252


Points=10x7 table
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBala
    ________    _________    _________    __________    _________    ________    ______

       0.479    -0.031252    -0.076317     0.43693        0.39607     0.15842    -0.017
       0.479      0.12696      0.31449     0.43693      -0.033752     0.15842    -0.017
     0.21445    -0.031252      0.31449    0.081611        0.39607    -0.19168    -0.017
     0.23039      0.12696      0.31449     0.43693      -0.044811     0.15842      0.35
       0.479      0.12696      0.31449     0.43693      -0.044811     0.15842    -0.017
       0.479      0.12696      0.31449     0.43693        0.39607     0.15842    -0.017
    -0.14036      0.12696    -0.076317    -0.10466      -0.033752     0.15842    -0.017
     0.23039      0.37641      0.31449     0.43693      -0.033752    -0.19168     -0.21
     0.23039    -0.031252    -0.076317     0.43693      -0.033752     0.15842      0.35
     0.23039      0.12696    -0.076317     0.43693      -0.033752     0.15842    -0.017
```

Verify that the minimum and maximum points assigned to the missing data correspond to the minimum and maximum points for `'CustAge'`. The points for `'CustAge'` are reported in the first five rows of the points information table.

```
PointsInfo = displaypoints(sc);
PointsInfo(1:5,:)
```

```
ans=5x3 table
    Predictors         Bin          Points
    __________    ___________    _________

    'CustAge'     '[-Inf,33)'     -0.15894
    'CustAge'     '[33,37)'       -0.14036
    'CustAge'     '[37,40)'      -0.060323
    'CustAge'     '[40,46)'       0.046408
    'CustAge'     '[46,48)'        0.21445
```

```
min(PointsInfo.Points(1:5))
```

```
ans = -0.1589
```

```
max(PointsInfo.Points(1:5))
```

```
ans = 0.2145
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sc = formatpoints(sc,'BasePoints',true,'Round','AllPoints','WorstAndBestScores',[100, 700])`

Note `ShiftAndSlope`, `PointsOddsAndPDO`, and `WorstAndBestScores` are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (`BasePoints`, `Missing`, and `Round`) are not scaling methods and can be used together or with any one of the three scaling methods.

`BasePoints` — Indicator for separating base points

`false` (default) | logical scalar

Indicator for separating base points, specified as a logical scalar. If `true`, the scorecard explicitly separates base points. If `false`, the base points are spread across all variables in the `creditscorecard` object.

Data Types: `char`

`Missing` — Indicator for points assigned to missing or out-of-range information when scoring

`NoScore` (default) | character vector with values `NoScore`, `ZeroWOE`, `MinPoints`, and `MaxPoints`

Indicator for points assigned to missing or out-of-range information when scoring, specified as a character vector with a value for `NoScore`, `ZeroPoints`, `MinPoints`, or `MaxPoints`, where:

- `NoScore` — Missing and out-of-range data do not get points assigned and points are set to `NaN`. Also, the total score is set to `NaN`.
- `ZeroWOE` — Missing or out-of-range data get assigned a zero Weight-of-Evidence (WOE) value.
- `MinPoints` — Missing or out-of-range data get the minimum possible points for that predictor. This penalizes the score if higher scores are better.
- `MaxPoints` — Missing or out-of-range data get the maximum possible points for that predictor. This penalizes the score if lower scores are better.

Data Types: `char`

`Round` — Indicator whether to round points or scores

`'None'` (default) | character vector with values`'AllPoints'`, `'FinalScore'`

Indicator whether to round points or scores, specified as a character vector with values `'AllPoints'`, `'FinalScore'` or `'None'`, where:

- `None` — No rounding is applied.
- `AllPoints` — Apply rounding to each predictor's points before adding up the total score.
- `FinalScore` — Round the final score only (rounding is applied after all points are added up).

Data Types: `char`

`ShiftAndSlope` — Indicator for shift and slope scaling parameters
`[0,1]` (default) | numeric array with two elements `[Shift,Slope]`

Indicator for shift and slope scaling parameters for the credit scorecard, specified using numeric array with two elements `[Shift, Slope]`. `Slope` cannot be zero. The `ShiftAndSlope` values are used scale the scoring model.

Note `ShiftAndSlope`, `PointsOddsAndPDO`, and `WorstAndBestScores` are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (`BasePoints`, `Missing`, and `Round`) are not scaling methods and can be used together or with any one of the three scaling methods.

To remove a previous scaling and revert to unscaled scores, set `ShiftAndSlope` to`[0,1]`.

Data Types: `double`

`PointsOddsAndPDO` — Indicator for target points for given odds and double odds level
numeric array with three elements `[Points,Odds,PDO]`

Indicator for target points (`Points`) for a given odds level (`Odds`) and the desired number of points to double the odds (`PDO`), specified using numeric array with three elements `[Points,Odds,PDO]`. `Odds` must be a positive number. The `PointsOddsAndPDO` values are used to find scaling parameters for the scoring model.

Note The points to double the odds (`PDO`) may be positive or negative, depending on whether higher scores mean lower risk, or vice versa.

`ShiftAndSlope`, `PointsOddsAndPDO`, and `WorstAndBestScores` are scaling methods and you can use only one of these name-value pair arguments at one time. The other

three name-value pair arguments (`BasePoints`, `Missing`, and `Round`) are not scaling methods and can be used together or with any one of the three scaling methods.

To remove a previous scaling and revert to unscaled scores, set `ShiftAndSlope` to `[0,1]`.

Data Types: `double`

`WorstAndBestScores` — Indicator for worst (highest risk) and best (lowest risk) scores in scorecard
numeric array with two elements `[WorstScore,BestScore]`

Indicator for worst (highest risk) and best (lowest risk) scores in the scorecard, specified as a numeric array with two elements `[WorstScore,BestScore]`. `WorstScore` and `BestScore` must be different values. These `WorstAndBestScores` values are used to find scaling parameters for the scoring model.

Note `WorstScore` means the riskiest score, and its value could be lower or higher than the 'best' score. In other words, the 'minimum' score may be the 'worst' score or the 'best' score, depending on the desired scoring scale.

`ShiftAndSlope`, `PointsOddsAndPDO`, and `WorstAndBestScores` are scaling methods and you can use only one of these name-value pair arguments at one time. The other three name-value pair arguments (`BasePoints`, `Missing`, and `Round`) are not scaling methods and can be used together or with any one of the three scaling methods.

To remove a previous scaling and revert to unscaled scores, set `ShiftAndSlope` to `[0,1]`.

Data Types: `double`

Output Arguments

`sc` — Credit scorecard model
`creditscorecard` object

Credit scorecard model returned as an updated `creditscorecard` object. For more information on using the `creditscorecard` object, see `creditscorecard`.

Algorithms

The score of an individual *i* is given by the formula

```
Score(i) = Shift + Slope*(b0 + b1*WOE1(i) + b2*WOE2(i)+ ... +bp*WOEp(i))
```

where *bj* is the coefficient of the *j*th variable in the model, and WOE*j*(*i*) is the Weight of Evidence (WOE) value for the *i*th individual corresponding to the *j*th model variable. `Shift` and `Slope` are scaling constants further discussed below. The scaling constant can be controlled with `formatpoints`.

If the data for individual *i* is in the *i*-th row of a given dataset, to compute a score, the data(*i*,*j*) is binned using existing binning maps, and converted into a corresponding Weight of Evidence value `WOE`*j*(*i*). Using the model coefficients, the unscaled score is computed as

```
s = b0 + b1*WOE1(i) + ... +bp*WOEp(i).
```

For simplicity, assume in the description above that the *j*-th variable in the model is the *j*-th column in the data input, although, in general, the order of variables in a given dataset does not have to match the order of variables in the model, and the dataset could have additional variables that are not used in the model.

The formatting options can be controlled using `formatpoints`. When the base points are reported separately (see the `formatpoints` parameter `BasePoints`), the base points are given by

```
Base Points = Shift + Slope*b0,
```

and the points for the *j*-th predictor, *i*-th row are given by

```
Points_ji = Slope*(bj*WOEj(i))).
```

By default, the base points are not reported separately, in which case

```
Points_ji = (Shift + Slope*b0)/p + Slope*(bj*WOEj(i)),
```

where *p* is the number of predictors in the scorecard model.

By default, no rounding is applied to the points by the `score` function (`Round` is `None`). If `Round` is set to `AllPoints` using `formatpoints`, then the points for individual *i* for variable *j* are given by

```
points if rounding is 'AllPoints': round( Points_ji )
```

and, if base points are reported separately, the are also rounded. This yields integer-valued points per predictor, hence also integer-valued scores. If `Round` is set to `FinalScore` using `formatpoints`, then the points per predictor are not rounded, and only the final score is rounded

```
score if rounding is 'FinalScore': round(Score(i)).
```

Regarding the scaling parameters, the `Shift` parameter, and the `Slope` parameter can be set directly with the `ShiftAndSlope` parameter of `formatpoints`. Alternatively, you can use the `formatpoints` parameter for `WorstAndBestScores`. In this case, the parameters `Shift` and `Slope` are found internally by solving the system

```
Shift + Slope*smin = WorstScore,
Shift + Slope*smax = BestScore,
```

where `WorstScore` and `BestScore` are the first and second elements in the `formatpoints` parameter for `WorstAndBestScores` and *smin* and *smax* are the minimum and maximum possible unscaled scores:

```
smin = b0 + min(b1*WOE1) + ... +min(bp*WOEp),
smax = b0 + max(b1*WOE1) + ... +max(bp*WOEp).
```

A third alternative to scale scores is the `PointsOddsAndPDO` parameter in `formatpoints`. In this case, assume that the unscaled score *s* gives the log-odds for a row, and the `Shift` and `Slope` parameters are found by solving the following system

```
Points = Shift + Slope*log(Odds)
Points + PDO = Shift + Slope*log(2*Odds)
```

where `Points`, `Odds`, and `PDO` ("points to double the odds") are the first, second, and third elements in the `PointsOddsAndPDO` parameter.

Whenever a given dataset has a missing or out-of-range value data (*i*,*j*), the points for predictor *j*, for individual *i*, are set to `NaN` by default, which results in a missing score for that row (a `NaN` score). Using the `Missing` parameter for `formatpoints`, you can modify this behavior and set the corresponding Weight-of-Evidence (WOE) value to zero, or set the points to the minimum points, or the maximum points for that predictor.

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57

Introduced in R2014b

displaypoints

Return points per predictor per bin

Syntax

```
PointsInfo = displaypoints(sc)
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

Description

`PointsInfo = displaypoints(sc)` returns a table of points for all bins of all predictor variables used in the `creditscorecard` object after a linear logistic regression model is fit using `fitmodel` to the Weight of Evidence data. The `PointsInfo` table displays information on the predictor name, bin labels, and the corresponding points per bin.

`[PointsInfo,MinScore,MaxScore] = displaypoints(sc)` returns a table of points for all bins of all predictor variables used in the `creditscorecard` object after a linear logistic regression model is fit (`fitmodel`) to the Weight of Evidence data. The `PointsInfo` table displays information on the predictor name, bin labels, and the corresponding points per bin and `displaypoints`. In addition, the optional `MinScore` and `MaxScore` values are returned.

Examples

Display Unscaled Points

This example shows how to use `displaypoints` after a model is fitted to compute the unscaled points per bin, for a given predictor in the `creditscorecard` model.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the

`creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                    Estimate       SE       tStat       pValue
                    ________    ________    ______    __________

    (Intercept)     0.70239     0.064001    10.975    5.0538e-28
    CustAge         0.60833      0.24932      2.44      0.014687
    ResStatus         1.377      0.65272    2.1097      0.034888
    EmpStatus       0.88565        0.293    3.0227     0.0025055
    CustIncome      0.70164      0.21844    3.2121     0.0013179
    TmWBank          1.1074      0.23271    4.7589    1.9464e-06
    OtherCC          1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the fitting model.

```
PointsInfo = displaypoints(sc)

PointsInfo=30x3 table null
     Predictors           Bin            Points
    _____________    _______________    _________

    'CustAge'        '[-Inf,33)'         -0.15894
    'CustAge'        '[33,37)'           -0.14036
    'CustAge'        '[37,40)'          -0.060323
    'CustAge'        '[40,46)'           0.046408
    'CustAge'        '[46,48)'            0.21445
    'CustAge'        '[48,58)'            0.23039
    'CustAge'        '[58,Inf]'             0.479
    'ResStatus'      'Tenant'           -0.031252
    'ResStatus'      'Home Owner'         0.12696
    'ResStatus'      'Other'              0.37641
    'EmpStatus'      'Unknown'          -0.076317
    'EmpStatus'      'Employed'           0.31449
    'CustIncome'     '[-Inf,29000)'      -0.45716
    'CustIncome'     '[29000,33000)'     -0.10466
    'CustIncome'     '[33000,35000)'     0.052329
    'CustIncome'     '[35000,40000)'     0.081611
```

Display Scaled Points

This example shows how to use `formatpoints` after a model is fitted to format scaled points, and then use `displaypoints` to display the scaled points per bin, for a given predictor in the `creditscorecard` model.

Points become scaled when a range is defined. Specifically, a linear transformation from the unscaled to the scaled points is necessary. This transformation is defined either by supplying a shift and slope or by specifying the worst and best scores possible. (For more information, see `formatpoints`.)

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                    Estimate       SE       tStat       pValue
                    ________    ________    ______    __________

    (Intercept)     0.70239     0.064001    10.975    5.0538e-28
    CustAge         0.60833      0.24932      2.44      0.014687
    ResStatus         1.377      0.65272    2.1097      0.034888
    EmpStatus       0.88565        0.293    3.0227     0.0025055
    CustIncome      0.70164      0.21844    3.2121     0.0013179
    TmWBank          1.1074      0.23271    4.7589    1.9464e-06
    OtherCC          1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `formatpoints` function to scale providing the `'Worst'` and `'Best'` score values. The range provided below is a common score range.

```
sc = formatpoints(sc,'WorstAndBestScores',[300 850]);
```

Display the points information again to verify that the points are now scaled and also display the scaled minimum and maximum scores.

```
sc = formatpoints(sc,'WorstAndBestScores',[300 850]);
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors            Bin           Points
    _____________    _________________    _______

    'CustAge'        '[-Inf,33)'         46.396
    'CustAge'        '[33,37)'           48.727
    'CustAge'        '[37,40)'           58.772
    'CustAge'        '[40,46)'           72.167
    'CustAge'        '[46,48)'           93.256
    'CustAge'        '[48,58)'           95.256
    'CustAge'        '[58,Inf]'          126.46
    'ResStatus'      'Tenant'            62.421
    'ResStatus'      'Home Owner'        82.276
    'ResStatus'      'Other'             113.58
    'EmpStatus'      'Unknown'           56.765
    'EmpStatus'      'Employed'          105.81
    'CustIncome'     '[-Inf,29000)'      8.9706
    'CustIncome'     '[29000,33000)'     53.208
    'CustIncome'     '[33000,35000)'      72.91
    'CustIncome'     '[35000,40000)'     76.585


MinScore = 300

MaxScore = 850.0000
```

Notice that, as expected, the values of `MinScore` and `MaxScore` correspond to the worst and best possible scores.

Separate the Base Points From the Total Points

This example shows how to use `displaypoints` after a model is fitted to separate the base points from the rest of the points assigned to each predictor variable. The name-value pair argument `'BasePoints'` in the `formatpoints` function is a boolean that serves this purpose. By default, the base points are spread across all variables in the scorecard.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                    Estimate       SE       tStat       pValue
                    ________    _________    ______    __________

    (Intercept)     0.70239     0.064001     10.975    5.0538e-28
    CustAge         0.60833      0.24932      2.44      0.014687
    ResStatus         1.377      0.65272    2.1097      0.034888
    EmpStatus       0.88565        0.293    3.0227     0.0025055
    CustIncome      0.70164      0.21844    3.2121     0.0013179
    TmWBank          1.1074      0.23271    4.7589    1.9464e-06
    OtherCC          1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `formatpoints` function to separate the base points by providing the `'BasePoints'` name-value pair argument.

```
sc = formatpoints(sc,'BasePoints',true);
```

Display the base points, separated out from the other points, for predictors retained in the fitting model.

```
PointsInfo = displaypoints(sc)
```

```
PointsInfo=31x3 table
     Predictors           Bin            Points
    _____________    _______________    _________

    'BasePoints'     'BasePoints'         0.70239
    'CustAge'        '[-Inf,33)'         -0.25928
    'CustAge'        '[33,37)'           -0.24071
    'CustAge'        '[37,40)'           -0.16066
    'CustAge'        '[40,46)'          -0.053933
    'CustAge'        '[46,48)'            0.11411
    'CustAge'        '[48,58)'            0.13005
    'CustAge'        '[58,Inf]'           0.37866
    'ResStatus'      'Tenant'            -0.13159
    'ResStatus'      'Home Owner'        0.026616
    'ResStatus'      'Other'              0.27607
    'EmpStatus'      'Unknown'           -0.17666
    'EmpStatus'      'Employed'           0.21415
    'CustIncome'     '[-Inf,29000)'       -0.5575
    'CustIncome'     '[29000,33000)'       -0.205
    'CustIncome'     '[33000,35000)'    -0.048013
```

Display Points After Modifying Bin Labels

This example shows how to use `displaypoints` after a model is fitted and the `modifybins` function is used to provide user-defined bin labels for a numeric predictor.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                    Estimate       SE       tStat       pValue
                    ________    ________    ______    __________

    (Intercept)     0.70239     0.064001    10.975    5.0538e-28
    CustAge         0.60833      0.24932      2.44      0.014687
    ResStatus         1.377      0.65272    2.1097      0.034888
    EmpStatus       0.88565        0.293    3.0227     0.0025055
    CustIncome      0.70164      0.21844    3.2121     0.0013179
    TmWBank          1.1074      0.23271    4.7589    1.9464e-06
    OtherCC          1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `displaypoints` function to display point information.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=30x3 table
     Predictors            Bin            Points
    _____________    _______________    _________

    'CustAge'        '[-Inf,33)'          -0.15894
    'CustAge'        '[33,37)'            -0.14036
    'CustAge'        '[37,40)'           -0.060323
    'CustAge'        '[40,46)'            0.046408
    'CustAge'        '[46,48)'             0.21445
    'CustAge'        '[48,58)'             0.23039
    'CustAge'        '[58,Inf]'              0.479
    'ResStatus'      'Tenant'            -0.031252
    'ResStatus'      'Home Owner'          0.12696
    'ResStatus'      'Other'               0.37641
    'EmpStatus'      'Unknown'           -0.076317
    'EmpStatus'      'Employed'            0.31449
    'CustIncome'     '[-Inf,29000)'       -0.45716
    'CustIncome'     '[29000,33000)'      -0.10466
    'CustIncome'     '[33000,35000)'      0.052329
    'CustIncome'     '[35000,40000)'      0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

Use the `modifybins` function to specify user-defined bin labels for `'CustAge'` so that the bin ranges are described in natural language.

```
labels = {'Up to 32','33 to 36','37 to 39','40 to 45','46 to 47','48 to 57','At least 5
sc = modifybins(sc,'CustAge','BinLabels',labels);
```

Rerun `displaypoints` to verify the updated bin labels.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)

PointsInfo=30x3 table
     Predictors            Bin            Points
    _____________    _______________    _________

    'CustAge'        'Up to 32'           -0.15894
    'CustAge'        '33 to 36'           -0.14036
    'CustAge'        '37 to 39'          -0.060323
    'CustAge'        '40 to 45'           0.046408
    'CustAge'        '46 to 47'            0.21445
```

```
    'CustAge'       '48 to 57'           0.23039
    'CustAge'       'At least 58'          0.479
    'ResStatus'     'Tenant'           -0.031252
    'ResStatus'     'Home Owner'         0.12696
    'ResStatus'     'Other'              0.37641
    'EmpStatus'     'Unknown'          -0.076317
    'EmpStatus'     'Employed'           0.31449
    'CustIncome'    '[-Inf,29000)'      -0.45716
    'CustIncome'    '[29000,33000)'     -0.10466
    'CustIncome'    '[33000,35000)'     0.052329
    'CustIncome'    '[35000,40000)'     0.081611


MinScore = -1.3100

MaxScore = 3.0726
```

Compute the Predictor Weights

This example shows how to use a credit scorecard to compute the weights of the predictors. The weights of the predictors are determined from the range of points of each predictor, divided by the total range of points for the scorecard. The points for the scorecard not only take into consideration the betas, but also implicitly the binning of the predictor values and the corresponding weights of evidence.

Create a scorecard.

```
load CreditCardData.mat
sc = creditscorecard(data,'IDVar','CustID');
sc = autobinning(sc);
sc = fitmodel(sc);

1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
```

```
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Compute scorecard points and the `MinPts` and `MaxPts` scores.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
[PointsTable,MinPts,MaxPts] = displaypoints(sc);
PtsRange = MaxPts-MinPts;
disp(PointsTable(1:10,:));
```

```
    Predictors          Bin          Points
    ___________    _____________    ______

    'CustAge'      '[-Inf,33)'      52.821
    'CustAge'      '[33,37)'        54.161
    'CustAge'      '[37,40)'        59.934
    'CustAge'      '[40,46)'        67.633
    'CustAge'      '[46,48)'        79.755
    'CustAge'      '[48,58)'        80.905
    'CustAge'      '[58,Inf]'       98.838
    'ResStatus'    'Tenant'         62.031
    'ResStatus'    'Home Owner'     73.444
    'ResStatus'    'Other'          91.438
```

```
 fprintf('Min points: %g, Max points: %g\n',MinPts,MaxPts);
```

```
Min points: 355.505, Max points: 671.64
```

Compute the predictor weights.

```
Predictor = unique(PointsTable.Predictors,'stable');
NumPred = length(Predictor);
Weight = zeros(NumPred,1);
for ii=1:NumPred
   Ind = cellfun(@(x)strcmpi(Predictor{ii},x),PointsTable.Predictors);
   MaxPtsPred = max(PointsTable.Points(Ind));
   MinPtsPred = min(PointsTable.Points(Ind));
   Weight(ii) = 100*(MaxPtsPred-MinPtsPred)/PtsRange;
end

PredictorWeights = table(Predictor,Weight);
PredictorWeights(end+1,:) = PredictorWeights(end,:);
PredictorWeights.Predictor{end} = 'Total';
PredictorWeights.Weight(end) = sum(Weight);
disp(PredictorWeights)
```

```
     Predictor      Weight
    ____________    _______

    'CustAge'       14.556
    'ResStatus'      9.302
    'EmpStatus'     8.9174
    'CustIncome'    20.401
    'TmWBank'       25.884
    'OtherCC'       7.9885
    'AMBalance'     12.951
    'Total'            100
```

The weights are defined as the range of points for the predictor divided by the range of points for the scorecard.

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

Output Arguments

`PointsInfo` — One row per bin, per predictor, with the corresponding points

table

One row per bin, per predictor, with the corresponding points, returned as a table. For example:

Predictors	Bin	Points
Predictor_1	Bin_11	Points_11
	...	...
Predictor_2	Bin_21	Points_21
	...	...
Predictor_*j*	Bin_*ji*	Points_*ji*
	...	...

When base points are reported separately (see `formatpoints`), the first row of the returned `PointsInfo` table contains the base points.

`MinScore` — Minimum possible total score

scalar

Minimum possible total score, returned as a scalar.

Note Minimum score is the lowest possible total score in the mathematical sense, independently of whether a low score means high risk or low risk.

`MaxScore` — Maximum possible total score

scalar

Maximum possible total score, returned as a scalar.

Note Maximum score is the highest possible total score in the mathematical sense, independently of whether a high score means high risk or low risk.

Algorithms

The points for predictor j and bin i are, by default, given by

```
Points_ji = (Shift + Slope*b0)/p + Slope*(bj*WOEj(i))
```

where bj is the model coefficient of predictor j, p is the number of predictors in the model, and WOEj(i) is the Weight of Evidence (WOE) value for the i-th bin corresponding to the j-th model predictor. `Shift` and `Slope` are scaling constants.

When the base points are reported separately (see the `formatpoints` name-value pair argument `BasePoints`), the base points are given by

```
Base Points = Shift + Slope*b0,
```

and the points for the j-th predictor, i-th row are given by

```
Points_ji = Slope*(bj*WOEj(i))).
```

By default, the base points are not reported separately.

The minimum and maximum scores are:

```
MinScore = Shift + Slope*b0 + min(Slope*b1*WOE1) + ... +min(Slope*bp*WOEp)),
MaxScore = Shift + Slope*b0 + max(Slope*b1*WOE1) + ... +max(Slope*bp*WOEp)).
```

Use `formatpoints` to control the way points are scaled, rounded, and whether the base points are reported separately. See `formatpoints` for more information on format parameters and for details and formulas on these formatting options.

References

[1] Anderson, R. *The Credit Scoring Toolkit.* Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

“Case Study for a Credit Scorecard Analysis” on page 8-78
“Troubleshooting Credit Scorecard Results” on page 8-68
“Credit Scorecard Modeling Workflow” on page 8-62
“About Credit Scorecards” on page 8-57

Introduced in R2014b

fitmodel

Fit logistic regression model to Weight of Evidence (WOE) data

Syntax

```
sc = fitmodel(sc)

[sc,mdl] = fitmodel(sc)
[sc,mdl] = fitmodel(___,Name,Value)
```

Description

`sc = fitmodel(sc)` fits a logistic regression model to the Weight of Evidence (WOE) data and stores the model predictor names and corresponding coefficients in the `creditscorecard` object.

`fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic or manual binning process. The response variable is mapped so that "Good" is `1`, and "Bad" is `0`. This implies that higher (unscaled) scores correspond to better (less risky) individuals (smaller probability of default).

Alternatively, you can use `setmodel` to provide names of the predictors that you want in the logistic regression model, along with their corresponding coefficients.

`[sc,mdl] = fitmodel(sc)` fits a logistic regression model to the Weight of Evidence (WOE) data and stores the model predictor names and corresponding coefficients in the `creditscorecard` object. `fitmodel` returns an updated `creditscorecard` object and a `GeneralizedLinearModel` object containing the fitted model.

`fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic or manual binning process. The response variable is mapped so that "Good" is `1`, and "Bad" is `0`. This implies that higher (unscaled) scores correspond to better (less risky) individuals (smaller probability of default).

Alternatively, you can use `setmodel` to provide names of the predictors that you want in the logistic regression model, along with their corresponding coefficients.

`[sc,mdl] = fitmodel(___,Name,Value)` fits a logistic regression model to the Weight of Evidence (WOE) data using optional name-value pair arguments and stores the model predictor names and corresponding coefficients in the `creditscorecard` object. Using name-value pair arguments, you can select which Generalized Linear Model to fit the data. `fitmodel` returns an updated `creditscorecard` object and a `GeneralizedLinearModel` object containing the fitted model.

Examples

Fit a Stepwise Logistic Model

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc)

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
```

```
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                 IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Use `fitmodel` to fit a logistic regression model using Weight of Evidence (WOE) data. `fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic binning process. `fitmodel` then fits a logistic regression model using a stepwise method (by default).

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Fit a Stepwise Logistic Model For a creditscorecard Object Containing Weights

Use the `CreditCardData.mat` file to load the data (`dataWeights`) that contains a column (`RowWeights`) for the weights (using a dataset from Refaat 2011).

```
load CreditCardData
```

Create a `creditscorecard` object using the optional name-value pair argument for `'WeightsVar'`.

```
sc = creditscorecard(dataWeights,'IDVar','CustID','WeightsVar','RowWeights')
```

```
sc =

  creditscorecard with properties:

                GoodLabel: 0
               ResponseVar: 'status'
               WeightsVar: 'RowWeights'
                 VarNames: {1x12 cell}
        NumericPredictors: {1x6 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                    IDVar: 'CustID'
             PredictorVars: {1x9 cell}
                      Data: [1200x12 table]
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc =

  creditscorecard with properties:

                GoodLabel: 0
               ResponseVar: 'status'
               WeightsVar: 'RowWeights'
                 VarNames: {1x12 cell}
        NumericPredictors: {1x6 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
```

```
                IDVar: 'CustID'
          PredictorVars: {1x9 cell}
                   Data: [1200x12 table]
```

Use `fitmodel` to fit a logistic regression model using Weight of Evidence (WOE) data. `fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic binning process. `fitmodel` then fits a logistic regression model using a stepwise method (by default). When the optional name-value pair argument `'WeightsVar'` is used to specify observation (sample) weights, the `mdl` output uses the weighted counts with `stepwiseglm` and `fitglm`.

```
[sc,mdl] = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 764.3187, Chi2Stat = 15.81927, PValue = 6.968927e-05
2. Adding TmWBank, Deviance = 751.0215, Chi2Stat = 13.29726, PValue = 0.0002657942
3. Adding AMBalance, Deviance = 743.7581, Chi2Stat = 7.263384, PValue = 0.007037455

Generalized linear regression model:
    logit(status) ~ 1 + CustIncome + TmWBank + AMBalance
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70642     0.088702     7.964    1.6653e-15
    CustIncome      1.0268      0.25758    3.9862    6.7132e-05
    TmWBank         1.0973      0.31294    3.5063     0.0004543
    AMBalance       1.0039      0.37576    2.6717     0.0075464


1200 observations, 1196 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 36.4, p-value = 6.22e-08
```

Fit a Logistic Model with All Predictors

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc,'Algorithm','EqualFrequency')

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Use `fitmodel` to fit a logistic regression model using Weight of Evidence (WOE) data. `fitmodel` internally transforms all the predictor variables into WOE values, using the bins found with the automatic binning process. Set the `VariableSelection` name-value pair argument to `FullModel` to specify that all predictors must be included in the fitted logistic regression model.

```
sc = fitmodel(sc,'VariableSelection','FullModel');

Generalized linear regression model:
    status ~ [Linear formula with 10 terms in 9 predictors]
```

```
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat      pValue
                   ________    ________    _______    _________

    (Intercept)    0.70262     0.063862     11.002    3.734e-28
    CustAge        0.57683      0.27064     2.1313     0.033062
    TmAtAddress     1.0653      0.55233     1.9287     0.053762
    ResStatus       1.4189      0.65162     2.1775     0.029441
    EmpStatus      0.89916      0.29217     3.0776     0.002087
    CustIncome     0.77506      0.21942     3.5323    0.0004119
    TmWBank         1.0826      0.26583     4.0727    4.648e-05
    OtherCC         1.1354      0.52827     2.1493     0.031612
    AMBalance      0.99315      0.32642     3.0425    0.0023459
    UtilRate       0.16723      0.55745    0.29999      0.76419


1200 observations, 1190 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 85.6, p-value = 1.25e-14
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[sc,mdl] = fitmodel(sc,'VariableSelection','FullModel')`

`PredictorVars` — Predictor variables for fitting `creditscorecard` object
all predictors in the `creditscorecard` object (default) | cell array of character vectors

Predictor variables for fitting the `creditscorecard` object, specified when using a cell array of character vectors. When provided, the `creditscorecard` object property `PredictorsVars` is updated. When not provided, the predictors used to create the `creditscorecard` object (by using `creditscorecard`) are used.

`VariableSelection` — Variable selection method to fit logistic regression model
`'Stepwise'` (default) | character vector with values `'Stepwise'`, `'FullModel'`

The variable selection method to fit the logistic regression model, specified as a character vector with values `'Stepwise'` or `'FullModel'`:

- `Stepwise` — Uses a stepwise selection method which calls the Statistics and Machine Learning Toolbox function `stepwiseglm`. Only variables in `PredictorVars` can potentially become part of the model and uses the `StartingModel` name-value pair argument to select the starting model.
- `FullModel` — Fits a model with all predictor variables in the `PredictorVars` name-value pair argument and calls `fitglm`.

Note Only variables in the `PredictorVars` property of the `creditscorecard` object can potentially become part of the logistic regression model and only linear terms are included in this model with no interactions or any other higher-order terms.

The response variable is mapped so that “Good” is `1` and “Bad” is `0`.

Data Types: `char`

`StartingModel` — Initial model for `Stepwise` variable selection
`'Constant'` (default) | character vector with values `'Constant'`, `'Linear'`

Initial model for the `Stepwise` variable selection method, specified using a character vector with values `'Constant'` or `'Linear'`. This option determines the initial model (constant or linear) that the Statistics and Machine Learning Toolbox function `stepwiseglm` starts with.

- `Constant` — Starts the stepwise method with an empty (constant only) model.
- `Linear` — Starts the stepwise method from a full (all predictors in) model.

Note `StartingModel` is used only for the `Stepwise` option of `VariableSelection` and has no effect for the `FullModel` option of `VariableSelection`.

Data Types: `char`

`Display` — Indicator to display model information at command line

`'On'` (default) | character vector with values `'On'`, `'Off'`

Indicator to display model information at command line, specified using a character vector with value `'On'` or `'Off'`.

Data Types: `char`

Output Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object. The `creditscorecard` object contains information about the model predictors and coefficients used to fit the WOE data. For more information on using the `creditscorecard` object, see `creditscorecard`.

`mdl` — Fitted logistic model

`GeneralizedLinearModel` object

Fitted logistic model, retuned as an object of type `GeneralizedLinearModel` containing the fitted model. For more information on a `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.

Note When creating the `creditscorecard` object with `creditscorecard`, if the optional name-value pair argument `WeightsVar` was used to specify observation (sample) weights, then `mdl` uses the weighted counts with `stepwiseglm` and `fitglm`.

Definitions

Using `fitmodel` with Weights

When observation weights are provided in the credit scorecard `data`, the weights are used to calibrate the model coefficients.

The underlying Statistics and Machine Learning Toolboxfunctionality for `stepwiseglm` and `fitglm` supports observation weights. The weights also affect the logistic model through the WOE values. The WOE transformation is applied to all predictors before fitting the logistic model. The observation weights directly impact the WOE values. For more information, see “Using bininfo with Weights” on page 18-2157 and “Credit Scorecard Modeling Using Observation Weights” on page 8-65.

Therefore, the credit scorecard points and final score depend on the observation weights through both the logistic model coefficients and the WOE values.

Algorithms

For the logistic regression model used in the `creditscorecard` object, the probability of being “Bad” is given by

```
ProbBad = exp(-s) / (1 + exp(-s)).
```

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`GeneralizedLinearModel` | `autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitglm` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `stepwiseglm` | `validatemodel`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57
"Credit Scorecard Modeling Using Observation Weights" on page 8-65
"What Are Generalized Linear Models?" (Statistics and Machine Learning Toolbox)

Introduced in R2014b

setmodel

Set model predictors and coefficients

Syntax

```
sc = setmodel(sc,ModelPredictors,ModelCoefficients)
```

Description

`sc = setmodel(sc,ModelPredictors,ModelCoefficients)` sets the predictors and coefficients of a linear logistic regression model fitted outside the `creditscorecard` object and returns an updated `creditscorecard` object. The predictors and coefficients are used for the computation of scorecard points. Use `setmodel` in lieu of `fitmodel`, which fits a linear logistic regression model, because `setmodel` offers increased flexibility. For example, when a model fitted with `fitmodel` needs to be modified, you can use `setmodel`. For more information, see “Workflows for Using setmodel” on page 18-2096.

Note When using `setmodel`, the following assumptions apply:

- The model coefficients correspond to a linear logistic regression model (where only linear terms are included in the model and there are no interactions or any other higher-order terms).
- The model was previously fitted using Weight of Evidence (WOE) data with the response mapped so that ‘Good’ is `1` and ‘Bad’ is `0`.

Examples

Modify a GLM Model Fitted with `fitmodel`

This example shows how to use `setmodel` to make modifications to a logistic regression model initially fitted using the `fitmodel` function, and then set the new logistic regression model predictors and coefficients back into the `creditscorecard` object.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

The standard workflow is to use the `fitmodel` function to fit a logistic regression model using a stepwise method. However, `fitmodel` only supports limited options regarding the stepwise procedure. You can use the optional `mdl` output argument from `fitmodel` to get a copy of the fitted `GeneralizedLinearModel` object, to later modify.

```
[sc,mdl] = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769
```

```
Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    _______    __________

    (Intercept)    0.70239     0.064001    10.975     5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Suppose you want to include, or "force," the predictor `'UtilRate'` in the logistic regression model, even though the stepwise method did not include it in the fitted model. You can add `'UtilRate'` to the logistic regression model using the `GeneralizedLinearModel` object `mdl` directly.

```
mdl = mdl.addTerms('UtilRate')

mdl =
Generalized linear regression model:
    status ~ [Linear formula with 9 terms in 8 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat        pValue
                   ________    ________    ________    __________

    (Intercept)     0.70239    0.064001      10.975    5.0538e-28
    CustAge         0.60843     0.24936        2.44     0.014687
    ResStatus        1.3773      0.6529      2.1096     0.034896
    EmpStatus       0.88556     0.29303      3.0221    0.0025103
    CustIncome      0.70146      0.2186      3.2089    0.0013324
```

```
    TmWBank          1.1071     0.23307      4.7503    2.0316e-06
    OtherCC          1.0882     0.52918      2.0563       0.03975
    AMBalance        1.0413     0.36557      2.8483      0.004395
    UtilRate       0.013157     0.60864    0.021618       0.98275


1200 observations, 1191 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 5.26e-16
```

Use `setmodel` to update the model predictors and model coefficients in the `creditscorecard` object. The `ModelPredictors` input argument does not explicitly include a string for the intercept. However, the `ModelCoefficients` input argument does have the intercept information as its first element.

```
ModelPredictors = mdl.PredictorNames

ModelPredictors = 8x1 cell array
    {'CustAge'   }
    {'ResStatus' }
    {'EmpStatus' }
    {'CustIncome'}
    {'TmWBank'   }
    {'OtherCC'   }
    {'AMBalance' }
    {'UtilRate'  }


ModelCoefficients = mdl.Coefficients.Estimate

ModelCoefficients =

    0.7024
    0.6084
    1.3773
    0.8856
    0.7015
    1.1071
    1.0882
    1.0413
    0.0132



sc = setmodel(sc,ModelPredictors,ModelCoefficients);
```

Verify that `'UtilRate'` is part of the scorecard predictors by displaying the scorecard points.

```
pi = displaypoints(sc)
```

```
pi=33x3 table
     Predictors           Bin            Points
    _____________    _______________    _________

    'CustAge'        '[-Inf,33)'         -0.17152
    'CustAge'        '[33,37)'           -0.15295
    'CustAge'        '[37,40)'          -0.072892
    'CustAge'        '[40,46)'           0.033856
    'CustAge'        '[46,48)'            0.20193
    'CustAge'        '[48,58)'            0.21787
    'CustAge'        '[58,Inf]'           0.46652
    'ResStatus'      'Tenant'           -0.043826
    'ResStatus'      'Home Owner'         0.11442
    'ResStatus'      'Other'              0.36394
    'EmpStatus'      'Unknown'          -0.088843
    'EmpStatus'      'Employed'           0.30193
    'CustIncome'     '[-Inf,29000)'      -0.46956
    'CustIncome'     '[29000,33000)'     -0.11715
    'CustIncome'     '[33000,35000)'      0.039798
    'CustIncome'     '[35000,40000)'      0.069073
```

Fit a Logistic Regression Model Outside of the `creditscorecard` Object

This example shows how to use `setmodel` to fit a logistic regression model directly, without using the `fitmodel` function, and then set the new model predictors and coefficients back into the `creditscorecard` object. This approach gives more flexibility regarding options to control the stepwise procedure. This example fits a logistic regression model with a nondefault value for the `'PEnter'` parameter, the criterion to admit a new predictor in the logistic regression model during the stepwise procedure.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

The logistic regression model needs to be fit with Weight of Evidence (WOE) data. The WOE transformation is a special case of binning, since the data first needs to be binned, and then the binned information is mapped to the corresponding WOE values. This transformation is done using the `bindata` function. `bindata` has an argument that prepares the data for the model fitting step. By setting the `bindata` name-value pair argument for `'OutputType'` to `'WOEModelInput'`:

- All predictors are converted to WOE values.
- The output contains only predictors and response (no `'IDVar'` or any unused variables).
- Predictors with infinite or undefined (`NaN`) WOE values are discarded.
- The response values are mapped so that "Good" is `1` and "Bad" is `0` (this implies that higher unscaled scores correspond to better, less risky customers).

```
bd = bindata(sc,'OutputType','WOEModelInput');
```

For example, the first ten rows in the original data for the variables `'CustAge'`, `'ResStatus'`, `'CustIncome'`, and `'status'` (response variable) look like this:

```
data(1:10,{'CustAge' 'ResStatus' 'CustIncome' 'status'})

ans=10x4 table
    CustAge    ResStatus     CustIncome    status
```

```
    ________    ___________    ___________    _______

    53          Tenant         50000          0
    61          Home Owner     52000          0
    47          Tenant         37000          0
    50          Home Owner     53000          0
    68          Home Owner     53000          0
    65          Home Owner     48000          0
    34          Home Owner     32000          1
    50          Other          51000          0
    50          Tenant         52000          1
    49          Home Owner     53000          1
```

Here is how the same ten rows look after calling `bindata` with the name-value pair argument `'OutputType'` set to `'WOEModelInput'`:

```
bd(1:10,{'CustAge' 'ResStatus' 'CustIncome' 'status'})
```

```
ans=10x4 table
    CustAge     ResStatus    CustIncome    status
    ________    _________    __________    _______

     0.21378    -0.095564      0.47972     1
     0.62245     0.019329      0.47972     1
     0.18758    -0.095564     -0.026696    1
     0.21378     0.019329      0.47972     1
     0.62245     0.019329      0.47972     1
     0.62245     0.019329      0.47972     1
    -0.39568     0.019329      -0.29217    0
     0.21378      0.20049      0.47972     1
     0.21378    -0.095564      0.47972     0
     0.21378     0.019329      0.47972     0
```

Fit a logistic linear regression model using a stepwise method with the Statistics and Machine Learning Toolbox™ function `stepwiseglm`, but use a nondefault value for the `'PEnter'` and `'PRemove'` optional arguments. The predictors `'ResStatus'` and `'OtherCC'` would normally be included in the logistic linear regression model using default options for the stepwise procedure.

```
mdl = stepwiseglm(bd,'constant','Distribution','binomial',...
'Upper','linear','PEnter',0.025,'PRemove',0.05)
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306

mdl =
Generalized linear regression model:
    logit(status) ~ 1 + CustAge + EmpStatus + CustIncome + TmWBank + AMBalance
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70263     0.063759     11.02    3.0544e-28
    CustAge        0.57265       0.2482    2.3072      0.021043
    EmpStatus      0.88356      0.29193    3.0266      0.002473
    CustIncome     0.70399      0.21781    3.2321      0.001229
    TmWBank            1.1      0.23185    4.7443    2.0924e-06
    AMBalance       1.0313      0.32007    3.2221     0.0012724


1200 observations, 1194 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 81.4, p-value = 4.18e-16
```

Use `setmodel` to update the model predictors and model coefficients in the `creditscorecard` object. The `ModelPredictors` input argument does not explicitly include a string for the intercept. However, the `ModelCoefficients` input argument does have the intercept information as its first element.

```
ModelPredictors = mdl.PredictorNames
```

```
ModelPredictors = 5x1 cell array
    {'CustAge'   }
    {'EmpStatus' }
    {'CustIncome'}
    {'TmWBank'   }
    {'AMBalance' }
```

```
ModelCoefficients = mdl.Coefficients.Estimate
```

```
ModelCoefficients =

    0.7026
    0.5726
    0.8836
    0.7040
    1.1000
    1.0313
```

```
sc = setmodel(sc,ModelPredictors,ModelCoefficients);
```

Verify that the desired model predictors are part of the scorecard predictors by displaying the scorecard points.

```
pi = displaypoints(sc)
```

```
pi=25x3 table
     Predictors           Bin           Points
    ____________    _______________    _________

    'CustAge'       '[-Inf,33)'         -0.10354
    'CustAge'       '[33,37)'          -0.086059
    'CustAge'       '[37,40)'          -0.010713
    'CustAge'       '[40,46)'           0.089757
    'CustAge'       '[46,48)'            0.24794
    'CustAge'       '[48,58)'            0.26294
    'CustAge'       '[58,Inf]'           0.49697
    'EmpStatus'     'Unknown'          -0.035716
    'EmpStatus'     'Employed'           0.35417
    'CustIncome'    '[-Inf,29000)'      -0.41884
    'CustIncome'    '[29000,33000)'    -0.065161
    'CustIncome'    '[33000,35000)'     0.092353
    'CustIncome'    '[35000,40000)'      0.12173
    'CustIncome'    '[40000,42000)'      0.13259
    'CustIncome'    '[42000,47000)'       0.2854
    'CustIncome'    '[47000,Inf]'        0.47824
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`ModelPredictors` — Predictor names included in fitted model

cell array of character vectors with predictor values `{'PredictorName1','PredictorName2',...}`

Predictor names included in the fitted model, specified as a cell array of character vectors as `{'PredictorName1','PredictorName2',...}`. The predictor names must match predictor variable names in the `creditscorecard` object.

Note Do not include a character vector for the constant term in `ModelPredictors`, `setmodel` internally handles the `'(Intercept)'` term based on the number of model coefficients (see `ModelCoefficients`).

Data Types: `cell`

`ModelCoefficients` — Model coefficients corresponding to model predictors

numeric array with values `[coeff1,coeff2,..]`

Model coefficients corresponding to the model predictors, specified as a numeric array of model coefficients, `[coeff1,coeff2,..]`. If *N* is the number of predictor names provided in `ModelPredictors`, the size of `ModelCoefficients` can be *N* or *N*+1. If `ModelCoefficients` has *N*+1 elements, then the first coefficient is used as the `'(Intercept)'` of the fitted model. Otherwise, the `'(Intercept)'` is set to `0`.

Data Types: `double`

Output Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object. The `creditscorecard` object contains information about the model predictors and coefficients of the fitted model. For more information on using the `creditscorecard` object, see `creditscorecard`.

Definitions

Workflows for Using `setmodel`

When using `setmodel`, there are two possible workflows to set the final model predictors and model coefficients into a `creditscorecard` object.

The first workflow is:

- Use `fitmodel` to get the optional output argument `mdl`. This is a `GeneralizedLinearModel` object and you can add and remove terms, or modify the parameters of the stepwise procedure. Only linear terms can be in the model (no interactions or any other higher-order terms).
- Once the `GeneralizedLinearModel` object is satisfactory, set the final model predictors and model coefficients into the `creditscorecard` object using the `setmodel` input arguments for `ModelPredictors` and `ModelCoefficients`.

An alternate workflow is:

- Obtain the Weight of Evidence (WOE) data using `bindata`. Use the `'WOEModelInput'` option for the `'OutputType'` name-value pair argument in `bindata` to ensure that:
  - The predictors data is transformed to WOE.
  - Only predictors whose bins have finite WOE values are included.
  - The response variable is placed in the last column.
  - The response variable is mapped ("Good" is `1` and "Bad" is `0`).
- Use the data from the previous step to fit a linear logistic regression model (only linear terms in the model, no interactions, or any other higher-order terms). See, for example, `stepwiseglm`.

- Once the `GeneralizedLinearModel` object is satisfactory, set the final model predictors and model coefficients into the `creditscorecard` object using the `setmodel` input arguments for `ModelPredictors` and `ModelCoefficients`.

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`GeneralizedLinearModel` | `autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitglm` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `stepwiseglm` | `validatemodel`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57

Introduced in R2014b

bindata

Binned predictor variables

Syntax

```
bdata = bindata(sc)
bdata = bindata(sc,data)
bdata = bindata(sc,Name,Value)
```

Description

`bdata = bindata(sc)` binned predictor variables returned as a table. This is a table of the same size as the data input, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

`bdata = bindata(sc,data)` returns a table of binned predictor variables. `bindata` returns a table of the same size as the `creditscorecard` data, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

`bdata = bindata(sc,Name,Value)` binned predictor variables returned as a table using optional name-value pair arguments. This is a table of the same size as the data input, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

Examples

Bin `creditscorecard` Data as Bin Numbers, Categories, or WOE Values

This example shows how to use the `bindata` function to simply bin or discretize data.

Suppose bin ranges of

- '0 to 30'
- '31 to 50'
- '51 and up'

are determined for the age variable (via manual or automatic binning). If a data point with age 41 is given, binning this data point means placing it in the bin for 41 years old, which is the second bin, or the '31 to 50' bin. Binning is then the mapping from the original data, into discrete groups or bins. In this example, you can say that a 41-year old is mapped into bin number 2, or that it is binned into the '31 to 50' category. If you know the Weight of Evidence (WOE) value for each of the three bins, you could also replace the data point 41 with the WOE value corresponding to the second bin. `bindata` supports the three binning formats just mentioned:

- Bin number (where the `'OutputType'` name-value pair argument is set to `'BinNumber'`); this is the default option, and in this case, 41 is mapped to bin 2.
- Categorical (where the `'OutputType'` name-value pair argument is set to `'Categorical'`); in this case, 41 is mapped to the '31 to 50' bin.
- WOE value (where the `'OutputType'` name-value pair argument is set to `'WOE'`); in this case, 41 is mapped to the WOE value of bin number 2.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc,'CustAge')
```

```
ans=8x6 table
        Bin         Good    Bad     Odds        WOE        InfoValue
    ____________    ____    ___    _______    _________    _________

    '[-Inf,33)'      70      53    1.3208      -0.42622     0.019746
    '[33,37)'        64      47    1.3617      -0.39568     0.015308
    '[37,40)'        73      47    1.5532      -0.26411    0.0072573
    '[40,46)'       174      94    1.8511     -0.088658     0.001781
    '[46,48)'        61      25      2.44       0.18758    0.0024372
    '[48,58)'       263     105    2.5048       0.21378     0.013476
    '[58,Inf]'       98      26    3.7692       0.62245       0.0352
    'Totals'        803     397    2.0227           NaN     0.095205
```

These are the first 10 age values in the original data, used to create the `creditscorecard` object.

```
data(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    _______

    53
    61
    47
    50
    68
    65
    34
    50
    50
    49
```

Bin scorecard data into bin numbers (default behavior).

```
bdata = bindata(sc);
```

According to the bin information, the first age should be mapped into the fourth bin, the second age into the fifth bin, etc. These are the first 10 binned ages, in bin-number format.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    _______

    6
    7
    5
    6
    7
    7
    2
    6
    6
    6
```

Bin the scorecard data and show their bin labels. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'Categorical'`.

```
bdata = bindata(sc,'OutputType','Categorical');
```

These are the first 10 binned ages, in categorical format.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    ________

    [48,58)
    [58,Inf]
    [46,48)
    [48,58)
    [58,Inf]
    [58,Inf]
    [33,37)
    [48,58)
    [48,58)
```

```
    [48,58)
```

Convert the scorecard data to WOE values. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'WOE'`.

```
bdata = bindata(sc,'OutputType','WOE');
```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values that are internally displayed using the `bininfo` function.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    ________

     0.21378
     0.62245
     0.18758
     0.21378
     0.62245
     0.62245
    -0.39568
     0.21378
     0.21378
     0.21378
```

Bin Additional "Test" Data

This example shows how to use the `bindata` function's optional input for the data to bin. If not provided, `bindata` bins the `creditscorecard` training data. However, if a different dataset needs to be binned, for example, some "test" data, this can be passed into `bindata` as an optional input.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc,'CustAge')
```

```
ans=8x6 table
        Bin         Good    Bad     Odds        WOE        InfoValue
    ____________    ____    ___    ______    _________    _________

    '[-Inf,33)'      70      53    1.3208     -0.42622     0.019746
    '[33,37)'        64      47    1.3617     -0.39568     0.015308
    '[37,40)'        73      47    1.5532     -0.26411    0.0072573
    '[40,46)'       174      94    1.8511    -0.088658     0.001781
    '[46,48)'        61      25      2.44      0.18758    0.0024372
    '[48,58)'       263     105    2.5048      0.21378     0.013476
    '[58,Inf]'       98      26    3.7692      0.62245       0.0352
    'Totals'        803     397    2.0227          NaN     0.095205
```

For the purpose of illustration, take a few rows from the original data as "test" data and display the first 10 age values in the test data.

```
tdata = data(101:110,:);
tdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    _______

    34
    59
    64
    61
    28
    65
    55
    37
    49
    51
```

Convert the test data to WOE values. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'WOE'`, passing the test data (`tdata`) as an optional input.

```
bdata = bindata(sc,tdata,'OutputType','WOE')
```

```
bdata=10x11 table
    CustID    CustAge     TmAtAddress    ResStatus    EmpStatus    CustIncome    TmWBar
    ______    ________    ___________    _________    _________    __________    ______

    101       -0.39568    -0.087767      -0.095564      0.2418     -0.011271      0.768
    102        0.62245      0.14288       0.019329    -0.19947       0.20579    -0.131
    103        0.62245      0.02263       0.019329      0.2418       0.47972    -0.121
    104        0.62245      0.02263      -0.095564      0.2418       0.47972    -0.121
    105       -0.42622      0.02263       0.019329      0.2418      -0.06843      0.768
    106        0.62245      0.02263       0.019329    -0.19947       0.20579    -0.131
    107        0.21378    -0.087767      -0.095564      0.2418       0.47972      0.267
    108       -0.26411    -0.087767       0.019329    -0.19947      -0.29217    -0.131
    109        0.21378    -0.087767      -0.095564      0.2418      -0.026696    -0.131
    110        0.21378    -0.087767       0.019329      0.2418       0.20579    -0.131
```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values displayed internally by `bininfo`.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
```

```
    ________

   -0.39568
    0.62245
    0.62245
    0.62245
   -0.42622
    0.62245
    0.21378
   -0.26411
    0.21378
    0.21378
```

Apply a Weight of Evidence (WOE) Transformation to Data

`bindata` supports the following types of WOE transformation:

- When the `'OutputType'` name-value argument is set to `'WOE'`, `bindata` simply applies the WOE transformation to all predictors and keeps the rest of the variables in the original data in place and unchanged.
- When the `'OutputType'` name-value pair argument is set to `'WOEModelInput'`, `bindata` returns a table that can be used directly as an input for fitting a logistic regression model for the scorecard. In this case, `bindata`:
- Applies WOE transformation to all predictors.
- Returns predictor variables, but no `IDVar` or unused variables are included in the output.
- Includes the mapped response variable as the last column.
- The `fitmodel` function calls `bindata` internally using the `'WOEModelInput'` option to fit the logistic regression model for the `creditscorecard` model.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc,'CustAge')
```

```
ans=8x6 table
        Bin         Good    Bad     Odds        WOE        InfoValue
    ____________    ____    ___    ______    _________    _________

    '[-Inf,33)'      70      53    1.3208     -0.42622     0.019746
    '[33,37)'        64      47    1.3617     -0.39568     0.015308
    '[37,40)'        73      47    1.5532     -0.26411    0.0072573
    '[40,46)'       174      94    1.8511    -0.088658     0.001781
    '[46,48)'        61      25      2.44      0.18758    0.0024372
    '[48,58)'       263     105    2.5048      0.21378     0.013476
    '[58,Inf]'       98      26    3.7692      0.62245       0.0352
    'Totals'        803     397    2.0227          NaN     0.095205
```

These are the first 10 age values in the original data, used to create the `creditscorecard` object.

```
data(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    _______
```

```
    53
    61
    47
    50
    68
    65
    34
    50
    50
    49
```

Convert the test data to WOE values. To do this, set the `bindata` name-value pair argument for `'OutputType'` to `'WOE'`.

```
bdata = bindata(sc,'OutputType','WOE');
```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values displayed internally by `bininfo`.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    ________

     0.21378
     0.62245
     0.18758
     0.21378
     0.62245
     0.62245
    -0.39568
     0.21378
     0.21378
     0.21378
```

These are the first 10 binned ages, in WOE format. The ages are mapped to the WOE values displayed internally by `bininfo`.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
```

```
    ________

     0.21378
     0.62245
     0.18758
     0.21378
     0.62245
     0.62245
    -0.39568
     0.21378
     0.21378
     0.21378
```

The size of the original data and the size of `bdata` output are the same because `bindata` leaves unused variables (such as `'IDVar'`) unchanged and in place.

```
whos data bdata
```

```
  Name           Size              Bytes  Class    Attributes

  bdata       1200x11             109027  table
  data        1200x11              84699  table
```

The response values are the same in the original data and in the binned data because, by default, `bindata` does not modify response values.

```
disp([data.status(1:10) bdata.status(1:10)])
```

```
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     1     1
     0     0
     1     1
     1     1
```

When fitting a logistic regression model with WOE data, set the `'OutputType'` name-value pair argument to `'WOEModelInput'`.

```
bdata = bindata(sc,'OutputType','WOEModelInput');
```

The binned predictor data is the same as when the `'OutputType'` name-value pair argument is set to `'WOE'`.

```
bdata(1:10,'CustAge')
```

```
ans=10x1 table
    CustAge
    ________

     0.21378
     0.62245
     0.18758
     0.21378
     0.62245
     0.62245
    -0.39568
     0.21378
     0.21378
     0.21378
```

However, the size of the original data and the size of `bdata` output are different. This is because `bindata` removes unused variables (such as `'IDVar'`).

```
whos data bdata
```

```
  Name          Size             Bytes  Class    Attributes

  bdata      1200x10             99191  table
  data       1200x11             84699  table
```

The response values are also modified in this case and are mapped so that "Good" is `1` and "Bad" is `0`.

```
disp([data.status(1:10) bdata.status(1:10)])
```

```
     0     1
     0     1
     0     1
     0     1
     0     1
     0     1
     1     0
     0     1
```

```
     1      0
     1      0
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`data` — Data to bin given the rules set in `creditscorecard` object

table

Data to bin given the rules set in the `creditscorecard` object, specified using a table. By default, `data` is set to the `creditscorecard` object's raw data.

Before creating a `creditscorecard` object, perform a data preparation task to have an appropriately structured `data` as input to a `creditscorecard` object.

Data Types: `table`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `bdata = bindata(sc,'OutputType','WOE','ResponseFormat','Mapped')`

`OutputType` — Output format

`'BinNumber'` (default) | character vector with values `'BinNumber'`, `'Categorical'`, `'WOE'`

Output format, specified as a character vector with the following values:

- `BinNumber` — Returns the bin numbers corresponding to each observation.
- `Categorical` — Returns the bin label corresponding to each observation.
- `WOE` — Returns the Weight of Evidence (WOE) corresponding to each observation.
- `WOEModelInput` — Use this option when fitting a model. This option:
  - Returns the Weight of Evidence (WOE) corresponding to each observation.
  - Returns predictor variables, but no `IDVar` or unused variables are included in the output.
  - Discards any predictors whose bins have `Inf` or `NaN` WOE values.
  - Includes the mapped response variable as the last column.

Note When the `bindata` name-value pair argument `'OutputType'` is set to `'WOEModelInput'`, the `bdata` output only contains the columns corresponding to predictors whose bins do not have `Inf` or `NaN` Weight of Evidence (WOE) values, and `bdata` includes the mapped response as the last column.

Missing data (if any) are included in the `bdata` output as missing data as well, and do not influence the rules to discard predictors when `'OutputType'` is set to `'WOEModelInput'`.

Data Types: `char`

`ResponseFormat` — Response values format

`'RawData'` (default) | character vector with values `'RawData'`, `'Mapped'`

Response values format, specified using a character vector with the following values:

- `RawData` — The response variable is copied unchanged into the `bdata` output.
- `Mapped` — The response values are modified (if necessary) so that "Good" is mapped to `1`, and "Bad" is mapped to `0`.

Data Types: `char`

Output Arguments

`bdata` — Binned predictor variables
table

Binned predictor variables, returned as a table. This is a table of the same size (see exception in the following Note) as the data input, but only the predictors specified in the `creditscorecard` object's `PredictorVars` property are binned and the remaining ones are unchanged.

Note When the `bindata` name-value pair argument `'OutputType'` is set to `'WOEModelInput'`, the `bdata` output only contains the columns corresponding to predictors whose bins do not have `Inf` or `NaN` Weight of Evidence (WOE) values, and `bdata` includes the mapped response as the last column.

Missing data (if any) are included in the `bdata` output as missing data as well, and do not influence the rules to discard predictors when `'OutputType'` is set to `'WOEModelInput'`.

References

[1] Anderson, R. *The Credit Scoring Toolkit.* Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

`autobinning` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

“Case Study for a Credit Scorecard Analysis” on page 8-78
“Troubleshooting Credit Scorecard Results” on page 8-68
“Credit Scorecard Modeling Workflow” on page 8-62
“About Credit Scorecards” on page 8-57

Introduced in R2014b

plotbins

Plot histogram counts for predictor variables

Syntax

```
plotbins(sc,PredictorName)
hFigure = plotbins(sc,PredictorName)
hFigure = plotbins(___,Name,Value)
```

Description

`plotbins(sc,PredictorName)` plots histogram counts for given predictor variables. When a predictor's bins are modified using `modifybins` or `autobinning`, rerun `plotbins` to update the figure to reflect the change.

`hFigure = plotbins(sc,PredictorName)` returns a handle to the figure. `plotbins` plots histogram counts for given predictor variables. When a predictor's bins are modified using `modifybins` or `autobinning`, rerun `plotbins` to update the figure to reflect the change.

`hFigure = plotbins(___,Name,Value)` returns a handle to the figure. `plotbins` plots histogram counts for given predictor variables using optional name-value pair arguments. When a predictor's bins are modified using `modifybins` or `autobinning`, rerun `plotbins` to update the figure to reflect the change.

Examples

Plot a Histogram for Bin Information

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the `PredictorName` input argument for `CustIncome` using the defaults for the algorithm `Monotone`.

```
sc = autobinning(sc, 'CustIncome')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x7 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: ''
            PredictorVars: {1x10 cell}
                    Data: [1200x11 table]
```

Use `bininfo` to display the autobinned data.

```
[bi, cp] = bininfo(sc, 'CustIncome')
```

```
bi=8x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,29000)'       53      58    0.91379     -0.79457       0.06364
    '[29000,33000)'      74      49     1.5102     -0.29217     0.0091366
    '[33000,35000)'      68      36     1.8889     -0.06843    0.00041042
    '[35000,40000)'     193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'      68      34          2    -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579     0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972      0.041657
    'Totals'            803     397     2.0227          NaN       0.12285


cp =

       29000
       33000
       35000
```

```
    40000
    42000
    47000
```

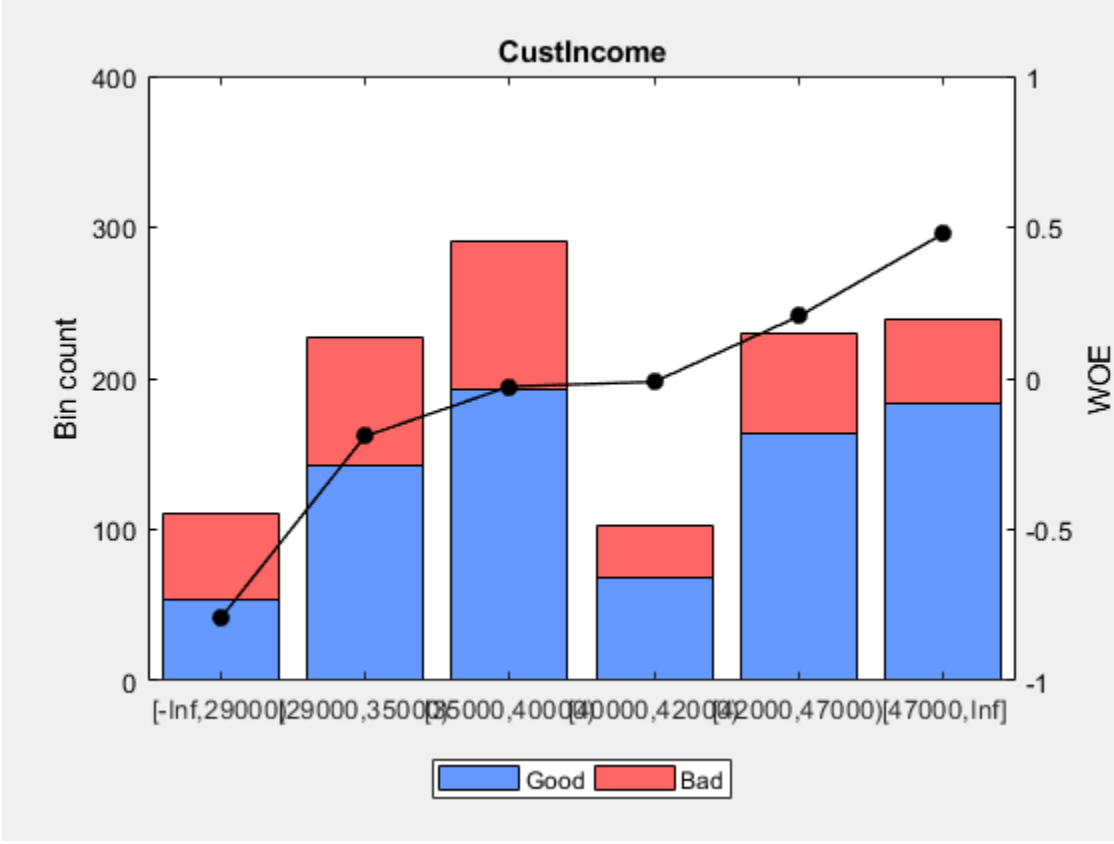
Manually remove the second cut point (the boundary between the second and third bins) to merge bins two and three. Use the `modifybins` function to update the scorecard and then display updated bin information.

```
cp(2) = [];
sc = modifybins(sc,'CustIncome','CutPoints',cp);
bi = bininfo(sc,'CustIncome')
```

```
bi=7x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,29000)'       53      58    0.91379     -0.79457       0.06364
    '[29000,35000)'     142      85     1.6706     -0.19124     0.0071274
    '[35000,40000)'     193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'      68      34          2    -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579     0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972      0.041657
    'Totals'            803     397     2.0227          NaN       0.12043
```

Plot the histogram count for updated bin information for the `PredictorName` called `CustIncome`.

```
plotbins(sc,'CustIncome');
```



Plot a Histogram for Bin Information Using Name-Value Pair Arguments

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the `PredictorName` input argument for `CustIncome` using the defaults for the algorithm `Monotone`.

```
sc = autobinning(sc, 'CustIncome')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x7 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: ''
            PredictorVars: {1x10 cell}
                    Data: [1200x11 table]
```

Use `bininfo` to display the autobinned data.

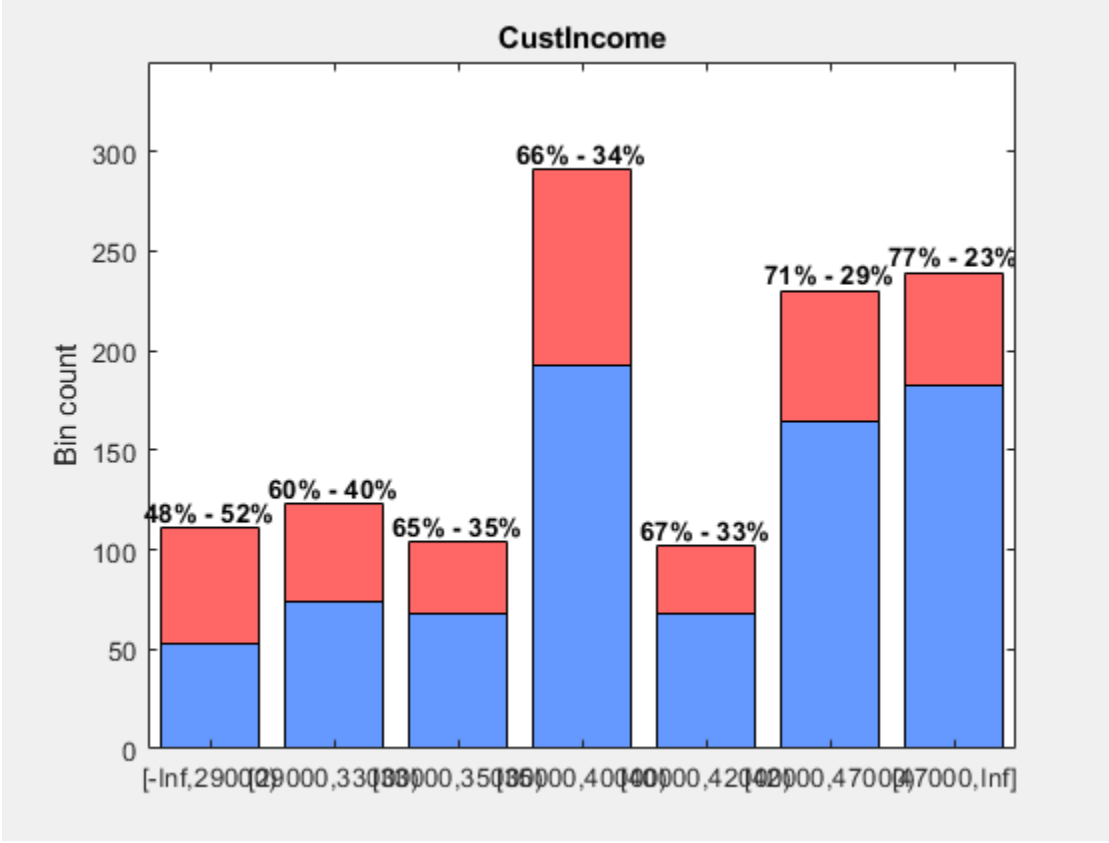
```
[bi, cp] = bininfo(sc, 'CustIncome')
```

```
bi=8x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,29000)'       53      58    0.91379     -0.79457        0.06364
    '[29000,33000)'      74      49     1.5102     -0.29217      0.0091366
    '[33000,35000)'      68      36     1.8889     -0.06843     0.00041042
    '[35000,40000)'     193      98     1.9694     -0.026696    0.00017359
    '[40000,42000)'      68      34          2     -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579      0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972       0.041657
    'Totals'            803     397     2.0227          NaN       0.12285


cp =

       29000
       33000
       35000
       40000
       42000
       47000
```

Plot the bin information for `CustIncome` without the Weight of Evidence (WOE) line and without a legend by setting the `'WOE'` and `'Legend'` name-value arguments to `'Off'`. Also, set the `'BinText'` name-value pair argument to `'PercentRows'` to show as text

over the plot bars for the proportion of "Good" and "Bad" within each bin, that is, the probability of "Good" and "Bad" within each bin.

```
plotbins(sc,'CustIncome','WOE','Off','Legend','Off','BinText','PercentRows');
```



- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`PredictorName` — Name of one or more predictors to plot

character vector with predicator name | cell array of character vectors with predictor names

Name of one or more predictors to plot, specified using a character vector or cell array of character vectors containing one or more names of the predictors.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `plotbins(sc,PredictorName,'BinText','Count','WOE','On')`

`BinText` — Information to display on top of plotted bin counts

`'None'` (default) | character vector with values `'None'`, `'Count'`, `'PercentRows'`, `'PercentCols'`, `'PercentTotal'`

Information to display on top of plotted bin counts, specified using a character vector with values:

- `None` — No text is displayed on top of the bins.
- `Count` — For each bin, displays the count for “Good” and “Bad.”
- `PercentRows` — For each bin, displays the count for “Good” and “Bad” as a percentage of the number of observations in the bin.
- `PercentCols` — For each bin, displays the count for “Good” and “Bad” as a percentage of the total “Good” and total “Bad” in the entire sample.

- `PercentTotal` — For each bin, displays the count for “Good” and “Bad” as a percentage of the total number of observations in the entire sample.

Data Types: `char`

`WOE` — Indicator for Weight of Evidence (WOE)
`'On'` (default) | character vector with values `'On'`, `'Off'`

Indicator for Weight of Evidence (WOE) line, specified using a character vector with values `On` or `Off`. When set to `On`, the WOE line is plotted on top of the histogram.

Data Types: `char`

`Legend` — Indicator for legend on plot
`'On'` (default) | character vector with values `'On'`, `'Off'`

Indicator for legend on the plot, specified using a character vector with values `On` or `Off`.

Data Types: `char`

Output Arguments

`hFigure` — Figure handle for histogram plot for predictor variables
figure object

Figure handle for histogram plot for predictor variables, returned as figure object or array of figure objects if more than one `PredictorName` is specified as an input.

References

[1] Anderson, R. *The Credit Scoring Toolkit.* Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

“Case Study for a Credit Scorecard Analysis” on page 8-78
“Troubleshooting Credit Scorecard Results” on page 8-68
“Credit Scorecard Modeling Workflow” on page 8-62
“About Credit Scorecards” on page 8-57

Introduced in R2014b

modifybins

Modify predictor's bins

Syntax

```
sc = modifybins(sc,PredictorName,Name,Value)
```

Description

`sc = modifybins(sc,PredictorName,Name,Value)` manually modifies predictor bins for numeric predictors or categorical predictors using optional name-value pair arguments. For numeric predictors, minimum value, maximum value, and cut points can be specified. For categorical predictors, category groupings can be specified. Bin labels can be specified for both types of predictors.

Examples

Modify Predictor Bins for Numeric Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The predictor `CustIncome` is numeric. By default, each value of a predictor is placed in a separate bin.

```
bi = bininfo(sc,'CustIncome')
```

```
bi=46x6 table
      Bin       Good     Bad      Odds          WOE         InfoValue
    ________    ____    ___    ________    __________    __________
```

```
    '18000'     2       3     0.66667      -1.1099     0.0056227
    '19000'     1       2         0.5      -1.3976     0.0053002
    '20000'     4       2           2    -0.011271    6.3641e-07
    '21000'     6       3           2    -0.011271    9.5462e-07
    '22000'     4       2           2    -0.011271    6.3641e-07
    '23000'     4       4           1     -0.70442     0.0035885
    '24000'     5       5           1     -0.70442     0.0044856
    '25000'     4       9     0.44444      -1.5153      0.026805
    '26000'     4      11     0.36364       -1.716      0.038999
    '27000'     6       6           1     -0.70442     0.0053827
    '28000'    13      11      1.1818     -0.53736     0.0061896
    '29000'    11      10         1.1     -0.60911     0.0069988
    '30000'    18      16       1.125     -0.58664      0.010493
    '31000'    24       8           3      0.39419     0.0038382
    '32000'    21      15         1.4     -0.36795     0.0042797
    '33000'    35      19      1.8421    -0.093509    0.00039951
```

Use `modifybins` to set a minimum value of 0, and cut points every 10000, from 20000 to 60000. Display updated bin information, including cut points.

```
sc = modifybins(sc,'CustIncome','MinValue',0,'CutPoints',20000:10000:60000);
[bi,cp] = bininfo(sc,'CustIncome')
```

```
bi=7x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    _________

    '[0,20000)'            3       5        0.6      -1.2152     0.010765
    '[20000,30000)'       61      63    0.96825     -0.73668     0.060942
    '[30000,40000)'      324     173     1.8728    -0.076967    0.0024846
    '[40000,50000)'      304     123     2.4715      0.20042     0.013781
    '[50000,60000)'      103      32     3.2188      0.46457     0.022144
    '[60000,Inf]'          8       1          8        1.375     0.010235
    'Totals'             803     397     2.0227          NaN      0.12035


cp =

       20000
       30000
       40000
       50000
       60000
```

The first and last bins contain very few points. To merge the first bin into the second one, remove the first cut point. Similarly, to merge the last bin into the second-to-last one, remove the last cut point. Then use modifybins to update the scorecard, and display updated bin information.

```
cp(1)=[];
cp(end)=[];
sc = modifybins(sc,'CustIncome','CutPoints',cp);
bi = bininfo(sc,'CustIncome')
```

```
bi=5x6 table
          Bin           Good    Bad     Odds        WOE       InfoValue
    ________________    ____    ___    _______    _________    _________

    '[0,30000)'          64      68    0.94118     -0.76504     0.070065
    '[30000,40000)'     324     173     1.8728    -0.076967    0.0024846
    '[40000,50000)'     304     123     2.4715      0.20042     0.013781
    '[50000,Inf]'       111      33     3.3636       0.5086     0.028028
    'Totals'            803     397     2.0227          NaN      0.11436
```

Modify Predictor Bins for Categorical Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The binning map or rules for categorical data are summarized in a "category grouping" table, returned as an optional output. By default, each category is placed in a separate bin. Here is the information for the predictor `ResStatus`.

```
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=4x6 table
        Bin          Good    Bad     Odds        WOE       InfoValue
    _____________    ____    ___    _______    _________    _________

    'Home Owner'     365     177    2.0621      0.019329    0.0001682
    'Tenant'         307     167    1.8383     -0.095564    0.0036638
    'Other'          131      53    2.4717       0.20049    0.0059418
```

```
    'Totals'        803     397    2.0227          NaN    0.0097738


cg=3x2 table
      Category      BinNumber
    ____________    _________

    'Home Owner'    1
    'Tenant'        2
    'Other'         3
```

To group categories `'Tenant'` and `'Other'`, modify the category grouping table `cg`, so the bin number for `'Other'` is the same as the bin number for `'Tenant'`. Then use `modifybins` to update the scorecard.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','CatGrouping',cg);
```

Display the updated bin information. Note that the bin labels has been updated and that the bin membership information is contained in the category grouping `cg`.

```
[bi,cg] = bininfo(sc,'ResStatus')

bi=3x6 table
      Bin       Good    Bad     Odds        WOE        InfoValue
    ________    ____    ___    ______    _________    __________

    'Group1'    365     177    2.0621     0.019329     0.0001682
    'Group2'    438     220    1.9909    -0.015827    0.00013772
    'Totals'    803     397    2.0227          NaN    0.00030592


cg=3x2 table
      Category      BinNumber
    ____________    _________

    'Home Owner'    1
    'Tenant'        2
    'Other'         2
```

Merge Bins for Numerical and Categorical Predictors

Create a `creditscorecard` object (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
```

For the numerical predictor `CustAge`, use the `modifybins` function to set the following cut points:

```
cp = [25 37 49 65];
sc = modifybins(sc,'CustAge','CutPoints',cp,'MinValue',0,'MaxValue',75);
bininfo(sc,'CustAge')
```

```
ans=6x6 table
       Bin       Good    Bad     Odds        WOE        InfoValue
    _________    ____    ___    ______    _________    _________

    '[0,25)'       9       8     1.125     -0.58664    0.0052464
    '[25,37)'    125      92    1.3587     -0.39789     0.030268
    '[37,49)'    340     183    1.8579    -0.084959    0.0031898
    '[49,65)'    298     108    2.7593      0.31054     0.030765
    '[65,75]'     31       6    5.1667      0.93781     0.022031
    'Totals'     803     397    2.0227          NaN       0.0915
```

Use the `modifybins` function to merge the 2nd and 3rd bins.

```
sc = modifybins(sc,'CustAge','CutPoints',cp([1 3 4]));
bininfo(sc,'CustAge')
```

```
ans=5x6 table
       Bin       Good    Bad     Odds        WOE        InfoValue
    _________    ____    ___    ______    _________    _________

    '[0,25)'       9       8     1.125     -0.58664    0.0052464
    '[25,49)'    465     275    1.6909     -0.17915     0.020355
    '[49,65)'    298     108    2.7593      0.31054     0.030765
    '[65,75]'     31       6    5.1667      0.93781     0.022031
    'Totals'     803     397    2.0227          NaN     0.078397
```

Display bin information for the categorical predictor `ResStatus`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
       Bin          Good    Bad     Odds        WOE        InfoValue
    ____________    ____    ___    _______    _________    _________

    'Home Owner'    365     177    2.0621      0.019329    0.0001682
    'Tenant'        307     167    1.8383     -0.095564    0.0036638
    'Other'         131      53    2.4717      0.20049     0.0059418
    'Totals'        803     397    2.0227           NaN    0.0097738
```

Use the `modifybins` function to merge categories 2 and 3.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','CatGrouping',cg);
bininfo(sc,'ResStatus')
```

```
ans=3x6 table
      Bin        Good    Bad     Odds        WOE        InfoValue
    _________    ____    ___    _______    _________    __________

    'Group1'     365     177    2.0621      0.019329     0.0001682
    'Group2'     438     220    1.9909     -0.015827     0.00013772
    'Totals'     803     397    2.0227           NaN     0.00030592
```

Split Bins for Numerical and Categorical Predictors

Create a `creditscorecard` object (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0)
```

```
sc =
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

For the numerical predictor `TmAtAddress`, use the `modifybins` function to set the following cut points:

```
cp = [30 80 120];
sc = modifybins(sc,'TmAtAddress','CutPoints',cp,'MinValue',0,'MaxValue',210);
bininfo(sc,'TmAtAddress')
```

```
ans=5x6 table
        Bin         Good    Bad     Odds        WOE        InfoValue
    ____________    ____    ___    _______    _________    __________

    '[0,30)'        330     154    2.1429      0.057722     0.0013305
    '[30,80)'       379     201    1.8856     -0.070187     0.0024086
    '[80,120)'       78      36    2.1667      0.068771    0.00044396
    '[120,210]'      16       6    2.6667       0.27641     0.0013301
    'Totals'        803     397    2.0227           NaN     0.0055131
```

Use the `modifybins` function to split the 2nd bin.

```
sc = modifybins(sc,'TmAtAddress','CutPoints',[cp(1) 50 cp(2:end)]);
bininfo(sc,'TmAtAddress')
```

```
ans=6x6 table
        Bin         Good    Bad     Odds        WOE        InfoValue
    ____________    ____    ___    _______    _________    __________

    '[0,30)'        330     154    2.1429      0.057722     0.0013305
    '[30,50)'       211     104    2.0288     0.0030488    2.4387e-06
    '[50,80)'       168      97     1.732      -0.15517      0.005449
    '[80,120)'       78      36    2.1667      0.068771    0.00044396
    '[120,210]'      16       6    2.6667       0.27641     0.0013301
    'Totals'        803     397    2.0227           NaN     0.0085559
```

Display bin information for the categorical predictor `ResStatus`.

```
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=4x6 table
        Bin          Good    Bad     Odds        WOE        InfoValue
    _____________    ____    ___    _______    _________    __________

    'Home Owner'     365     177    2.0621      0.019329     0.0001682
    'Tenant'         307     167    1.8383     -0.095564     0.0036638
```

```
    'Other'          131      53    2.4717      0.20049    0.0059418
    'Totals'         803     397    2.0227          NaN    0.0097738


cg=3x2 table
      Category      BinNumber
    ____________    _________

    'Home Owner'    1
    'Tenant'        2
    'Other'         3
```

Use the `modifybins` function to merge categories 2 and 3.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','CatGrouping',cg);
bininfo(sc,'ResStatus')
```

```
ans=3x6 table
      Bin        Good    Bad     Odds         WOE         InfoValue
    ________     ____    ___    ______    _________    __________

    'Group1'     365     177    2.0621     0.019329     0.0001682
    'Group2'     438     220    1.9909    -0.015827     0.00013772
    'Totals'     803     397    2.0227          NaN     0.00030592
```

Use the `modifybins` function to split bin 2 and put `Other` under bin 3.

```
cg.BinNumber(3) = 3;
sc = modifybins(sc,'ResStatus','CatGrouping',cg);
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=4x6 table
        Bin          Good    Bad     Odds         WOE         InfoValue
    ____________     ____    ___    ______    _________    _________

    'Home Owner'     365     177    2.0621     0.019329    0.0001682
    'Tenant'         307     167    1.8383    -0.095564    0.0036638
    'Other'          131      53    2.4717      0.20049    0.0059418
    'Totals'         803     397    2.0227          NaN    0.0097738


cg=3x2 table
      Category      BinNumber
```

```
    _____________    _________

    'Home Owner'     1
    'Tenant'         2
    'Other'          3
```

Modify Bin Labels

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Use `modifybins` to reset the minimum value and create three bins for the predictor `CustIncome` and display updated bin information.

```
sc = modifybins(sc,'CustIncome','MinValue',0,'CutPoints',[30000 50000]);
bi = bininfo(sc,'CustIncome')
```

```
bi=4x6 table
          Bin           Good    Bad     Odds        WOE        InfoValue
    ________________    ____    ___    _______    ________    _________

    '[0,30000)'          64      68    0.94118    -0.76504     0.070065
    '[30000,50000)'     628     296     2.1216    0.047762    0.0017421
    '[50000,Inf]'       111      33     3.3636      0.5086     0.028028
    'Totals'            803     397     2.0227         NaN     0.099836
```

Modify the bin labels and display updated bin information.

```
NewLabels = {'Up to 30k','30k to 50k','50k and more'};
sc = modifybins(sc,'CustIncome','BinLabels',NewLabels);
bi = bininfo(sc,'CustIncome')
```

```
bi=4x6 table
          Bin           Good    Bad     Odds        WOE        InfoValue
    ________________    ____    ___    _______    ________    _________

    'Up to 30k'          64      68    0.94118    -0.76504     0.070065
```

```
    '30k to 50k'      628     296     2.1216    0.047762    0.0017421
    '50k and more'    111      33     3.3636      0.5086     0.028028
    'Totals'          803     397     2.0227         NaN     0.099836
```

Bin labels should be the last bin-modification step. As in this example, user-defined bin labels often contain information about the cut points, minimum, or maximum values for numeric data, or information about category groupings for categorical data. To prevent situations where user-defined labels and cut points are inconsistent (and labels are misleading), the `creditscorecard` object overrides user-defined labels every time the bins are modified using `modifybins`.

To illustrate `modifybins` overriding user-defined labels every time the bins are modified, reset the first cut point to 31000 and display updated bin information. Note that the bin labels are reset to their default format and accurately reflect the change in the cut points.

```
sc = modifybins(sc,'CustIncome','CutPoints',[31000 50000]);
bi = bininfo(sc,'CustIncome')
```

```
bi=4x6 table
          Bin           Good    Bad     Odds        WOE       InfoValue
    ________________    ____    ___    _______    ________    _________

    '[0,31000)'          82      84    0.97619    -0.72852     0.079751
    '[31000,50000)'     610     280     2.1786    0.074251    0.0040364
    '[50000,Inf]'       111      33     3.3636      0.5086     0.028028
    'Totals'            803     397     2.0227         NaN      0.11182
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

sc — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`PredictorName` — Name of predictor
character vector

Name of predictor, specified as a character vector containing the name of the predictor. `PredictorName` is case-sensitive.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sc = modifybins(sc,PredictorName,'MinValue',10,'CutPoints', [23, 44, 66, 88])`

`MinValue` — Minimum acceptable value (numeric predictors only)
`-Inf` (default) | numeric

Minimum acceptable value, specified as a numeric value (for numeric predictors only). Values below this number are considered out of range.

Data Types: `double`

`MaxValue` — Maximum acceptable value (numeric predictors only)
`Inf` (default) | numeric

Maximum acceptable value, specified as a numeric value (for numeric predictors only). Values above this number are considered out of range.

Data Types: `double`

`CutPoints` — Split points between bins
each observed value of the predictor is placed in a separate bin (default) | nondecreasing numeric array

Split points between bins, specified using a nondecreasing numeric array. If there are `NumBins` bins, there are n = `NumBins` – `1` cut points so that C1, C2,..., Cn describe the bin boundaries with the following convention:

- The first bin includes any values >= `MinValue`, but < C1.
- The second bin includes any values >= C1, but < C2.
- The last bin includes any values >= C*n*, and <= `MaxValue`.

Note Cut points do not include `MinValue` or `MaxValue`.

By default, cut points are defined so that each observed value of the predictor is placed in a separate bin. If the sorted observed values are V1, …, V*M*, the default cut points are V2, …, V*M*, which define *M* bins.

Data Types: `double`

`CatGrouping` — Table with two columns named `Category` and `BinNumber`

each category is placed in a separate bin (default) | table with two columns named `Category` and `BinNumber`

Table with two columns named `Category` and `BinNumber` specified using a table, where the first column contains an exhaustive list of categories for the predictor, and the second column contains the bin number to which each category belongs.

By default, each category is placed in a separate bin. If the observed categories are `'Cat1'`…,`'Cat`*M*`'`, the default category grouping is as follows.

Category	BinNumber
`'Cat1'`	1
`'Cat2'`	2
...	...
`'Cat`*M*`'`	*M*

Data Types: `double`

`BinLabels` — Bin labels for each bin

automatically generated bin labels depending on the predictor's type (default) | cell array of character vectors

Bin labels for each bin, specified using a cell array of character vectors with bin label names. Bin labels are used to tag the bins in different object functions such as `bininfo`, `plotbins`, and `displaypoints`. A `creditscorecard` object automatically sets default

bins whenever bins are modified. The default format for bin labels depends on the predictor's type.

The format for `BinLabels` is:

- Numeric data — Before any manual or automatic modification of the predictor bins, there is a bin for each observed predictor value by default. In that case, the bin labels simply show the predictor values. Once the predictor bins have been modified, there are nondefault values for `MinValue` or `MaxValue`, or nondefault cut points C1, C2,..., C*n*. In that case, the bin labels are:

  - Bin 1 label: `'[MinValue, C1)'`
  - Bin 2 label: `'[C1, C2)'`
  - Last bin label: `'[Cn, MaxValue]'`

  For example, if there are three bins, `MinValue` is 0 and `MaxValue` is 40, and cut point 1 is 20 and cut point 2 is 30, then the corresponding three bin labels are:

  ```
  '[0,20)'
  '[20,30)'
  '[30,40]'
  ```

- Categorical data — For categorical data, before any modification of the predictor bins, there is one bin per category. In that case, the bin labels simply show the predictor categories. Once the bins have been modified, the labels are set to `'Group1'`, `'Group2'`, etc., for bin 1, bin 2, etc., respectively. For example, suppose that we have the following category grouping

Category	BinNumber
'Cat1'	1
'Cat2'	2
'Cat3'	2

  Bin 1 contains `'Cat1'` only and its bin label is set to `'Group1'`. Bin 2 contains `'Cat2'` and `'Cat3'` and its bin label is set to `'Group2'`.

Tip Using `BinLabels` should be the last step (if needed) in modifying bins. `BinLabels` definitions are overridden each time that the bins are modified using the `modifybins` or `autobinning` functions.

Data Types: `cell`

Output Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object. For more information on using the `creditscorecard` object, see `creditscorecard`.

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`autobinning` | `bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

“Case Study for a Credit Scorecard Analysis” on page 8-78
“Troubleshooting Credit Scorecard Results” on page 8-68
“Credit Scorecard Modeling Workflow” on page 8-62
“About Credit Scorecards” on page 8-57

Introduced in R2014b

modifypredictor

Set properties of credit scorecard predictors

Syntax

```
sc = modifypredictor(sc,PredictorName)
sc = modifypredictor(___,Name,Value)
```

Description

`sc = modifypredictor(sc,PredictorName)` sets the properties of the credit scorecard predictors.

`sc = modifypredictor(___,Name,Value)` sets the properties of the credit scorecard predictors using optional name-value pair arguments.

Examples

Modify a Predictor to Change the Predictor Type from Numeric to Categorical

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011). In practice, categorical data many times is represented with numeric values. To show the case where categorical data is given as numeric data, the data for the variable `'ResStatus'` is intentionally converted to numeric values.

```
load CreditCardData
data.ResStatus = double(data.ResStatus);
sc = creditscorecard(data,'IDVar','CustID')
```

```
sc =
  creditscorecard with properties:

                GoodLabel: 0
```

```
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {1x11 cell}
        NumericPredictors: {1x7 cell}
    CategoricalPredictors: {'EmpStatus'  'OtherCC'}
                  IDVar: 'CustID'
          PredictorVars: {1x9 cell}
                   Data: [1200x11 table]
```

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1x2 table
                PredictorType     LatestBinning
                _____________    _________________

    ResStatus    'Numeric'        'Original Data'


Stats=4x1 table
             Value
            _______

    Min           1
    Max           3
    Mean     1.7017
    Std     0.71863
```

Note that `'ResStatus'` appears as part of the `NumericPredictors` property. Assume that you want `'ResStatus'` to be treated as categorical data. For example, you may want to allow automatic binning algorithms to reorder the categories. Use `modifypredictor` to change the `'PredictorType'` of the `PredictorName` `'ResStatus'` from numeric to categorical.

```
sc = modifypredictor(sc,'ResStatus','PredictorType','Categorical')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {1x11 cell}
        NumericPredictors: {1x6 cell}
```

```
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                  IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                     Data: [1200x11 table]
```

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1x3 table
                PredictorType    Ordinal     LatestBinning
                _____________    _______    _________________

    ResStatus    'Categorical'    false      'Original Data'


Stats=3x1 table
          Count
          _____

    C1    542
    C2    474
    C3    184
```

Notice that `'ResStatus'` now appears as part of the `'Categorical'` predictors.

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`PredictorName` — Predictor name

character vector | cell array of character vectors

Predictor name, specified using a character vector or cell array of character vectors containing the names of the credit scorecard predictors. `PredictorName` is case-sensitive.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sc = modifypredictor(sc, {'CustAge','CustIncome'},'PredictorType','Categorical','Ordinal',true)`

`PredictorType` — Predictor type that one or more predictors are converted to
`''` no conversion occurs (default) | character vector with values `'Numeric'`, `'Categorical'`

Predictor type that one or more predictors are converted to, specified as a character vector. Possible values are:

- `''` — No conversion occurs.
- `'Numeric'` — The predictor data specified by `PredictorName` is converted to numeric.
- `'Categorical'` — The predictor data specified by `PredictorName` is converted to categorical.

Data Types: `char`

`Ordinal` — Indicator for whether predictors being converted to categorical are ordinal
`false` (default) | logical with values `true`, `false`

Indicator for whether predictors being converted to categorical or existing categorical predictors are treated as ordinal data, specified as a logical with values `true` or `false`.

Note This optional input parameter is only used for predictors of type `'Categorical'`.

Data Types: `logical`

Output Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object.

See Also

`bininfo` | `creditscorecard` | `modifybins` | `predictorinfo`

Introduced in R2015b

predictorinfo

Summary of credit scorecard predictor properties

Syntax

```
[T,Stats] = predictorinfo(sc,PredictorName)
```

Description

`[T,Stats] = predictorinfo(sc,PredictorName)` returns a summary of credit scorecard predictor properties and some basic predictor statistics.

Examples

Obtain Information for a Specified `PredictorName`

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
          PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Obtain the predictor statistics for the `PredictorName` of `CustAge`.

```
[T,Stats] = predictorinfo(sc,'CustAge')
```

```
T=1x2 table
               PredictorType     LatestBinning
               _____________    _______________

    CustAge    'Numeric'        'Original Data'


Stats=4x1 table
           Value
           ______

    Min      21
    Max      74
    Mean     45.174
    Std      9.8343
```

Obtain the predictor statistics for the `PredictorName` of `ResStatus`.

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1x3 table
                 PredictorType    Ordinal     LatestBinning
                 _____________    _______    _______________

    ResStatus    'Categorical'    false      'Original Data'


Stats=3x1 table
                  Count
                  _____

    Home Owner    542
    Tenant        474
    Other         184
```

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`PredictorName` — Predictor name

character vector

Predictor name, specified using a character vector containing the names of the credit scorecard predictor of interest. `PredictorName` is case-sensitive.

Data Types: `char`

Output Arguments

`T` — Summary information for specified predictor

table

Summary information for specified predictor, returned as table with the following columns:

- `'PredictorType'` — `'Numeric'` or `'Categorical'`.
- `'Ordinal'` — For categorical predictors, a boolean indicating whether it is ordinal.
- `'LatestBinning'` — Character vector indicating the last applied algorithm for the input argument `PredictorName`. The values are:
  - `'Original Data'` — When no binning is applied to the predictor.
  - `'Automatic / BinningName'` — Where `'BinningName'` is one of the following: `Monotone`, `Equal Width`, or `Equal Frequency`.
  - `'Manual'` — After each call of `modifybins`, where either `'CutPoints'`, `'CatGrouping'`, `'MinValue'`, or `'MaxValue'` are modified.

The predictor's name is used as a row name in the table that is returned.

`Stats` — Summary statistics for the input `PredictorName`

table

Summary statistics for the input `PredictorName`, returned as a table. The corresponding value is stored in the `'Value'` column.

The table's row names indicate the relevant statistics for numeric predictors:

- `'Min'` — Minimum value in the sample.
- `'Max'` — Maximum value in the sample.
- `'Mean'` — Mean value in the sample.
- `'Std'` — Standard deviation of the sample.

> **Note** For data types other than 'double' or 'single', numeric precision may be lost for the standard deviation. Data types other than 'double' or 'single' are cast as 'double' before computing the standard deviation.

For categorical predictors, the row names contain the names of the categories, with corresponding total count in the `'Count'` column.

See Also

`bininfo` | `creditscorecard` | `modifybins` | `modifypredictor`

Introduced in R2015b

bininfo

Return predictor's bin information

Syntax

```
bi = bininfo(sc,PredictorName)
bi = bininfo(___,Name,Value)

[bi,bm] = bininfo(sc,PredictorName,Name,Value)

[bi,bm,mv] = bininfo(sc,PredictorName,Name,Value)
```

Description

`bi = bininfo(sc,PredictorName)` returns information at bin level, such as frequencies of "Good," "Bad," and bin statistics for the predictor specified in `PredictorName`.

`bi = bininfo(___,Name,Value)` adds optional name-value arguments.

`[bi,bm] = bininfo(sc,PredictorName,Name,Value)` adds optional name-value arguments.`bininfo` also optionally returns the binning map (`bm`) or bin rules in the form of a vector of cut points for numeric predictors, or a table of category groupings for categorical predictors.

`[bi,bm,mv] = bininfo(sc,PredictorName,Name,Value)` returns information at bin level, such as frequencies of "Good," "Bad," and bin statistics for the predictor specified in `PredictorName` using optional name-value pair arguments. `bininfo` optionally returns the binning map or bin rules in the form of a vector of cut points for numeric predictors, or a table of category groupings for categorical predictors. In addition, optional name-value pair arguments `mv` returns a numeric array containing the minimum and maximum values, as set (or defined) by the user. The `mv` output argument is set to an empty array for categorical predictors.

Examples

Display Bin Information Using Default Options

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Display bin information for the categorical predictor `ResStatus`.

```
bi = bininfo(sc,'ResStatus')
```

```
bi=4x6 table null
        Bin          Good    Bad     Odds        WOE        InfoValue
    _____________    ____    ___    _______    _________    _________

    'Home Owner'     365     177    2.0621      0.019329    0.0001682
    'Tenant'         307     167    1.8383     -0.095564    0.0036638
    'Other'          131      53    2.4717       0.20049    0.0059418
    'Totals'         803     397    2.0227           NaN    0.0097738
```

Display Bin Information For a creditscorecard Object Containing Weights

Use the `CreditCardData.mat` file to load the data (`dataWeights`) that contains a column (`RowWeights`) for the weights (using a dataset from Refaat 2011).

```
load CreditCardData
```

Create a `creditscorecard` object using the optional name-value pair argument for `'WeightsVar'`.

```
sc = creditscorecard(dataWeights,'WeightsVar','RowWeights')
```

```
sc =

  creditscorecard with properties:
```

```
                GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: 'RowWeights'
                VarNames: {1x12 cell}
       NumericPredictors: {1x7 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: ''
           PredictorVars: {1x10 cell}
                    Data: [1200x12 table]
```

Display bin information for the numerical predictor `'CustIncome'`. When the optional name-value pair argument `'WeightsVar'` is used to specify observation (sample) weights, the `bi` table contains weighted counts.

```
bi = bininfo(sc,'CustIncome');
bi(1:10,:)
```

```
ans =

  10x6 table

      Bin       Good       Bad       Odds       WOE       InfoValue
    _______    _______    _______    _______    ________    __________

    '18000'    0.94515      1.496    0.63179     -1.1667     0.0059198
    '19000'    0.47588    0.80569    0.59065     -1.2341     0.0034716
    '20000'     2.1671     1.4636     1.4806    -0.31509    0.00061392
    '21000'     3.2522    0.88064      3.693     0.59889     0.0021303
    '22000'     1.5438     1.2714     1.2142    -0.51346     0.0012913
    '23000'      1.787     2.7529    0.64913     -1.1397      0.010509
    '24000'     3.4111     2.2538     1.5135    -0.29311    0.00082663
    '25000'     2.2333     6.1383    0.36383     -1.7186      0.042642
    '26000'     2.1246     4.4754    0.47474     -1.4525      0.024526
    '27000'     3.1058      3.528    0.88032    -0.83501     0.0082144
```

Display Bin Information Using Name-Value Arguments

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Display customized bin information for the categorical predictor `ResStatus`, keeping only the WOE column.

```
bi = bininfo(sc,'ResStatus','Statistics','WOE')
```

```
bi=4x4 table
        Bin          Good    Bad       WOE
    _____________    ____    ___    _________

    'Home Owner'     365     177     0.019329
    'Tenant'         307     167    -0.095564
    'Other'          131      53      0.20049
    'Totals'         803     397          NaN
```

Display customized bin information for the categorical predictor `ResStatus`, keeping only the Odds and WOE columns, without the `Totals` row.

```
bi = bininfo(sc,'ResStatus','Statistics',{'Odds','WOE'},'Totals','Off')
```

```
bi=3x5 table
        Bin          Good    Bad     Odds        WOE
    _____________    ____    ___    ______    _________

    'Home Owner'     365     177    2.0621     0.019329
    'Tenant'         307     167    1.8383    -0.095564
    'Other'          131      53    2.4717      0.20049
```

Display Bin Information and Binning Map for Categorical Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The binning map or rules for categorical data are summarized in a "category grouping" table, returned as an optional output. By default, each category is placed in a separate bin. Here is the information for the predictor `ResStatus`.

```
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=4x6 table
        Bin          Good    Bad     Odds        WOE        InfoValue
    _____________    ____    ___    _______    _________    _________

    'Home Owner'     365     177    2.0621      0.019329    0.0001682
    'Tenant'         307     167    1.8383     -0.095564    0.0036638
    'Other'          131      53    2.4717       0.20049    0.0059418
    'Totals'         803     397    2.0227           NaN    0.0097738


cg=3x2 table
      Category       BinNumber
    _____________    _________

    'Home Owner'     1
    'Tenant'         2
    'Other'          3
```

To group categories `Tenant` and `Other`, modify the category grouping table `cg` so that the bin number for `Other` is the same as the bin number for `Tenant`. Then use the `modifybins` function to update the scorecard.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','CatGrouping',cg);
```

Display the updated bin information. The bin labels have been updated and that the bin membership information is contained in the category grouping `cg`.

```
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=3x6 table
      Bin         Good    Bad     Odds        WOE        InfoValue
    ________    ____    ___    _______    _________    __________

    'Group1'    365     177    2.0621      0.019329     0.0001682
    'Group2'    438     220    1.9909     -0.015827    0.00013772
    'Totals'    803     397    2.0227           NaN    0.00030592
```

```
cg=3x2 table
      Category       BinNumber
    _____________    _________

    'Home Owner'     1
    'Tenant'         2
    'Other'          2
```

Display Bin Information and Binning Map for Numeric Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

The predictor `CustIncome` is numeric. By default, each value of the predictor is placed in a separate bin.

```
bi = bininfo(sc,'CustIncome')
```

```
bi=46x6 table
      Bin       Good     Bad      Odds          WOE         InfoValue
    ________    ____    ____    ________    __________    __________

    '18000'      2        3     0.66667       -1.1099      0.0056227
    '19000'      1        2         0.5       -1.3976      0.0053002
    '20000'      4        2           2     -0.011271     6.3641e-07
    '21000'      6        3           2     -0.011271     9.5462e-07
    '22000'      4        2           2     -0.011271     6.3641e-07
    '23000'      4        4           1      -0.70442      0.0035885
    '24000'      5        5           1      -0.70442      0.0044856
    '25000'      4        9     0.44444       -1.5153       0.026805
    '26000'      4       11     0.36364        -1.716       0.038999
    '27000'      6        6           1      -0.70442      0.0053827
    '28000'     13       11      1.1818      -0.53736      0.0061896
    '29000'     11       10         1.1      -0.60911      0.0069988
    '30000'     18       16       1.125      -0.58664       0.010493
    '31000'     24        8           3       0.39419      0.0038382
    '32000'     21       15         1.4      -0.36795      0.0042797
```

```
    '33000'    35      19      1.8421    -0.093509    0.00039951
```

Reduce the number of bins using the `autobinning` function (the `modifybins` function can also be used).

```
sc = autobinning(sc,'CustIncome');
```

Display the updated bin information. The binning map or rules for numeric data are summarized as "cut points," returned as an optional output (`cp`).

```
[bi,cp] = bininfo(sc,'CustIncome')
```

```
bi=8x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,29000)'       53      58    0.91379     -0.79457       0.06364
    '[29000,33000)'      74      49     1.5102     -0.29217     0.0091366
    '[33000,35000)'      68      36     1.8889     -0.06843    0.00041042
    '[35000,40000)'     193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'      68      34          2    -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579     0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972      0.041657
    'Totals'            803     397     2.0227          NaN       0.12285


cp =

       29000
       33000
       35000
       40000
       42000
       47000
```

Manually remove the second cut point (the boundary between the second and third bins) to merge bins two and three. Use the `modifybins` function to update the scorecard.

```
cp(2) = [];
sc = modifybins(sc,'CustIncome','CutPoints',cp,'MinValue',0);
```

Display the updated bin information.

```
[bi,cp,mv] = bininfo(sc,'CustIncome')
```

```
bi=7x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[0,29000)'          53      58    0.91379     -0.79457       0.06364
    '[29000,35000)'     142      85     1.6706     -0.19124     0.0071274
    '[35000,40000)'     193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'      68      34          2    -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579     0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972      0.041657
    'Totals'            803     397     2.0227          NaN       0.12043
```

```
cp =

       29000
       35000
       40000
       42000
       47000
```

```
mv =

     0   Inf
```

Note, it is recommended to avoid having bins with frequencies of zero because they lead to infinite or undefined (`NaN`) statistics. Use the `modifybins` or `autobinning` functions to modify bins.

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`PredictorName` — Predictor name
character vector

Predictor name, specified using a character vector containing the name of the predictor. `PredictorName` is case-sensitive.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `bi = bininfo(sc, PredictorName,'Statistics','WOE','Totals','On')`

`Statistics` — List of statistics to include for bin information
`{'Odds','WOE','InfoValue'}` (default) | character vector with values `'Odds'`, `'WOE'`, `'InfoValue','Entropy'` | cell array of character vectors with values `'Odds'`, `'WOE'`, `'InfoValue'`, `'Entropy'`

List of statistics to include in the bin information, specified as a character vector or a cell array of character vectors. For more information, see “Statistics for a Credit Scorecard” on page 18-2156. Possible values are:

- `'Odds'` — Odds information is the ratio of “Goods” over “Bads.”
- `'WOE'` — Weight of Evidence. The WOE Statistic measures the deviation between the distribution of “Goods” and “Bads.”
- `'InfoValue'` — Information value. Closely tied to the WOE, it is a statistic used to determine how strong a predictor is to use in the fitting model. It measures how strong the deviation is between the distributions of “Goods” and “Bads.” However, bins with only “Good” or only “Bad” observations do lead to an infinite Information Value. Consider modifying the bins in those cases by using `modifybins` or `autobinning`.

- `'Entropy'` — Entropy is a measure of unpredictability contained in the bins. The more the number of “Goods” and “Bads” differ within the bins, the lower the entropy.

Note Avoid having bins with frequencies of zero because they lead to infinite or undefined (`NaN`) statistics. Use `modifybins` or `autobinning` to modify bins.

Data Types: `char` | `cell`

`Totals` — Indicator to include row of totals at bottom information table

`'On'` (default) | character vector with values `'On'`, `'Off'`

Indicator to include a row of totals at the bottom of the information table, specified as a character vector with values `On` or `Off`.

Data Types: `char`

Output Arguments

`bi` — Bin information

table

Bin information, returned as a table. The bin information table contains one row per bin and a row of totals. The columns contain bin descriptions, frequencies of “Good” and “Bad,” and bin statistics. Avoid having bins with frequencies of zero because they lead to infinite or undefined (`NaN`) statistics. Use `modifybins` or `autobinning` to modify bins.

Note When creating the `creditscorecard` object with `creditscorecard`, if the optional name-value pair argument `WeightsVar` was used to specify observation (sample) weights, then the`bi` table contains weighted counts.

`bm` — Binning map or rules

vector of cut points for numeric predictors | table of category groupings for categorical predictors

Binning map or rules, returned as a vector of cut points for numeric predictors, or a table of category groupings for categorical predictors. For more information, see `modifybins`.

`mv` — Binning minimum and maximum values

numeric array

Binning minimum and maximum values (as set or defined by the user), returned as a numeric array. The `mv` output argument is set to an empty array for categorical predictors.

Definitions

Statistics for a Credit Scorecard

Weight of Evidence (WOE) is a measure of the difference of the distribution of “Goods” and “Bads” within a bin.

Suppose the predictor's data takes on *M* possible values b1, ..., b*M*. For binned data, *M* is a small number. The response takes on two values, “Good” and “Bad.” The frequency table of the data is given by:

	Good	Bad	Total
b1:	n11	n12	n1
b2:	n21	n22	n2
b*M*:	n*M*1	n*M*2	n*M*
Total:	nGood	nBad	nTotal

The Weight of Evidence (WOE) is defined for each data value `bi` as

```
WOE(i) = log((ni1/nGood)/(ni2/nBad)).
```

If you define

```
pGood(i) = ni1/nGood, pBad(i) = ni2/nBad
```

then `pGood`(*i*) is the proportion of “Good” observations that take on the value `bi`, and similarly for `pBad`(*i*). In other words, `pGood`(*i*) gives the distribution of good observations over the *M* observed values of the predictor, and similarly for `pBad`(*i*). With this, an equivalent formula for the WOE is

```
WOE(i) = log(pGood(i)/pBad(i)).
```

Using the same frequency table, the odds for row *i* are defined as

```
Odds(i) = ni1 / ni2,
```

and the odds for the sample are defined as

```
OddsTotal = nGood / nBad.
```

For each row *i*, you can also compute its contribution to the total Information Value, given by

```
InfoValue(i) = (pGood(i) - pBad(i)) * WOE(i),
```

and the total Information Value is simply the sum of all the `InfoValue`(*i*) terms. (A `nansum` is returned to discard contributions from rows with no observations at all.)

Likewise, for each row *i*, we can compute its contribution to the total Entropy, given by

```
 Entropy(i) = -1/log(2)*(ni1/ni*log(ni1/ni)+ni2/ni*log(ni2/ni),
```

and the total Entropy is simply the weighted sum of the row entropies,

```
Entropy = sum(ni/nTotal * Entropy(i)), i = 1...M
```

Using `bininfo` with Weights

When observation weights are defined using the optional `WeightsVar` argument when creating a `creditscorecard` object, instead of counting the rows that are good or bad in each bin, the `bininfo` function accumulates the weight of the rows that are good or bad in each bin.

The "frequencies" reported are no longer the basic "count" of rows, but the "cumulative weight" of the rows that are good or bad and fall in a particular bin. Once these "weighted frequencies" are known, all other relevant statistics (`Good`, `Bad`, `Odds`, `WOE`, and `InfoValue`) are computed with the usual formulas. For more information, see "Credit Scorecard Modeling Using Observation Weights" on page 8-65.

References

[1] Anderson, R. *The Credit Scoring Toolkit.* Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

`autobinning` | `bindata` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

“Case Study for a Credit Scorecard Analysis” on page 8-78
“Troubleshooting Credit Scorecard Results” on page 8-68
“Credit Scorecard Modeling Workflow” on page 8-62
“About Credit Scorecards” on page 8-57
“Credit Scorecard Modeling Using Observation Weights” on page 8-65

Introduced in R2014b

autobinning

Perform automatic binning of given predictors

Syntax

```
sc = autobinning(sc)
sc = autobinning(sc,PredictorNames)
sc = autobinning(___,Name,Value)
```

Description

`sc = autobinning(sc)` performs automatic binning of all predictors.

Automatic binning finds binning maps or rules to bin numeric data and to group categories of categorical data. The binning rules are stored in the `creditscorecard` object. To apply the binning rules to the `creditscorecard` object data, or to a new dataset, use `bindata`.

`sc = autobinning(sc,PredictorNames)` performs automatic binning of the predictors given in `PredictorNames`.

Automatic binning finds binning maps or rules to bin numeric data and to group categories of categorical data. The binning rules are stored in the `creditscorecard` object. To apply the binning rules to the `creditscorecard` object data, or to a new dataset, use `bindata`.

`sc = autobinning(___,Name,Value)` performs automatic binning of the predictors given in `PredictorNames` using optional name-value pair arguments. See the name-value argument `Algorithm` for a description of the supported binning algorithms.

Automatic binning finds binning maps or rules to bin numeric data and to group categories of categorical data. The binning rules are stored in the `creditscorecard` object. To apply the binning rules to the `creditscorecard` object data, or to a new dataset, use `bindata`.

Examples

Perform Automatic Binning Using the Defaults

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning using the default options. By default, autobinning bins all predictors and uses the `Monotone` algorithm.

```
sc = autobinning(sc);
```

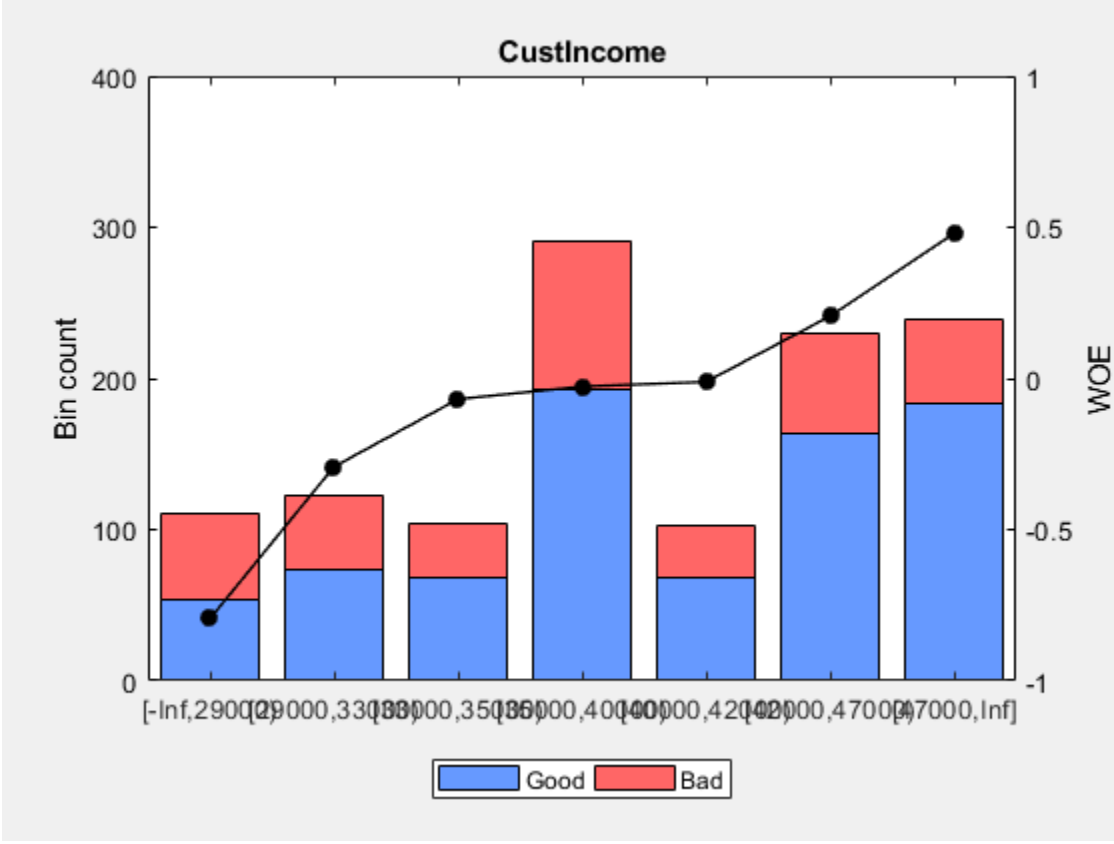
Use `bininfo` to display the binned data for the predictor `CustIncome`.

```
bi = bininfo(sc, 'CustIncome')
```

```
bi=8x6 table
          Bin           Good    Bad     Odds        WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,29000)'       53      58    0.91379     -0.79457       0.06364
    '[29000,33000)'      74      49     1.5102     -0.29217     0.0091366
    '[33000,35000)'      68      36     1.8889     -0.06843    0.00041042
    '[35000,40000)'     193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'      68      34          2    -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579     0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972      0.041657
    'Totals'            803     397     2.0227          NaN       0.12285
```

Use `plotbins` to display the histogram and WOE curve for the predictor `CustIncome`.

```
plotbins(sc,'CustIncome')
```



Perform Automatic Binning with a Named Predictor Using the Defaults

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the predictor `CustIncome` using the default options. By default, `autobinning` uses the `Monotone` algorithm.

```
sc = autobinning(sc,'CustIncome');
```

Use `bininfo` to display the binned data.

```
bi = bininfo(sc, 'CustIncome')
```

```
bi=8x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,29000)'       53      58    0.91379     -0.79457       0.06364
    '[29000,33000)'      74      49     1.5102     -0.29217     0.0091366
    '[33000,35000)'      68      36     1.8889     -0.06843    0.00041042
    '[35000,40000)'     193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'      68      34          2    -0.011271    1.0819e-05
    '[42000,47000)'     164      66     2.4848      0.20579     0.0078175
    '[47000,Inf]'       183      56     3.2679      0.47972      0.041657
    'Totals'            803     397     2.0227          NaN       0.12285
```

Perform Automatic Binning Using Two Name-Value Pair Arguments

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the predictor `CustIncome` using the `Monotone` algorithm with the initial number of bins set to 20. This example explicitly sets both the `Algorithm` and the `AlgorithmOptions` name-value arguments.

```
AlgoOptions = {'InitialNumBins',20};
sc = autobinning(sc,'CustIncome','Algorithm','Monotone','AlgorithmOptions',...
     AlgoOptions);
```

Use `bininfo` to display the binned data. Here, the cut points, which delimit the bins, are also displayed.

```
[bi,cp] = bininfo(sc,'CustIncome')
```

```
bi=11x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________
```

```
    '[-Inf,19000)'       2       3    0.66667      -1.1099     0.0056227
    '[19000,29000)'     51      55    0.92727     -0.77993      0.058516
    '[29000,31000)'     29      26     1.1154     -0.59522      0.017486
    '[31000,34000)'     80      42     1.9048    -0.060061     0.0003704
    '[34000,35000)'     33      17     1.9412    -0.041124     7.095e-05
    '[35000,40000)'    193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'     68      34          2    -0.011271    1.0819e-05
    '[42000,43000)'     39      16     2.4375      0.18655      0.001542
    '[43000,47000)'    125      50        2.5      0.21187     0.0062972
    '[47000,Inf]'      183      56     3.2679      0.47972      0.041657
    'Totals'           803     397     2.0227          NaN       0.13175
```

```
cp =

       19000
       29000
       31000
       34000
       35000
       40000
       42000
       43000
       47000
```

Perform Automatic Binning Using Multiple Name-Value Pair Arguments

This example shows how to use the `autobinning` default `Monotone` algorithm and the `AlgorithmOptions` name-value pair arguments associated with the `Monotone` algorithm. The `AlgorithmOptions` for the `Monotone` algorithm are three name-value pair parameters: `'InitialNumBins'`, `'Trend'`, and `'SortCategories'`. `'InitialNumBins'` and `'Trend'` are applicable for numeric predictors and `'Trend'` and `'SortCategories'` are applicable for categorical predictors.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning for the numeric predictor `CustIncome` using the `Monotone` algorithm with 20 bins. This example explicitly sets both the `Algorithm` argument and the `AlgorithmOptions` name-value arguments for `'InitialNumBins'` and `'Trend'`.

```
AlgoOptions = {'InitialNumBins',20,'Trend','Increasing'};

sc = autobinning(sc,'CustIncome','Algorithm','Monotone',...
    'AlgorithmOptions',AlgoOptions);
```

Use `bininfo` to display the binned data.

```
bi = bininfo(sc,'CustIncome')
```

```
bi=11x6 table
          Bin           Good    Bad     Odds        WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,19000)'       2       3    0.66667      -1.1099     0.0056227
    '[19000,29000)'     51      55    0.92727     -0.77993      0.058516
    '[29000,31000)'     29      26     1.1154     -0.59522      0.017486
    '[31000,34000)'     80      42     1.9048    -0.060061     0.0003704
    '[34000,35000)'     33      17     1.9412    -0.041124     7.095e-05
    '[35000,40000)'    193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'     68      34          2    -0.011271    1.0819e-05
    '[42000,43000)'     39      16     2.4375      0.18655      0.001542
    '[43000,47000)'    125      50        2.5      0.21187     0.0062972
    '[47000,Inf]'      183      56     3.2679      0.47972      0.041657
    'Totals'           803     397     2.0227          NaN       0.13175
```

Perform Automatic Binning for Multiple Predictors

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning for the predictor `CustIncome` and `CustAge` using the default `Monotone` algorithm with `AlgorithmOptions` for `InitialNumBins` and `Trend`.

```
AlgoOptions = {'InitialNumBins',20,'Trend','Increasing'};

sc = autobinning(sc,{'CustAge','CustIncome'},'Algorithm','Monotone',...
    'AlgorithmOptions',AlgoOptions);
```

Use `bininfo` to display the binned data.

```
bi1 = bininfo(sc, 'CustIncome')
```

```
bi1=11x6 table
          Bin           Good    Bad     Odds         WOE        InfoValue
    ________________    ____    ___    _______    _________    __________

    '[-Inf,19000)'       2       3    0.66667      -1.1099     0.0056227
    '[19000,29000)'     51      55    0.92727     -0.77993      0.058516
    '[29000,31000)'     29      26     1.1154     -0.59522      0.017486
    '[31000,34000)'     80      42     1.9048    -0.060061     0.0003704
    '[34000,35000)'     33      17     1.9412    -0.041124     7.095e-05
    '[35000,40000)'    193      98     1.9694    -0.026696    0.00017359
    '[40000,42000)'     68      34          2    -0.011271    1.0819e-05
    '[42000,43000)'     39      16     2.4375      0.18655      0.001542
    '[43000,47000)'    125      50        2.5      0.21187     0.0062972
    '[47000,Inf]'      183      56     3.2679      0.47972      0.041657
    'Totals'           803     397     2.0227          NaN       0.13175
```

```
bi2 = bininfo(sc, 'CustAge')
```

```
bi2=8x6 table
        Bin         Good    Bad     Odds         WOE        InfoValue
    ____________    ____    ___    ______    _________    __________

    '[-Inf,35)'      93      76    1.2237     -0.50255      0.038003
    '[35,40)'       114      71    1.6056      -0.2309     0.0085141
    '[40,42)'        52      30    1.7333     -0.15437     0.0016687
    '[42,44)'        58      32    1.8125     -0.10971    0.00091888
    '[44,47)'        97      51     1.902    -0.061533    0.00047174
    '[47,62)'       333     130    2.5615      0.23619      0.020605
    '[62,Inf]'       56       7         8        1.375      0.071647
    'Totals'        803     397    2.0227          NaN       0.14183
```

Perform Automatic Binning for a Categorical Predictor Using the Defaults

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data);
```

Perform automatic binning for the predictor that is a categorical predictor called `ResStatus` using the default options. By default, `autobinning` uses the `Monotone` algorithm.

```
sc = autobinning(sc,'ResStatus');
```

Use `bininfo` to display the binned data.

```
bi = bininfo(sc, 'ResStatus')
```

```
bi=4x6 table null
        Bin          Good    Bad     Odds        WOE        InfoValue
    _____________    ____    ___    ______    _________    _________

    'Tenant'         307     167    1.8383    -0.095564    0.0036638
    'Home Owner'     365     177    2.0621     0.019329    0.0001682
    'Other'          131      53    2.4717      0.20049    0.0059418
    'Totals'         803     397    2.0227          NaN    0.0097738
```

Perform Automatic Binning for a Categorical Predictor Using Name-Value Pair Arguments

This example shows how to modify the data (for this example only) to illustrate binning categorical predictors using the `Monotone` algorithm.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
```

Add two new categories and updating the response variable.

```
newdata = data;
rng('default'); %for reproducibility
```

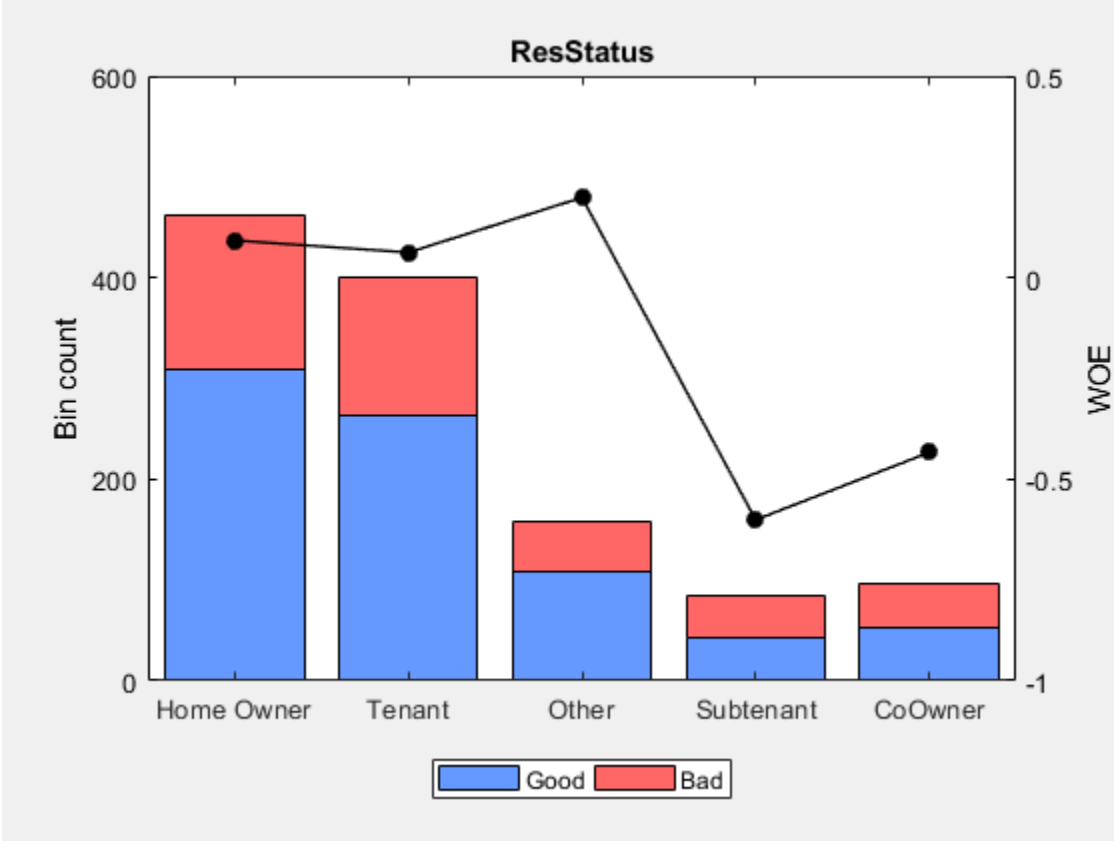
```
Predictor = 'ResStatus';
Status    = newdata.status;
NumObs    = length(newdata.(Predictor));
Ind1 = randi(NumObs,100,1);
Ind2 = randi(NumObs,100,1);
newdata.(Predictor)(Ind1) = 'Subtenant';
newdata.(Predictor)(Ind2) = 'CoOwner';
Status(Ind1) = randi(2,100,1)-1;
Status(Ind2) = randi(2,100,1)-1;

newdata.status = Status;
```

Update the `creditscorecard` object using the `newdata` and plot the bins for a later comparison.

```
scnew = creditscorecard(newdata,'IDVar','CustID');
[bi,cg] = bininfo(scnew,Predictor)
```

```
bi=6x6 table
        Bin          Good    Bad     Odds       WOE        InfoValue
    _____________    ____    ___    ______    ________    _________

    'Home Owner'     308     154         2    0.092373    0.0032392
    'Tenant'         264     136    1.9412     0.06252    0.0012907
    'Other'          109      49    2.2245     0.19875    0.0050386
    'Subtenant'       42      42         1    -0.60077     0.026813
    'CoOwner'         52      44    1.1818    -0.43372     0.015802
    'Totals'         775     425    1.8235         NaN     0.052183


cg=5x2 table
      Category       BinNumber
    _____________    _________

    'Home Owner'     1
    'Tenant'         2
    'Other'          3
    'Subtenant'      4
    'CoOwner'        5
```

```
plotbins(scnew,Predictor)
```



Perform automatic binning for the categorical `Predictor` using the default `Monotone` algorithm with the `AlgorithmOptions` name-value pair arguments for `'SortCategories'` and `'Trend'`.

```
AlgoOptions = {'SortCategories','Goods','Trend','Increasing'};

scnew = autobinning(scnew,Predictor,'Algorithm','Monotone',...
    'AlgorithmOptions',AlgoOptions);
```

Use `bininfo` to display the bin information. The second output parameter `'cg'` captures the bin membership, which is the bin number that each group belongs to.

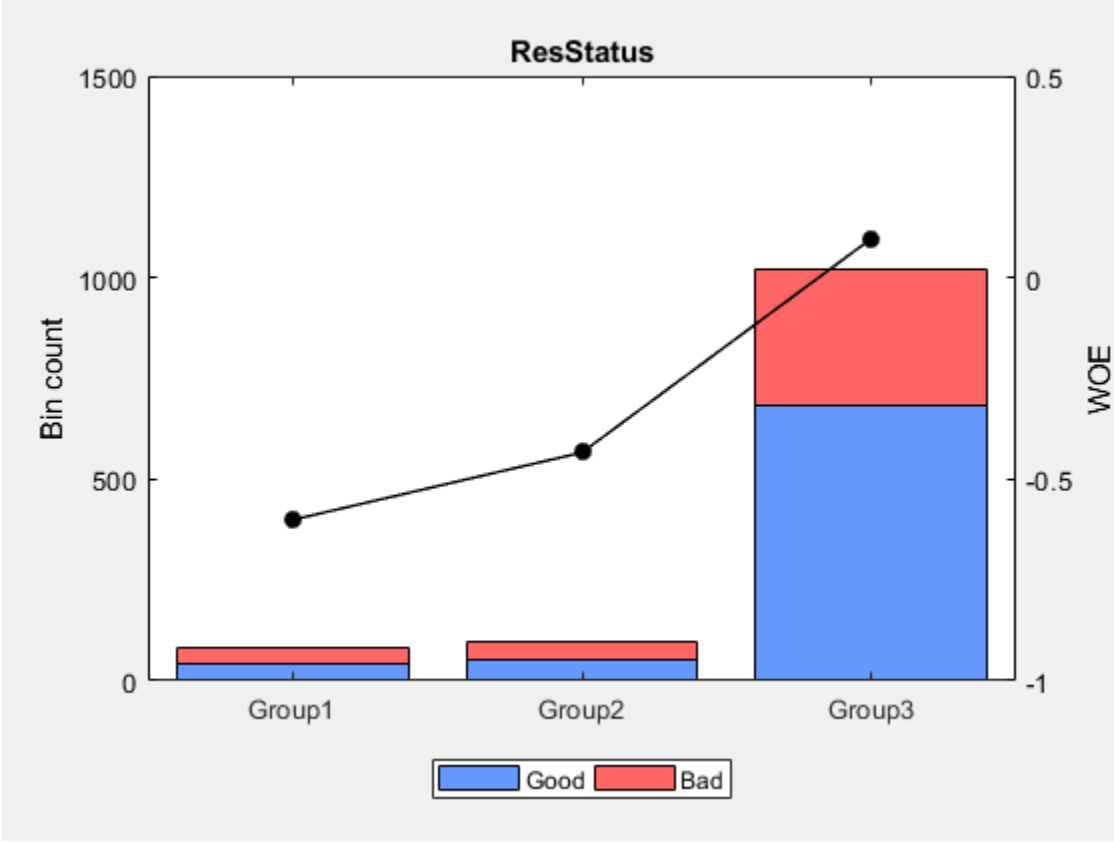
```
[bi,cg] = bininfo(scnew,Predictor)
```

```
bi=4x6 table
      Bin        Good    Bad     Odds       WOE        InfoValue
    _________    ____    ___    _______    ________    _________

    'Group1'      42      42          1    -0.60077     0.026813
    'Group2'      52      44     1.1818    -0.43372     0.015802
    'Group3'     681     339     2.0088    0.096788    0.0078459
    'Totals'     775     425     1.8235         NaN      0.05046
```

```
cg=5x2 table
      Category       BinNumber
    _____________    _________

    'Subtenant'      1
    'CoOwner'        2
    'Other'          3
    'Tenant'         3
    'Home Owner'     3
```

Plot bins and compare with the histogram plotted pre-binning.

```
plotbins(scnew,Predictor)
```



- "Case Study for a Credit Scorecard Analysis" on page 8-78
- "Troubleshooting Credit Scorecard Results" on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. Use `creditscorecard` to create a `creditscorecard` object.

`PredictorNames` — Predictor or predictors names to automatically bin
character vector | cell array of character vectors

Predictor or predictors names to automatically bin, specified as a character vector or a cell array of character vectors containing the name of the predictor or predictors. `PredictorNames` are case-sensitive and when no `PredictorNames` are defined, all predictors in the `PredictorVars` property of the `creditscorecard` object are binned.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sc = autobinning(sc,'Algorithm','EqualFrequency')`

`Algorithm` — Algorithm selection
`'Monotone'` (default) | character vector with values `'Monotone'`,`'EqualFrequency'`, `'EqualWidth'`

Algorithm selection, specified using a character vector indicating which algorithm to use. The same algorithm is used for all predictors in `PredictorNames`. Possible values are:

- `'Monotone'` — (default) Monotone Adjacent Pooling Algorithm (MAPA), also known as Maximum Likelihood Monotone Coarse Classifier (MLMCC). Supervised optimal binning algorithm that aims to find bins with a monotone Weight-Of-Evidence (WOE) trend. This algorithm assumes that only neighboring attributes can be grouped. Thus, for categorical predictors, categories are sorted before applying the algorithm (see `'SortCategories'` option for `AlgorithmOptions`). For more information, see “Monotone” on page 18-2175.
- `'EqualFrequency'` — Unsupervised algorithm that divides the data into a predetermined number of bins that contain approximately the same number of observations. This algorithm is also known as “equal height” or “equal depth.” For categorical predictors, categories are sorted before applying the algorithm (see `'SortCategories'` option for `AlgorithmOptions`). For more information, see “Equal Frequency” on page 18-2177.

- `'EqualWidth'` — Unsupervised algorithm that divides the range of values in the domain of the predictor variable into a predetermined number of bins of "equal width." For numeric data, the width is measured as the distance between bin edges. For categorical data, width is measured as the number of categories within a bin. For categorical predictors, categories are sorted before applying the algorithm (see `'SortCategories'` option for `AlgorithmOptions`). For more information, see "Equal Width" on page 18-2178.

Data Types: `char`

`AlgorithmOptions` — Algorithm options for selected `Algorithm`

`{'InitialNumBins',10,'Trend','Auto','SortCategories','Odds'}` for `Monotone` (default) | cell array with `{'OptionName',`*OptionValue*`}` for `Algorithm` options

Algorithm options for the selected `Algorithm`, specified using a cell array. Possible values are:

- For `Monotone` algorithm:
  - `{'InitialNumBins',`*n*`}` — Initial number (*n*) of bins (default is 10). `'InitialNumBins'` must be an integer > `2`. Used for numeric predictors only.
  - `{'Trend','TrendOption'}` — Determines whether the Weight-Of-Evidence (WOE) monotonic trend is expected to be increasing or decreasing. The values for `'TrendOption'` are:
    - `'Auto'` — (Default) Automatically determines if the WOE trend is increasing or decreasing.
    - `'Increasing'` — Look for an increasing WOE trend.
    - `'Decreasing'` — Look for a decreasing WOE trend.

    The value of the optional input parameter `'Trend'` does not necessarily reflect that of the resulting WOE curve. The parameter `'Trend'` tells the algorithm to "look for" an increasing or decreasing trend, but the outcome may not show the desired trend. For example, the algorithm cannot find a decreasing trend when the data actually has an increasing WOE trend. For more information on the `'Trend'` option, see "Monotone" on page 18-2175.
  - `{'SortCategories','SortOption'}` — Used for categorical predictors only. Used to determine how the predictor categories are sorted as a preprocessing step before applying the algorithm. The values of `'SortOption'` are:

- `'Odds'` — (default) The categories are sorted by order of increasing values of odds, defined as the ratio of "Good" to "Bad" observations, for the given category.
- `'Goods'` — The categories are sorted by order of increasing values of "Good."
- `'Bads'` — The categories are sorted by order of increasing values of "Bad."
- `'Totals'` — The categories are sorted by order of increasing values of total number of observations ("Good" plus "Bad").
- `'None'` — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm. (The existing order of the categories can be seen in the category grouping optional output from `bininfo`.)

For more information, see Sort Categories on page 18-2179

- For `EqualFrequency` algorithm:
  - `{'NumBins',n}` — Specifies the desired number (*n*) of bins. The default is `{'NumBins',5}` and the number of bins must be a positive number.
  - `{'SortCategories','SortOption'}` — Used for categorical predictors only. Used to determine how the predictor categories are sorted as a preprocessing step before applying the algorithm. The values of `'SortOption'` are:
    - `'Odds'` — (default) The categories are sorted by order of increasing values of odds, defined as the ratio of "Good" to "Bad" observations, for the given category.
    - `'Goods'` — The categories are sorted by order of increasing values of "Good."
    - `'Bads'` — The categories are sorted by order of increasing values of "Bad."
    - `'Totals'` — The categories are sorted by order of increasing values of total number of observations ("Good" plus "Bad").
    - `'None'` — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm. (The existing order of the categories can be seen in the category grouping optional output from `bininfo`.)

  For more information, see Sort Categories on page 18-2179
- For `EqualWidth` algorithm:
  - `{'NumBins',n}` — Specifies the desired number (*n*) of bins. The default is `{'NumBins',5}` and the number of bins must be a positive number.

- `{'SortCategories','SortOption'}` — Used for categorical predictors only. Used to determine how the predictor categories are sorted as a preprocessing step before applying the algorithm. The values of `'SortOption'` are:
  - `'Odds'` — (default) The categories are sorted by order of increasing values of odds, defined as the ratio of "Good" to "Bad" observations, for the given category.
  - `'Goods'` — The categories are sorted by order of increasing values of "Good."
  - `'Bads'` — The categories are sorted by order of increasing values of "Bad."
  - `'Totals'` — The categories are sorted by order of increasing values of total number of observations ("Good" plus "Bad").
  - `'None'` — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm. (The existing order of the categories can be seen in the category grouping optional output from `bininfo`.)

  For more information, see Sort Categories on page 18-2179

Example: `sc = autobinning(sc,'CustAge','Algorithm','Monotone','AlgorithmOptions',{'Trend','Increasing'})`

Data Types: `cell`

`Display` — Indicator to display information on status of the binning process at command line
`'Off'` (default) | character vector with values `'On'`, `'Off'`

Indicator to display the information on status of the binning process at command line, specified using a character vector with a value of `'On'` or `'Off'`.

Data Types: `char`

Output Arguments

`sc` — Credit scorecard model
`creditscorecard` object

Credit scorecard model, returned as an updated `creditscorecard` object containing the automatically determined binning maps or rules (cut points or category groupings) for

one or more predictors. For more information on using the `creditscorecard` object, see `creditscorecard`.

Note If you have previously used the `modifybins` function to manually modify bins, these changes are lost when running `autobinning` because all the data is automatically binned based on internal autobinning rules.

Definitions

Monotone

The `'Monotone'` algorithm is an implementation of the Monotone Adjacent Pooling Algorithm (MAPA), also known as Maximum Likelihood Monotone Coarse Classifier (MLMCC); see Anderson or Thomas in the "References" on page 18-2180.

Preprocessing

During the preprocessing phase, preprocessing of numeric predictors consists in applying equal frequency binning, with the number of bins determined by the `'InitialNumBins'` parameter (the default is 10 bins). The preprocessing of categorical predictors consists in sorting the categories according to the `'SortCategories'` criterion (the default is to sort by odds in increasing order). Sorting is not applied to ordinal predictors. See the "Sort Categories" on page 18-2179 definition or the description of `AlgorithmOptions` option for `'SortCategories'` for more information.

Main Algorithm

The following example illustrates how the `'Monotone'` algorithm arrives at cut points for numeric data.

Bin	Good	Bad	Iteration1	Iteration2	Iteration3	Iteration4
`'[-Inf, 33000)'`	127	107	0.543			
`'[33000, 38000)'`	194	90	0.620	0.683		

Bin	Good	Bad	Iteration1	Iteration2	Iteration3	Iteration4
'[38000, 42000)'	135	78	0.624	0.662		
'[42000, 47000)'	164	66	0.645	0.678	0.713	
'[47000, Inf]'	183	56	0.669	0.700	0.740	0.766

Initially, the numeric data is preprocessed with an equal frequency binning. In this example, for simplicity, only the five initial bins are used. The first column indicates the equal frequency bin ranges, and the second and third columns have the “Good” and “Bad” counts per bin. (The number of observations is 1,200, so a perfect equal frequency binning would result in five bins with 240 observations each. In this case, the observations per bin do not match 240 exactly. This is a common situation when the data has repeated values.)

Monotone finds break points based on the cumulative proportion of “Good” observations. In the `'Iteration1'` column, the first value (0.543) is the number of “Good” observations in the first bin (127), divided by the total number of observations in the bin (127+107). The second value (0.620) is the number of “Good” observations in bins 1 and 2, divided by the total number of observations in bins 1 and 2. And so forth. The first cut point is set where the minimum of this cumulative ratio is found, which is in the first bin in this example. This is the end of iteration 1.

Starting from the second bin (the first bin after the location of the minimum value in the previous iteration), cumulative proportions of “Good” observations are computed again. The second cut point is set where the minimum of this cumulative ratio is found. In this case, it happens to be in bin number 3, therefore bins 2 and 3 are merged.

The algorithm proceeds the same way for two more iterations. In this particular example, in the end it only merges bins 2 and 3. The final binning has four bins with cut points at 33,000, 42,000, and 47,000.

For categorical data, the only difference is that the preprocessing step consists in reordering the categories. Consider the following categorical data:

Bin	Good	Bad	Odds
'Home Owner'	365	177	2.062

Bin	Good	Bad	Odds
'Tenant'	307	167	1.838
'Other'	131	53	2.474

The preprocessing step, by default, sorts the categories by `'Odds'`. (See the "Sort Categories" on page 18-2179 definition or the description of `AlgorithmOptions` option for `'SortCategories'` for more information.) Then, it applies the same steps described above, shown in the following table:

Bin	Good	Bad	Odds	Iteration1	Iteration2	Iteration3
'Tenant'	307	167	1.838	0.648		
'Home Owner'	365	177	2.062	0.661	0.673	
'Other'	131	53	2.472	0.669	0.683	0.712

In this case, the Monotone algorithm would not merge any categories. The only difference, compared with the data before the application of the algorithm, is that the categories are now sorted by `'Odds'`.

In both the numeric and categorical examples above, the implicit `'Trend'` choice is `'Increasing'`. (See the description of `AlgorithmOptions` option for the `'Monotone' 'Trend'` option.) If you set the trend to `'Decreasing'`, the algorithm looks for the maximum (instead of the minimum) cumulative ratios to determine the cut points. In that case, at iteration 1, the maximum would be in the last bin, which would imply that all bins should be merged into a single bin. Binning into a single bin is a total loss of information and has no practical use. Therefore, when the chosen trend leads to a single bin, the Monotone implementation rejects it, and the algorithm returns the bins found after the preprocessing step. This state is the initial equal frequency binning for numeric data and the sorted categories for categorical data. The implementation of the Monotone algorithm by default uses a heuristic to identify the trend (`'Auto'` option for `'Trend'`).

Equal Frequency

Unsupervised algorithm that divides the data into a predetermined number of bins that contain approximately the same number of observations.

`EqualFrequency` is defined as:

Let v[1], v[2],..., v[N] be the sorted list of different values or categories observed in the data. Let f[*i*] be the frequency of v[*i*]. Let F[*k*] = f[1]+...+f[*k*] be the cumulative sum of frequencies up to the *k*th sorted value. Then F[*N*] is the same as the total number of observations.

Define AvgFreq = F[*N*] / *NumBins*, which is the ideal average frequency per bin after binning. The *n*th cut point index is the index *k* such that the distance abs(F[*k*] - *n**AvgFreq) is minimized.

This rule attempts to match the cumulative frequency up to the *n*th bin. If a single value contains too many observations, equal frequency bins are not possible, and the above rule yields less than *NumBins* total bins. In that case, the algorithm determines *NumBins* bins by breaking up bins, in the order in which the bins were constructed.

The preprocessing of categorical predictors consists in sorting the categories according to the `'SortCategories'` criterion (the default is to sort by odds in increasing order). Sorting is not applied to ordinal predictors. See the “Sort Categories” on page 18-2179 definition or the description of `AlgorithmOptions` option for `'SortCategories'` for more information.

Equal Width

Unsupervised algorithm that divides the range of values in the domain of the predictor variable into a predetermined number of bins of “equal width.” For numeric data, the width is measured as the distance between bin edges. For categorical data, width is measured as the number of categories within a bin.

The `EqualWidth` option is defined as:

For numeric data, if `MinValue` and `MaxValue` are the minimum and maximum data values, then

```
Width = (MaxValue - MinValue)/NumBins
```

and the `CutPoints` are set to `MinValue` + Width, `MinValue` + 2*Width, ... `MaxValue` – Width. If a `MinValue` or `MaxValue` have not been specified using the `modifybins` function, the `EqualWidth` option sets `MinValue` and `MaxValue` to the minimum and maximum values observed in the data.

For categorical data, if there are *NumCats* numbers of original categories, then

```
Width = NumCats / NumBins,
```

and set cut point indices to the rounded values of Width, 2*Width, ..., *NumCats* – Width, plus 1.

The preprocessing of categorical predictors consists in sorting the categories according to the `'SortCategories'` criterion (the default is to sort by odds in increasing order). Sorting is not applied to ordinal predictors. See the “Sort Categories” on page 18-2179 definition or the description of `AlgorithmOptions` option for `'SortCategories'` for more information.

Sort Categories

As a preprocessing step for categorical data, `'Monotone'`, `'EqualFrequency'`, and `'EqualWidth'` support the `'SortCategories'` input. This serves the purpose of reordering the categories before applying the main algorithm. The default sorting criterion is to sort by `'Odds'`. For example, suppose that the data originally looks like this:

Bin	Good	Bad	Odds
`'Home Owner'`	365	177	2.062
`'Tenant'`	307	167	1.838
`'Other'`	131	53	2.472

After the preprocessing step, the rows would be sorted by `'Odds'` and the table looks like this:

Bin	Good	Bad	Odds
`'Tenant'`	307	167	1.838
`'Home Owner'`	365	177	2.062
`'Other'`	131	53	2.472

The three algorithms only merge adjacent bins, so the initial order of the categories makes a difference for the final binning. The `'None'` option for `'SortCategories'` would leave the original table unchanged. For a description of the sorting criteria supported, see the description of the `AlgorithmOptions` option for `'SortCategories'`.

Upon the construction of a scorecard, the initial order of the categories, before any algorithm or any binning modifications are applied, is the order shown in the first output

of `bininfo`. If the bins have been modified (either manually with `modifybins` or automatically with `autobinning`), use the optional output (`cg,'category grouping'`) from `bininfo` to get the current order of the categories.

The `'SortCategories'` option has no effect on categorical predictors for which the `'Ordinal'` parameter is set to true (see the `'Ordinal'` input parameter in MATLAB categorical arrays for `categorical`. Ordinal data has a natural order, which is honored in the preprocessing step of the algorithms by leaving the order of the categories unchanged. Only categorical predictors whose `'Ordinal'` parameter is false (default option) are subject to reordering of categories according to the `'SortCategories'` criterion.

Using `autobinning` with Weights

When observation weights are defined using the optional `WeightsVar` argument when creating a `creditscorecard` object, instead of counting the rows that are good or bad in each bin, the `autobinning` function accumulates the weight of the rows that are good or bad in each bin.

The "frequencies" reported are no longer the basic "count" of rows, but the "cumulative weight" of the rows that are good or bad and fall in a particular bin. Once these "weighted frequencies" are known, all other relevant statistics (`Good`, `Bad`, `Odds`, `WOE`, and `InfoValue`) are computed with the usual formulas. For more information, see "Credit Scorecard Modeling Using Observation Weights" on page 8-65.

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Data Preparation for Data Mining Using SAS*. Morgan Kaufmann, 2006.

[3] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

[4] Thomas, L., et al. *Credit Scoring and Its Applications*. Society for Industrial and Applied Mathematics, 2002.

See Also

`bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `validatemodel`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57
"Credit Scorecard Modeling Using Observation Weights" on page 8-65

Introduced in R2014b

probdefault

Likelihood of default for given data set

Syntax

```
pd = probdefault(sc)
pd = probdefault(sc,data)
```

Description

`pd = probdefault(sc)` computes the probability of default for `sc`, the data used to build the `creditscorecard` object.

`pd = probdefault(sc,data)` computes the probability of default for a given data set specified using the optional argument `data`.

By default, the data used to build the `creditscorecard` object are used. You can also supply input data, to which the same computation of probability of default is applied.

Examples

Compute Probability for Default Using Credit ScoreCard Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID')

sc =
  creditscorecard with properties:

               GoodLabel: 0
            ResponseVar: 'status'
```

```
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                 IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

Perform automatic binning using the default options. By default, `autobinning` uses the `Monotone` algorithm.

```
sc = autobinning(sc);
```

Fit the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    _______    __________

    (Intercept)    0.70239     0.064001    10.975     5.0538e-28
    CustAge        0.60833      0.24932      2.44       0.014687
    ResStatus        1.377      0.65272    2.1097       0.034888
    EmpStatus      0.88565        0.293    3.0227      0.0025055
    CustIncome     0.70164      0.21844    3.2121      0.0013179
    TmWBank         1.1074      0.23271    4.7589     1.9464e-06
    OtherCC         1.0883      0.52912    2.0569       0.039696
    AMBalance         1.045      0.32214    3.2439      0.0011792


1200 observations, 1192 error degrees of freedom
```

```
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Compute the probability of default.

```
pd = probdefault(sc);
disp(pd(1:15,:))
```

```
    0.2503
    0.1878
    0.3173
    0.1711
    0.1895
    0.1307
    0.5218
    0.2848
    0.2612
    0.3047
    0.3418
    0.2237
    0.2793
    0.3615
    0.1653
```

- “Case Study for a Credit Scorecard Analysis” on page 8-78
- “Troubleshooting Credit Scorecard Results” on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. To create this object, use `creditscorecard`.

`data` — Dataset to apply probability of default rules

table

(Optional) Dataset to apply probability of default rules, specified as a MATLAB table, where each row corresponds to individual observations. The data must contain columns for each of the predictors in the `creditscorecard` object.

Data Types: `table`

Output Arguments

`pd` — Probability of default

array

Probability of default, returned as a `NumObs`-by-`1` numerical array of default probabilities.

Definitions

Default Probability

After the unscaled scores are computed (see "Algorithms for Computing and Scaling Scores" on page 18-2032), the probability of the points being "Good" is represented by the following formula:

```
ProbGood = 1./(1 + exp(-UnscaledScores))
```

Thus, the probability of default is

```
pd = 1 - ProbGood
```

References

[1] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `score` | `setmodel` | `table` | `validatemodel`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78

"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57

Introduced in R2015a

validatemodel

Validate quality of credit scorecard model

Syntax

```
Stats = validatemodel(sc)
Stats = validatemodel(sc,data)
[Stats,T] = validatemodel(sc,Name,Value)
[Stats,T,hf] = validatemodel(sc,Name,Value)
```

Description

`Stats = validatemodel(sc)` validates the quality of the `creditscorecard` model.

By default, the data used to build the `creditscorecard` object is used. You can also supply input data to which the validation is applied.

`Stats = validatemodel(sc,data)` validates the quality of the `creditscorecard` model for a given data set specified using the optional argument `data`.

`[Stats,T] = validatemodel(sc,Name,Value)` validates the quality of the `creditscorecard` model using the optional name-value pair arguments, and returns `Stats` and `T` outputs.

`[Stats,T,hf] = validatemodel(sc,Name,Value)` validates the quality of the `creditscorecard` model using the optional name-value pair arguments, and returns the figure handle `hf` to the CAP, ROC, and KS plots.

Examples

Validate a Credit Scorecard Model

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data, 'IDVar','CustID')
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                     Data: [1200x11 table]
```

Perform automatic binning using the default options. By default, `autobinning` uses the `Monotone` algorithm.

```
sc = autobinning(sc);
```

Fit the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-0
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    status ~ [Linear formula with 8 terms in 7 predictors]
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
```

```
                ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance         1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Format the unscaled points.

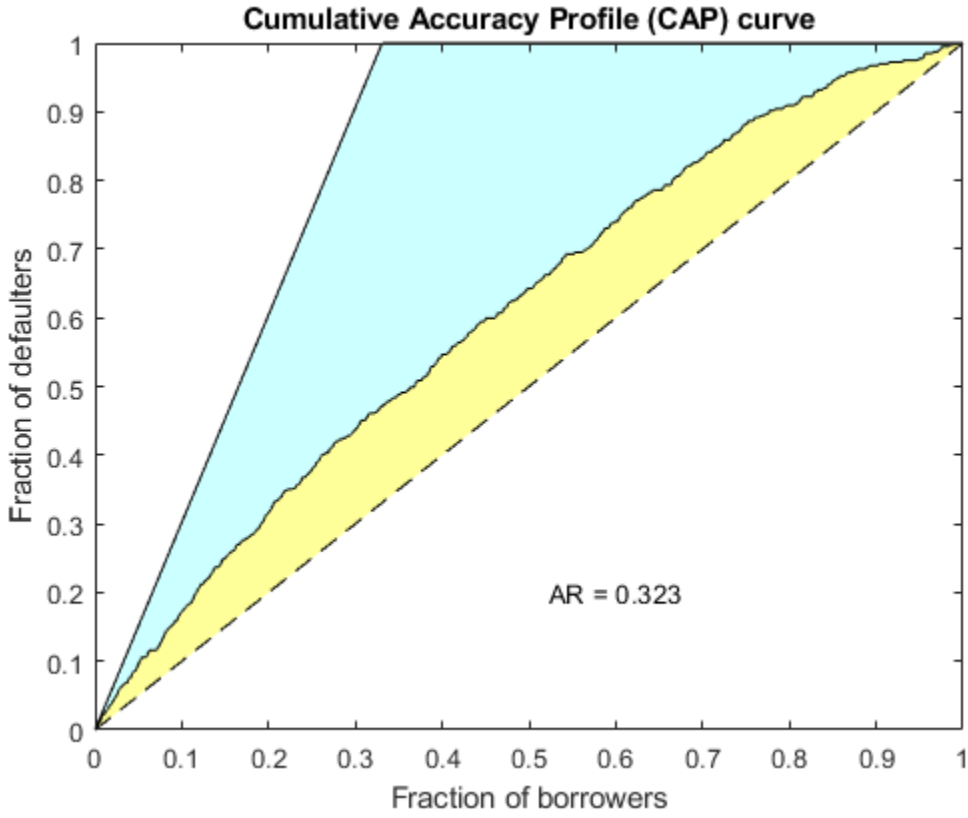
```
sc = formatpoints(sc, 'PointsOddsAndPDO',[500,2,50]);
```

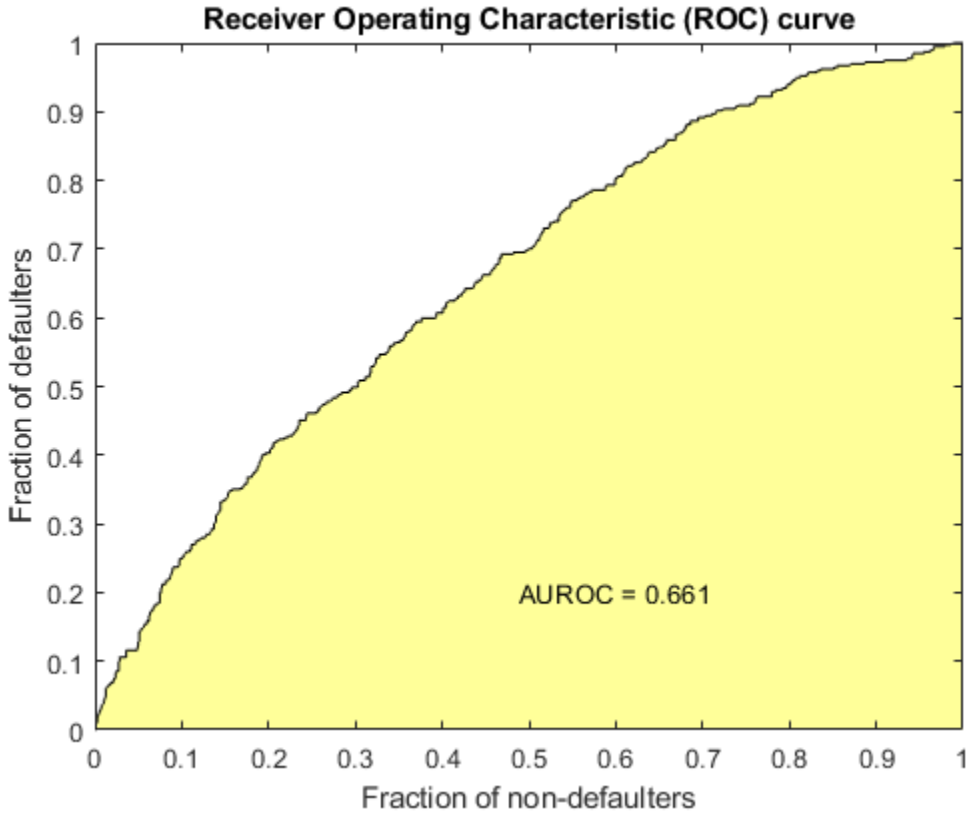
Score the data.

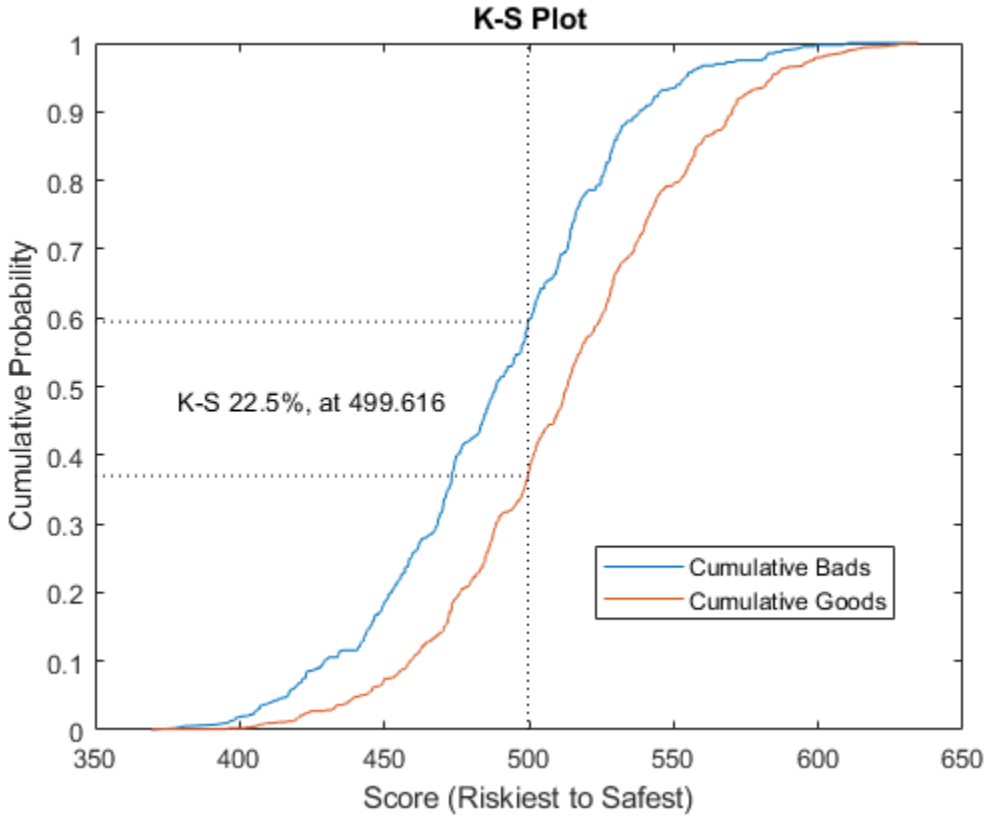
```
scores = score(sc);
```

Validate the credit scorecard model by generating the CAP, ROC, and KS plots.

```
[Stats,T] = validatemodel(sc,'Plot',{'CAP','ROC','KS'});
```


Cumulative Accuracy Profile (CAP) curve
Fraction of defaulters
Fraction of borrowers
AR = 0.323
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Receiver Operating Characteristic (ROC) curve
Fraction of defaulters
Fraction of non-defaulters
AUROC = 0.661
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1




```
disp(Stats)
```

Measure	Value
'Accuracy Ratio'	0.32258
'Area under ROC curve'	0.66129
'KS statistic'	0.2246
'KS score'	499.62

```
disp(T(1:15,:))
```

Scores	ProbDefault	TrueBads	FalseBads	TrueGoods	FalseGoods	Sensit

```
    369.54    0.75313         0          1          802          397
    378.19    0.73016         1          1          802          396          0.0025
    380.28    0.72444         2          1          802          395          0.0050
    391.49    0.69234         3          1          802          394          0.0075
    395.57    0.68017         4          1          802          393           0.010
    396.14    0.67846         4          2          801          393           0.010
    396.45    0.67752         5          2          801          392           0.012
    398.61    0.67094         6          2          801          391           0.015
    398.68    0.67072         7          2          801          390           0.017
    401.33    0.66255         8          2          801          389           0.020
    402.66    0.65842         8          3          800          389           0.020
    404.25    0.65346         9          3          800          388            0.02
    404.73    0.65193         9          4          799          388            0.02
    405.53    0.64941        11          4          799          386           0.027
     405.7    0.64887        11          5          798          386           0.027
```

Validate a Credit Score Card Model With Weights

Use the `CreditCardData.mat` file to load the data (`dataWeights`) that contains a column (`RowWeights`) for the weights (using a dataset from Refaat 2011).

```
load CreditCardData
```

Create a `creditscorecard` object using the optional name-value pair argument for `'WeightsVar'`.

```
sc = creditscorecard(dataWeights,'IDVar','CustID','WeightsVar','RowWeights')
```

```
sc =

  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: 'RowWeights'
               VarNames: {1x12 cell}
      NumericPredictors: {1x6 cell}
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                  IDVar: 'CustID'
         PredictorVars: {1x9 cell}
```

```
                    Data: [1200x12 table]
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc =

  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: 'RowWeights'
                VarNames: {1x12 cell}
       NumericPredictors: {1x6 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                 IDVar: 'CustID'
          PredictorVars: {1x9 cell}
                    Data: [1200x12 table]
```

Fit the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 764.3187, Chi2Stat = 15.81927, PValue = 6.968927e-05
2. Adding TmWBank, Deviance = 751.0215, Chi2Stat = 13.29726, PValue = 0.0002657942
3. Adding AMBalance, Deviance = 743.7581, Chi2Stat = 7.263384, PValue = 0.007037455

Generalized linear regression model:
    logit(status) ~ 1 + CustIncome + TmWBank + AMBalance
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70642     0.088702     7.964    1.6653e-15
    CustIncome      1.0268      0.25758    3.9862    6.7132e-05
    TmWBank         1.0973      0.31294    3.5063     0.0004543
    AMBalance       1.0039      0.37576    2.6717     0.0075464
```

```
1200 observations, 1196 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 36.4, p-value = 6.22e-08
```

Format the unscaled points.

```
sc = formatpoints(sc, 'PointsOddsAndPDO',[500,2,50]);
```

Score the data.

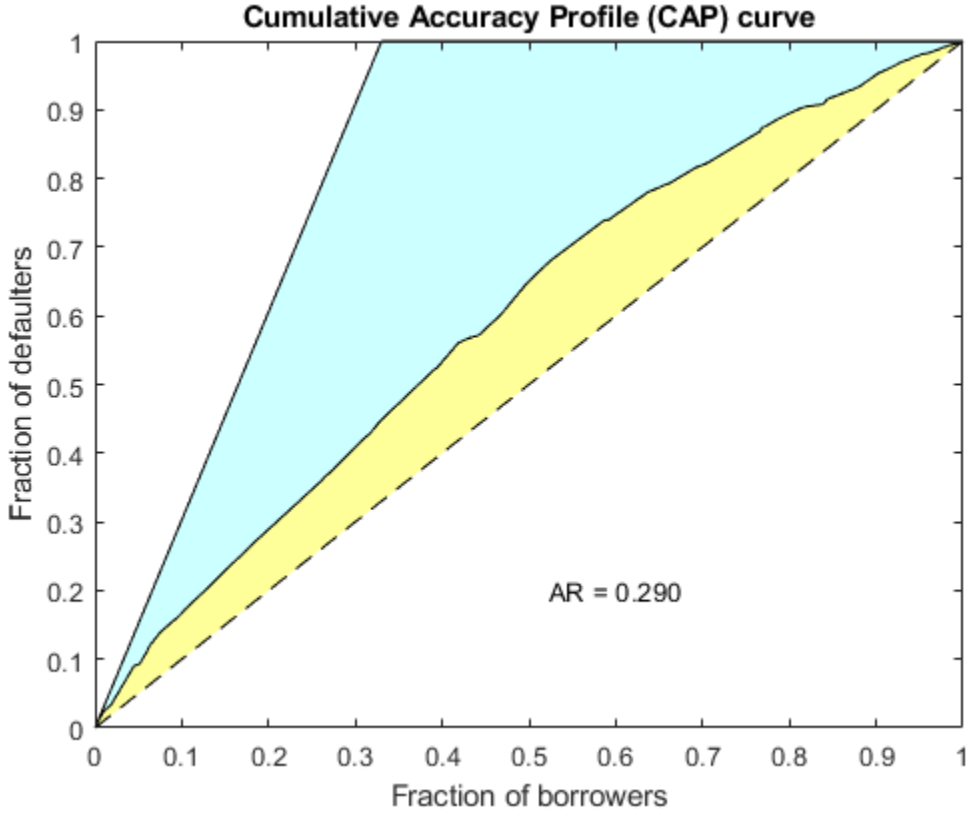
```
scores = score(sc);
```

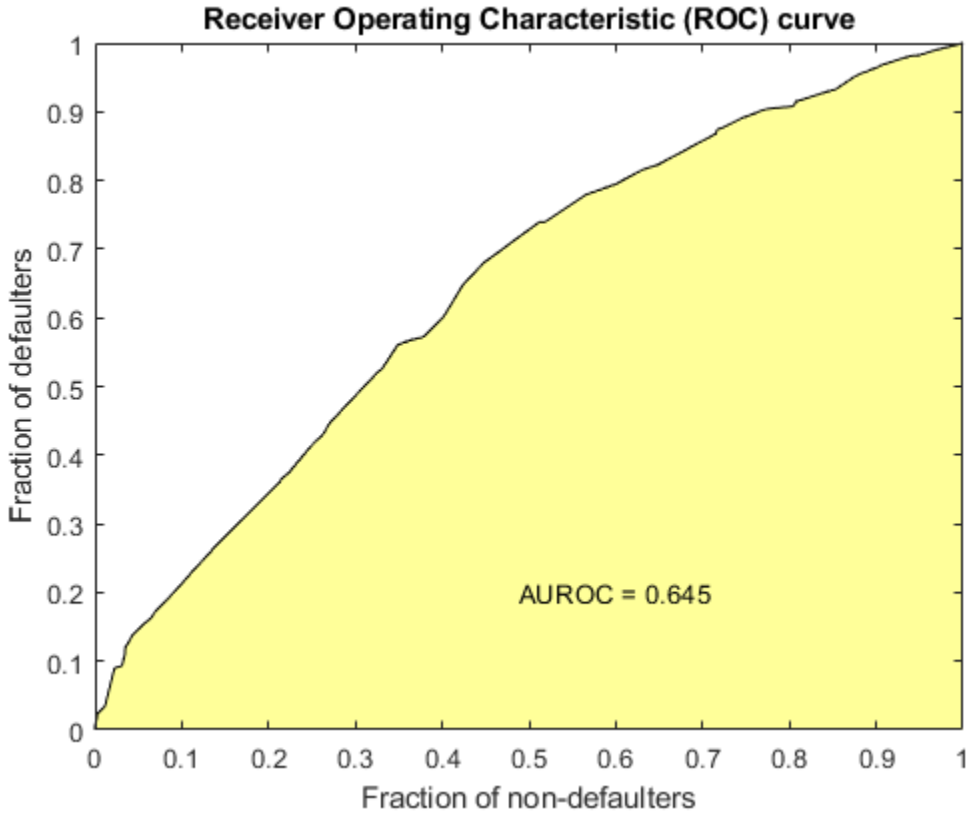
Validate the credit scorecard model by generating the CAP, ROC, and KS plots. When the optional name-value pair argument `'WeightsVar'` is used to specify observation (sample) weights, the `T` table uses statistics, sums, and cumulative sums that are weighted counts.

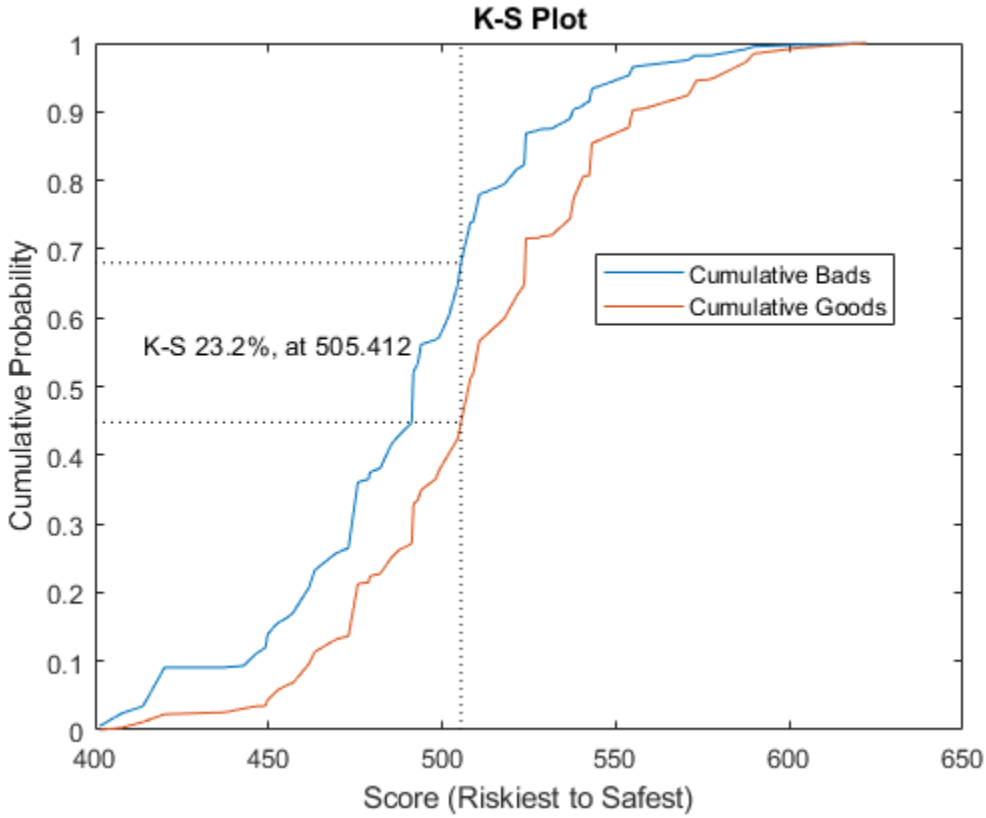
```
[Stats,T] = validatemodel(sc,'Plot',{'CAP','ROC','KS'});
Stats
T(1:10,:)
```

```
Stats =

  4x2 table

           Measure              Value
    ________________________    _______

    'Accuracy Ratio'           0.28972
    'Area under ROC curve'     0.64486
    'KS statistic'             0.23215
    'KS score'                  505.41


ans =

  10x9 table

    Scores    ProbDefault    TrueBads    FalseBads    TrueGoods    FalseGoods    Sensit
    ______    ___________    ________    _________    _________    __________    ______

    401.34    0.66253        1.0788           0       411.95       201.95        0.0053
    407.59    0.64289        4.8363      1.2768       410.67       198.19         0.023
    413.79    0.62292        6.9469      4.6942       407.25       196.08         0.034
```

```
    420.04    0.60236     18.459     9.3899     402.56     184.57     0.090
    437.27      0.544     18.459     10.514     401.43     184.57     0.090
    442.83    0.52481     18.973     12.794     399.15     184.06     0.093
    446.19    0.51319     22.396      14.15      397.8     180.64      0.11
    449.08    0.50317     24.325     14.405     397.54     178.71      0.11
    449.73    0.50095     28.246     18.049      393.9     174.78      0.13
    452.44    0.49153     31.511     23.565     388.38     171.52       0.1
```




Receiver Operating Characteristic (ROC) curve
Fraction of defaulters
Fraction of non-defaulters
AUROC = 0.645
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



- "Case Study for a Credit Scorecard Analysis" on page 8-78
- "Troubleshooting Credit Scorecard Results" on page 8-68

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object. To create this object, use `creditscorecard`.

`data` — Validation data

table

(Optional) Validation data, specified as a MATLAB table, where each table row corresponds to individual observations. The `data` must contain columns for each of the predictors in the credit scorecard model. The columns of data can be any one of the following data types:

- Numeric
- Logical
- Cell array of character vectors
- Character array
- Categorical
- String
- String array

In addition, the table must contain a binary response variable.

Note When observation weights are defined using the optional `WeightsVar` name-value pair argument when creating a `creditscorecard` object, the weights stored in the `WeightsVar` column are used when validating the model on the training data. If a different validation data set is provided using the optional `data` input, observation weights for the validation data must be included in a column whose name matches `WeightsVar`, otherwise unit weights are used for the validation data. For more information, see “Using validatemodel with Weights” on page 18-2203.

Data Types: `table`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sc = validatemodel(sc,data,'AnalysisLevel','Deciles','Plot','CAP')`

`AnalysisLevel` — Type of analysis level

`'Scores'` (default) | character vector with values `'Deciles'`, `'Scores'`

Type of analysis level, specified as character vector with one of the following values:

- `'Scores'` — Returns the statistics (`Stats`) at the observation level. Scores are sorted from riskiest to safest, and duplicates are removed.
- `'Deciles'` — Returns the statistics (`Stats`) at decile level. Scores are sorted from riskiest to safest and binned with their corresponding statistics into 10 deciles (10%, 20%, ..., 100%).

Data Types: `char`

`Plot` — Type of plot

`'None'` (default) | character vector with values `'None'`, `'CAP'`, `'ROC'`,`'KS'` | cell array of character vectors with values `'None'`, `'CAP'`, `'ROC'`,`'KS'`

Type of plot, specified as character vector with one of the following values:

- `'None'` — No plot is displayed.
- `'CAP'` — Cumulative Accuracy Profile. Plots the fraction of borrowers up to score "s" versus the fraction of defaulters up to score "s" (`'PctObs'` versus `'Sensitivity'` columns of `T` optional output argument). For more details, see "Cumulative Accuracy Profile (CAP)" on page 18-2202.
- `'ROC'` — Receiver Operating Characteristic. Plots the fraction of non-defaulters up to score "s" versus the fraction of defaulters up to score "s" (`'FalseAlarm'` versus `'Sensitivity'` columns of `T` optional output argument). For more details, see "Receiver Operating Characteristic (ROC)" on page 18-2203.
- `'KS'` — Kolmogorov-Smirnov. Plots each score "s" versus the fraction of defaulters up to score "s," and also versus the fraction of non-defaulters up to score "s" (`'Scores'` versus both `'Sensitivity'` and `'FalseAlarm'` columns of the optional output argument `T`). For more details, see "Kolmogorov-Smirnov statistic (KS)" on page 18-2203.

Tip For the Kolmogorov-Smirnov statistic option, you can enter `'KS'` or `'K-S'`.

Data Types: `char` | `cell`

Output Arguments

`Stats` — Validation measures

table

Validation measures, returned as a `4`-by-`2` table. The first column, `'Measure'`, contains the names of the following measures:

- Accuracy ratio (AR)
- Area under the ROC curve (AUROC)
- The KS statistic
- KS score

The second column, `'Value'`, contains the values corresponding to these measures.

`T` — Validation statistics data

array

Validation statistics data, returned as an `N`-by-`9` table of validation statistics data, sorted, by score, from riskiest to safest. When `AnalysisLevel` is set to `'Deciles'`, `N` is equal to `10`. Otherwise, `N` is equal to the total number of unique scores, that is, scores without duplicates.

The table `T` contains the following nine columns, in this order:

- `'Scores'` — Scores sorted from riskiest to safest. The data in this row corresponds to all observations up to, and including the score in this row.
- `'ProbDefault'` — Probability of default for observations in this row. For deciles, the average probability of default for all observations in the given decile is reported.
- `'TrueBads'` — Cumulative number of “bads” up to, and including, the corresponding score.
- `'FalseBads'` — Cumulative number of “goods” up to, and including, the corresponding score.
- `'TrueGoods'` — Cumulative number of “goods” above the corresponding score.
- `'FalseGoods'` — Cumulative number of “bads” above the corresponding score.
- `'Sensitivity'` — Fraction of defaulters (or the cumulative number of “bads” divided by total number of “bads”). This is the distribution of “bads” up to and including the corresponding score.

- `'FalseAlarm'` — Fraction of non-defaulters (or the cumulative number of “goods” divided by total number of “goods”). This is the distribution of “goods” up to and including the corresponding score.
- `'PctObs'` — Fraction of borrowers, or the cumulative number of observations, divided by total number of observations up to and including the corresponding score.

Note When creating the `creditscorecard` object with `creditscorecard`, if the optional name-value pair argument `WeightsVar` was used to specify observation (sample) weights, then the `T` table uses statistics, sums, and cumulative sums that are weighted counts.

`hf` — Handle to the plotted measures
figure handle

Figure handle to plotted measures, returned as a figure handle or array of handles. When `Plot` is set to `'None'`, `hf` is an empty array.

Definitions

Cumulative Accuracy Profile (CAP)

CAP is generally a concave curve and is also known as the Gini curve, Power curve, or Lorenz curve.

The scores of given observations are sorted from riskiest to safest. For a given fraction `M` (0% to 100%) of the total borrowers, the height of the CAP curve is the fraction of defaulters whose scores are less than or equal to the maximum score of the fraction `M`, also known as “Sensitivity.”

The area under the CAP curve, known as the AUCAP, is then compared to that of the perfect or “ideal” model, leading to the definition of a summary index known as the accuracy ratio (*AR*) or the Gini coefficient:

$$AR = \frac{A_R}{A_P}$$

where A_R is the area between the CAP curve and the diagonal, and A_P is the area between the perfect model and the diagonal. This represents a “random” model, where

scores are assigned randomly and therefore the proportion of defaulters and non-defaulters is independent of the score. The perfect model is the model for which all defaulters are assigned the lowest scores, and therefore, perfectly discriminates between defaulters and nondefaulters. Thus, the closer to unity *AR* is, the better the scoring model.

Receiver Operating Characteristic (ROC)

To find the receiver operating characteristic (ROC) curve, the proportion of defaulters up to a given score “s,” or “Sensitivity,” is computed.

This proportion is known as the true positive rate (TPR). Additionally, the proportion of nondefaulters up to score “s,“ or “False Alarm Rate,” is also computed. This proportion is also known as the false positive rate (FPR). The ROC curve is the plot of the “Sensitivity” vs. the “False Alarm Rate.” Computing the ROC curve is similar to computing the equivalent of a confusion matrix at each score level.

Similar to the CAP, the ROC has a summary statistic known as the area under the ROC curve (AUROC). The closer to unity, the better the scoring model. The accuracy ratio (*AR*) is related to the area under the curve by the following formula:

$$AR = 2(AUROC) - 1$$

Kolmogorov-Smirnov statistic (KS)

The Kolmogorov-Smirnov (KS) plot, also known as the fish-eye graph, is a common statistic used to measure the predictive power of scorecards.

The KS plot shows the distribution of defaulters and the distribution of non-defaulters on the same plot. For the distribution of defaulters, each score “s” is plotted versus the proportion of defaulters up to “s," or “Sensitivity." For the distribution of non-defaulters, each score “s” is plotted versus the proportion of non-defaulters up to “s," or “False Alarm." The statistic of interest is called the KS statistic and is the maximum difference between these two distributions (“Sensitivity” minus “False Alarm”). The score at which this maximum is attained is also of interest.

Using `validatemodel` with Weights

Model validation statistics incorporate observation weights when these are provided by the user.

Without weights, the validation statistics are based on how many good and bad observations fall below a particular score. On the other hand, when observation weights are provided, the weight (not the count) is accumulated for the good and the bad observations that fall below a particular score.

When observation weights are defined using the optional `WeightsVar` name-value pair argument when creating a `creditscorecard` object, the weights stored in the `WeightsVar` column are used when validating the model on the training data. When a different validation data set is provided using the optional `data` input, observation weights for the validation data must be included in a column whose name matches `WeightsVar`, otherwise unit weights are used for the validation data set.

Not only the validation statistics, but the credit scorecard scores themselves depend on the observation weights of the training data. For more information, see "Using fitmodel with Weights" on page 18-2084 and "Credit Scorecard Modeling Using Observation Weights" on page 8-65.

References

[1] *"Basel Committee on Banking Supervision: Studies on the Validation of Internal Rating Systems."* Working Paper No. 14, February 2005.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

[3] Loeffler, G. and Posch, P. N. *Credit Risk Modeling Using Excel and VBA*. Wiley Finance, 2007.

See Also

`bindata` | `bininfo` | `creditscorecard` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `table`

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Credit Scorecard Modeling Workflow" on page 8-62
"About Credit Scorecards" on page 8-57

"Credit Scorecard Modeling Using Observation Weights" on page 8-65

Introduced in R2015a

creditscorecard

Create `creditscorecard` object to build credit scorecard model

Description

Build a credit scorecard model by creating a `creditscorecard` object and specify input data in a table format.

After creating a `creditscorecard` object, you can use the associated object functions to bin the data and perform logistic regression analysis to develop a credit scorecard model to guide credit decisions. This workflow shows how to develop a credit scorecard model.

1. Create a `creditscorecard` object (see "Create creditscorecard" on page 18-2206 and "Properties" on page 18-2210).
2. Bin the data.
3. Fit a logistic regression model.
4. Review and format the credit scorecard points.
5. Score the data.
6. Calculate the probabilities of default for the data.
7. Validate the quality of the credit scorecard model.

For more detailed information on this workflow, see "Credit Scorecard Modeling Workflow" on page 8-62.

Creation

Syntax

```
sc = creditscorecard(data)
sc = creditscorecard(___,Name,Value)
```

Description

`sc = creditscorecard(data)` creates a `creditscorecard` object by specifying `data`. The credit scorecard model, returned as a `creditscorecard` object, contains the binning maps or rules (cut points or category groupings) for one or more predictors.

`sc = creditscorecard(___,Name,Value)` sets Properties on page 18-2210 using name-value pairs and any of the arguments in the previous syntax. For example, `sc = creditscorecard(data,'GoodLabel', 0,'IDVar','CustID','ResponseVar','status','PredictorVars', {'CustID','CustIncome'})`. You can specify multiple name-value pairs.

Note To use observation (sample) weights in the credit scorecard workflow, when creating a `creditscorecard` object, you must use the optional name-value pair `WeightsVar` to define which column in the `data` contains the weights.

Input Arguments

`data` — Data for `creditscorecard` object

table

Data for the `creditscorecard` object, specified as a MATLAB table, where each column of data can be any one of the following data types:

- Numeric
- Logical
- Cell array of character vectors
- Character array
- Categorical
- String

In addition, the table must contain a binary response variable. Before creating a `creditscorecard` object, perform a data preparation task to have appropriately structured `data` as input to a `creditscorecard` object. The `data` input sets the Data on page 18-0 property.

Data Types: `table`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sc = creditscorecard(data,'GoodLabel', 0,'IDVar','CustID','ResponseVar','status','PredictorVars', {'CustID','CustIncome'})`

`GoodLabel` — Indicator for which of two possible values in response variable correspond to "Good" observations

set to the response value with the highest count (default) | character vector | numeric scalar | logical

Indicator for which of the two possible values in the response variable correspond to "Good" observations, specified as a numeric scalar, logical, or character vector. The `GoodLabel` name-value pair argument sets the GoodLabel on page 18-0 property.

When specifying `GoodLabel`, follow these guidelines.

If Response Is...	`GoodLabel` Must be...
numeric	numeric
logical	logical or numeric
cell array of character vectors	character vector
character array	character vector
categorical	character vector

If not specified, `GoodLabel` is set to the response value with the highest count. However, if the optional `WeightsVar` argument is provided when creating the `creditscorecard` object, then counts are replaced with weighted frequencies. For more information, see "Credit Scorecard Modeling Using Observation Weights" on page 8-65.

`GoodLabel` can only be set when creating the `creditscorecard` object. This parameter cannot be set using dot notation.

Data Types: `char` | `double`

`IDVar` — Variable name used as ID or tag for observations

empty character vector `''` (default) | character vector

Variable name used as ID or tag for the observations, specified as a character vector. The `IDVar` data could be an ordinal number (for example, 1,2,3...), a Social Security number. This is provided as a convenience to remove this column from the predictor variables. `IDVar` is case-sensitive. The `IDVar` name-value pair argument sets the IDVar on page 18-0 property.

You can set this optional parameter using the `creditscorecard` function or by using dot notation at the command line, as follows.

Example: `sc.IDVar = 'CustID'`

Data Types: `char`

`ResponseVar` — Response variable name for "Good" or "Bad" indicator
last column of the data input (default) | character vector

Response variable name for the "Good" or "Bad" indicator, specified as a character vector. The response variable data must be binary. The `ResponseVar` name-value pair argument sets the ResponseVar on page 18-0 property.

If not specified, `ResponseVar` is set to the last column of the data input. `ResponseVar` can only be set when creating the `creditscorecard` object using the `creditscorecard` function. `ResponseVar` is case-sensitive.

Data Types: `char`

`WeightsVar` — Weights variable name
empty character vector `''` (default) | character vector

Weights variable name, specified as a character vector to indicate which column name in the `data` table contains the row weights. `WeightsVar` is case-sensitive. The `WeightsVar` name-value pair argument sets the WeightsVar on page 18-0 property, and this property can only be set at the creation of a `creditscorecard` object. If the name-value pair argument `WeightsVar` is not specified when creating a `creditscorecard` object, then observation weights are set to unit weights by default.

The `WeightsVar` values are used in the credit scorecard workflow by `autobinning`, `bininfo`, `fitmodel`, and `validatemodel`. For more information, see "Credit Scorecard Modeling Using Observation Weights" on page 8-65.

Data Types: `char`

`PredictorVars` — Predictor variable names
set difference between `VarNames` and `{IDVar, ResponseVar}` (default) | cell array of character vectors containing names

Predictor variable names, specified using a cell array of character vectors containing names. By default, when you create a credit scorecard `creditscorecard`, all variables are predictors except for `IDVar` and `ResponseVar`.

The `PredictorVars` name-value pair argument sets the PredictorVars on page 18-0 property.This property can be modified by using dot notation or by using a name-value pair argument for the `fitmodel` function. `PredictorVars` is case-sensitive and the predictor variable name cannot be the same as the `IDVar` or `ResponseVar`.

Data Types: `cell`

Properties

`Data` — Data used to create the `creditscorecard` object
table

Data used to create the `creditscorecard` object, specified as a table when creating a `creditscorecard` object. In the `Data` property, categorical predictors are stored as categorical arrays.

Example: `sc.Data(1:10,:)`

Data Types: `table`

`IDVar` — Name of the variable used as ID or tag for the observations
empty character vector `''` (default) | character vector

Name of the variable used as ID or tag for the observations, specified as a character vector. This property can be set as an optional parameter when creating a `creditscorecard` object or by using dot notation at the command line. `IDVar` is case-sensitive.

Example: `sc.IDVar = 'CustID'`

Data Types: `char`

`VarNames` — All variable names from the data input
`VarNames` come directly from data input to `creditscorecard` object (default)

This property is read-only.

`VarNames` is a cell array of character vectors containing the names of all variables in the data. The `VarNames` come directly from the data input to the `creditscorecard` object. `VarNames` is case-sensitive.

Data Types: `cell`

`ResponseVar` — Name of the response variable, "Good" or "Bad" indicator
last column of the data input (default) | character vector

Name of the response variable, "Good" or "Bad" indicator, specified as a character vector. The response variable data must be binary. If not specified, `ResponseVar` is set to the last column of the data input. This property can only be set with an optional parameter when creating a `creditscorecard` object. `ResponseVar` is case-sensitive.

Data Types: `char`

`WeightsVar` — Name of the variable used as ID or tag for weights
empty character vector `''` (default) | character vector

Name of the variable used as ID or tag to indicate which column name in the `data` table contains the row weights, specified as a character vector. This property can be set as an optional parameter (`WeightsVar`) when creating a `creditscorecard` object. `WeightsVar` is case-sensitive.

Data Types: `char`

`GoodLabel` — Indicator for which of the two possible values in the response variable correspond to "Good" observations
set to the response value with the highest count (default) | character vector | numeric scalar | logical

Indicator for which of the two possible values in the response variable correspond to "Good" observations. When specifying `GoodLabel`, follow these guidelines:

If Response is...	`GoodLabel` must be:
numeric	numeric
logical	logical or numeric
cell array of character vectors	character vector
character array	character vector

If Response is...	`GoodLabel` must be:
categorical	character vector

If not specified, `GoodLabel` is set to the response value with the highest count. This property can only be set with an optional parameter when creating a `creditscorecard` object. This property cannot be set using dot notation.

Data Types: `char` | `double`

`PredictorVars` — Predictor variable names

set difference between `VarNames` and `{IDVar,ResponseVar}` (default) | cell array of character vectors containing names

Predictor variable names, specified using a cell array of character vectors containing names. By default, when you create a `creditscorecard` object, all variables are predictors except for `IDVar` and `ResponseVar`. This property can be modified using a name-value pair argument for the `fitmodel` function or by using dot notation. `PredictorVars` is case-sensitive and the predictor variable name cannot be the same as the `IDVar` or `ResponseVar`.

Example: `sc.PredictorVars = {'CustID','CustIncome'}`

Data Types: `cell`

`NumericPredictors` — Name of numeric predictors

empty character vector `''` (default) | character vector

Name of numeric predictors, specified as a character vector. This property cannot be set by using dot notation at the command line. It can only be modified using the `modifypredictor` function.

Data Types: `char`

`CategoricalPredictors` — Name of categorical predictors

empty character vector `''` (default) | character vector

Name of categorical predictors, specified as a character vector. This property cannot be set by using dot notation at the command line. It can only be modified using the `modifypredictor` function.

Data Types: `char`

`creditscorecard` Property	Set/Modify Property from Command Line Using `creditscorecard` Function	Modify Property Using Dot Notation	Property Not User-Defined and Value Is Defined Internally
`Data`	No	No	Yes, copy of `data` input
`IDVar`	Yes	Yes	No, but the user specifies this
`VarNames`	No	No	Yes
`ResponseVar`	Yes	No	If not specified, set to last column of `data` input
`WeightsVar`	No	No	Yes
`GoodLabel`	Yes	No	If not specified, set to response value with highest count
`PredictorVars`	Yes (also modifiable using `fitmodel` function)	Yes	Yes, but the user can modify this
`NumericPredictors`	No (can only be modified using `modifypredictor` function)	No	Yes, but the user can modify this
`CategoricalPredictors`	No (can only be modified using `modifypredictor` function)	No	Yes, but the user can modify this

Object Functions

autobinning	Perform automatic binning of given predictors
bininfo	Return predictor's bin information
predictorinfo	Summary of credit scorecard predictor properties
modifypredictor	Set properties of credit scorecard predictors

modifybins	Modify predictor's bins
bindata	Binned predictor variables
plotbins	Plot histogram counts for predictor variables
fitmodel	Fit logistic regression model to Weight of Evidence (WOE) data
setmodel	Set model predictors and coefficients
displaypoints	Return points per predictor per bin
formatpoints	Format scorecard points and scaling
score	Compute credit scores for given data
probdefault	Likelihood of default for given data set
validatemodel	Validate quality of credit scorecard model

Examples

Create a `creditscorecard` Object

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data)

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x7 cell}
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: ''
            PredictorVars: {1x10 cell}
                    Data: [1200x11 table]
```

Create a `creditscorecard` Object Containing Weights

Use the `CreditCardData.mat` file to load the data (`dataWeights`) that contains a column (`RowWeights`) for the weights (using a dataset from Refaat 2011).

```
load CreditCardData
```

Create a `creditscorecard` object using the optional name-value pair argument for `'WeightsVar'`.

```
sc = creditscorecard(dataWeights,'WeightsVar','RowWeights')
```

```
sc =

  creditscorecard with properties:

                GoodLabel: 0
               ResponseVar: 'status'
                  VarNames: {1×12 cell}
        NumericPredictors: {1×7 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                    IDVar: ''
             PredictorVars: {1×10 cell}
                WeightsVar: 'RowWeights'
                      Data: [1200×12 table]
```

Display `creditscorecard` Object Properties

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data)
```

```
sc =
  creditscorecard with properties:

                GoodLabel: 0
               ResponseVar: 'status'
                WeightsVar: ''
```

```
                VarNames: {1x11 cell}
    NumericPredictors: {1x7 cell}
CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
              IDVar: ''
     PredictorVars: {1x10 cell}
               Data: [1200x11 table]
```

To display the `creditscorecard` object properties, use dot notation.

```
sc.PredictorVars
```

```
ans = 1x10 cell array
  Columns 1 through 4

    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}

  Columns 5 through 8

    {'EmpStatus'}    {'CustIncome'}    {'TmWBank'}    {'OtherCC'}

  Columns 9 through 10

    {'AMBalance'}    {'UtilRate'}
```

```
sc.VarNames
```

```
ans = 1x11 cell array
  Columns 1 through 4

    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}

  Columns 5 through 8

    {'EmpStatus'}    {'CustIncome'}    {'TmWBank'}    {'OtherCC'}

  Columns 9 through 11

    {'AMBalance'}    {'UtilRate'}    {'status'}
```

Change a Property Value for a `creditscorecard` Object

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data)

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
        NumericPredictors: {1x7 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                    IDVar: ''
            PredictorVars: {1x10 cell}
                     Data: [1200x11 table]
```

Since the `IDVar` property has public access, you can change its value at the command line.

```
sc.IDVar = 'CustID'

sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
        NumericPredictors: {1x6 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                    IDVar: 'CustID'
            PredictorVars: {1x9 cell}
                     Data: [1200x11 table]
```

Create a `creditscorecard` Object

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(data)
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
       NumericPredictors: {1x7 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                    IDVar: ''
           PredictorVars: {1x10 cell}
                    Data: [1200x11 table]
```

In this example, the default values for the properties `ResponseVar`, `PredictorVars` and `GoodLabel` are assigned when this object is created. By default, the property `ResponseVar` is set to the variable name that is in the last column of the input data (`'status'` in this example). The property `PredictorVars` contains the names of all the variables that are in `VarNames`, but excludes `IDVar` and `ResponseVar`. Also, by default in the previous example, `GoodLabel` is set to `0`, since it is the value in the response variable (`ResponseVar`) with the highest count.

Display the `creditscorecard` object properties using dot notation.

```
sc.PredictorVars
```

```
ans = 1x10 cell array
  Columns 1 through 4

    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}

  Columns 5 through 8

    {'EmpStatus'}    {'CustIncome'}    {'TmWBank'}    {'OtherCC'}

  Columns 9 through 10
```

```
    {'AMBalance'}    {'UtilRate'}
```

```
sc.VarNames
```

```
ans = 1x11 cell array
  Columns 1 through 4

    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}

  Columns 5 through 8

    {'EmpStatus'}    {'CustIncome'}    {'TmWBank'}    {'OtherCC'}

  Columns 9 through 11

    {'AMBalance'}    {'UtilRate'}    {'status'}
```

Since `IDVar` and `PredictorVars` have public access, you can change their values at the command line.

```
sc.IDVar = 'CustID'
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: ''
               VarNames: {1x11 cell}
       NumericPredictors: {1x6 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
                   IDVar: 'CustID'
           PredictorVars: {1x9 cell}
                    Data: [1200x11 table]
```

```
sc.PredictorVars = {'CustIncome','ResStatus','AMBalance'}
```

```
sc =
  creditscorecard with properties:

               GoodLabel: 0
```

```
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {1x11 cell}
        NumericPredictors: {'CustIncome'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'}
                    IDVar: 'CustID'
             PredictorVars: {'CustIncome'  'ResStatus'  'AMBalance'}
                      Data: [1200x11 table]
```

```
disp(sc)
```

```
  creditscorecard with properties:

              GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
        NumericPredictors: {'CustIncome'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'}
                    IDVar: 'CustID'
             PredictorVars: {'CustIncome'  'ResStatus'  'AMBalance'}
                      Data: [1200x11 table]
```

Create a `creditscorecard` Object and Set `GoodLabel` and `ResponseVar`

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Then use name-value pair arguments for `creditscorecard` to define `GoodLabel` and `ResponseVar`.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0,'ResponseVar','status')
```

```
sc =
  creditscorecard with properties:

              GoodLabel: 0
             ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {1x11 cell}
        NumericPredictors: {1x6 cell}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
```

```
            IDVar: 'CustID'
      PredictorVars: {1x9 cell}
               Data: [1200x11 table]
```

`GoodLabel` and `ResponseVar` can only be set (enforced) when creating a `creditscorecard` object using `creditscorecard`.

- "Case Study for a Credit Scorecard Analysis" on page 8-78
- "Troubleshooting Credit Scorecard Results" on page 8-68
- "Binning Explorer Case Study Example" (Risk Management Toolbox)

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Data Preparation for Data Mining Using SAS*. Morgan Kaufmann, 2006.

[3] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

Functions

`autobinning` | `bindata` | `bininfo` | `displaypoints` | `fitmodel` | `formatpoints` | `modifybins` | `modifypredictor` | `plotbins` | `predictorinfo` | `probdefault` | `score` | `setmodel` | `table` | `validatemodel`

Apps

Binning Explorer

Topics

"Case Study for a Credit Scorecard Analysis" on page 8-78
"Troubleshooting Credit Scorecard Results" on page 8-68
"Binning Explorer Case Study Example" (Risk Management Toolbox)
"Credit Scorecard Modeling Workflow" on page 8-62
on page 8-57

“Credit Scorecard Modeling Using Observation Weights” on page 8-65
“Overview of Binning Explorer” (Risk Management Toolbox)

External Websites

Credit Risk Modeling with MATLAB (53 min 10 sec)

Introduced in R2014b

Bibliography

Bibliography

Note For the well-known algorithms and formulas used in Financial Toolbox software (such as how to compute a loan payment given principal, interest rate, and length of the loan), no references are given here. The references here pertain to less common formulas.

Bond Pricing and Yields

The pricing and yield formulas for fixed-income securities come from:

[1] Golub, B.W. and L.M. Tilman. *Risk Management: Approaches for Fixed Income Markets.* Wiley, 2000.

[2] Martellini, L., P. Priaulet, and S. Priaulet. *Fixed Income Securities.* Wiley, 2003.

[3] Mayle, Jan. *Standard Securities Calculation Methods.* New York: Securities Industry Association, Inc. Vol. 1, 3rd ed., 1993, ISBN 1-882936-01-9. Vol. 2, 1994, ISBN 1-882936-02-7.

[4] Tuckman, B. *Fixed Income Securities: Tools for Today's Markets.* Wiley, 2002.

In many cases these formulas compute the price of a security given yield, dates, rates, and other data. These formulas are nonlinear, however; so when solving for an independent variable within a formula, Financial Toolbox software uses Newton's method. See any elementary numerical methods textbook for the mathematics underlying Newton's method.

Term Structure of Interest Rates

The formulas and methodology for term structure functions come from:

[5] Fabozzi, Frank J. "The Structure of Interest Rates." Ch. 6 in Fabozzi, Frank J. and T. Dessa Fabozzi, eds. *The Handbook of Fixed Income Securities.* 4th ed. New York, Irwin Professional Publishing, 1995, ISBN 0-7863-0001-9.

[6] McEnally, Richard W. and James V. Jordan. "The Term Structure of Interest Rates." Ch. 37 in Fabozzi and Fabozzi, *ibid*.

[7] Das, Satyajit. "Calculating Zero Coupon Rates." *Swap and Derivative Financing.* Appendix to Ch. 8, pp. 219–225, New York, Irwin Professional Publishing., 1994, ISBN 1-55738-542-4.

Derivatives Pricing and Yields

The pricing and yield formulas for derivative securities come from:

[8] Chriss, Neil A. *Black-Scholes and Beyond: Option Pricing Models.* Chicago, Irwin Professional Publishing, 1997, ISBN 0-7863-1025-1.

[9] Cox, J., S. Ross, and M. Rubenstein. "Option Pricing: A Simplified Approach." *Journal of Financial Economics.* Vol. 7, Sept. 1979, pp. 229–263.

[10] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003, ISBN 0-13-009056-5.

Portfolio Analysis

The Markowitz model is used for portfolio analysis computations. For a discussion of this model see Chapter 7 of:

[11] Bodie, Zvi, Alex Kane, and Alan J. Marcus. *Investments.* 2nd. Edition. Burr Ridge, IL, IrwinProfessional Publishing, 1993, ISBN 0-256-08342-8.

Investment Performance Metrics

The risk and ratio formulas for investment performance metrics come from:

[12] Daniel Bernoulli. "Exposition of a New Theory on the Measurement of Risk." *Econometrica.* Vol. 22, No 1, January 1954, pp. 23–36 (English translation of "Specimen Theoriae Novae de Mensura Sortis." *Commentarii Academiae Scientiarum Imperialis Petropolitanae.* Tomus V, 1738, pp. 175–192).

[13] Martin Eling and Frank Schuhmacher. *Does the Choice of Performance Measure Influence the Evaluation of Hedge Funds?* Working Paper, November 2005.

[14] John Lintner. "The Valuation of Risk Assets and the Selection of Risky Investments in Stocks Portfolios and Capital Budgets." *Review of Economics and Statistics.* Vol. 47, No. 1, February 1965, pp. 13–37.

[15] Malik Magdon-Ismail, Amir F. Atiya, Amrit Pratap, and Yaser S. Abu-Mostafa. "On the Maximum Drawdown of a Brownian Motion." *Journal of Applied Probability.* Volume 41, Number 1, March 2004, pp. 147–161.

[16] Malik Magdon-Ismail and Amir Atiya. "Maximum Drawdown." http://www.risk.net/risk-magazine, October 2004.

[17] Harry Markowitz. "Portfolio Selection." *Journal of Finance.* Vol. 7, No. 1, March 1952, pp. 77–91.

[18] Harry Markowitz. *Portfolio Selection: Efficient Diversification of Investments.* John Wiley & Sons, 1959.

[19] Jan Mossin. "Equilibrium in a Capital Asset Market." *Econometrica.* Vol. 34, No. 4, October 1966, pp. 768–783.

[20] Christian S. Pedersen and Ted Rudholm-Alfvin. "Selecting a Risk-Adjusted Shareholder Performance Measure." *Journal of Asset Management.* Vol. 4, No. 3, 2003, pp. 152–172.

[21] William F. Sharpe. "Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk." *Journal of Finance.* Vol. 19, No. 3, September 1964, pp. 425–442.

[22] Katerina Simons. "Risk-Adjusted Performance of Mutual Funds." *New England Economic Review.* September/October 1998, pp. 34–48.

Financial Statistics

The discussion of computing statistical values for portfolios containing missing data elements derives from the following references:

[23] Little, Roderick J.A. and Donald B. Rubin. *Statistical Analysis with Missing Data.* 2nd Edition. John Wiley & Sons, Inc., 2002.

[24] Meng, Xiao-Li, and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika.* Vol. 80, No. 2, 1993, pp. 267–278.

[25] Sexton, Joe and Anders Rygh Swensen. "ECM Algorithms That Converge at the Rate of EM." *Biometrika.* Vol. 87, No. 3, 2000, pp. 651–662.

[26] Dempster, A.P., N.M. Laird, and Donald B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society.* Series B, Vol. 39, No. 1, 1977, pp. 1–37.

Standard References

Standard references include:

[27] Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures.* Vol. 2, Spring 1995. This addendum explains and clarifies the end-of-month rule.

[28] Brealey, Richard A. and Stewart C. Myers. *Principles of Corporate Finance.* New York, McGraw-Hill. 4th ed., 1991, ISBN 0-07-007405-4.

[29] Daigler, Robert T. *Advanced Options Trading.* Chicago, Probus Publishing Co., 1994, ISBN 1-55738-552-1.

[30] *A Dictionary of Finance.* Oxford, Oxford University Press., 1993, ISBN 0-19-285279-5.

[31] Fabozzi, Frank J. and T. Dessa Fabozzi, eds. *The Handbook of Fixed-Income Securities.* 4th Edition. Burr Ridge, IL, Irwin, 1995, ISBN 0-7863-0001-9.

[32] Fitch, Thomas P. *Dictionary of Banking Terms.* 2nd Edition. Hauppauge, NY, Barron's. 1993, ISBN 0-8120-1530-4.

[33] Hill, Richard O., Jr. *Elementary Linear Algebra.* Orlando, FL, Academic Press. 1986, ISBN 0-12-348460-X.

[34] Luenberger, David G. *Investment Science.* Oxford University Press, 1998. ISBN 0195108094.

[35] Marshall, John F. and Vipul K. Bansal. *Financial Engineering: A Complete Guide to Financial Innovation.* New York, New York Institute of Finance. 1992, ISBN 0-13-312588-2.

[36] Sharpe, William F. *Macro-Investment Analysis.* An "electronic work-in-progress" published on the World Wide Web, 1995, at `http://www.stanford.edu/~wfsharpe/mia/mia.htm`.

[37] Sharpe, William F. and Gordon J. Alexander. *Investments.* Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990, ISBN 0-13-504382-4.

[38] Stigum, Marcia, with Franklin Robinson. *Money Market and Bond Calculations.* Richard D. Irwin., 1996, ISBN 1-55623-476-7.

Credit Risk Analysis

The credit rating and estimation transition probabilities come from:

[39] Altman, E. "Financial Ratios, Discriminant Analysis and the Prediction of Corporate Bankruptcy." *Journal of Finance.* Vol. 23, No. 4, (Sep., 1968), pp. 589–609.

[40] Basel Committee on Banking Supervision, *International Convergence of Capital Measurement and Capital Standards: A Revised Framework, Bank for International Settlements (BIS).* comprehensive version, June 2006.

[41] Hanson, S. and T. Schuermann. "Confidence Intervals for Probabilities of Default." *Journal of Banking & Finance.* Vol. 30(8), Elsevier, August 2006, pp. 2281–2301.

[42] Jafry, Y. and T. Schuermann. "Measurement, Estimation and Comparison of Credit Migration Matrices." *Journal of Banking & Finance.* Vol. 28(11), Elsevier, November 2004, pp. 2603–2639.

[43] Löffler, G. and P. N. Posch. *Credit Risk Modeling Using Excel and VBA.* West Sussex, England: Wiley Finance, 2007.

[44] Schuermann, T. "Credit Migration Matrices." in E. Melnick and B. Everitt (eds.), *Encyclopedia of Quantitative Risk Analysis and Assessment.* Wiley, 2008.

Credit Derivatives

Beumee, J., D. Brigo, D. Schiemert, and G. Stoyle. “Charting a Course Through the CDS Big Bang.” *Fitch Solutions, Quantitative Research.* Global Special Report. April 7, 2009.

Hull, J., and A. White. “Valuing Credit Default Swaps I: No Counterparty Default Risk.” *Journal of Derivatives.* Vol. 8, pp. 29–40.

O'Kane, D. and S. Turnbull. “Valuation of Credit Default Swaps.” *Lehman Brothers, Fixed Income Quantitative Credit Research.* April, 2003.

O'Kane, D. *Modelling Single-name and Multi-name Credit Derivatives.* Wiley Finance, 2008, pp. 156–169.

Portfolio Optimization

The Markowitz model is used for portfolio optimization computations.

[45] Kelley, J. E. "The Cutting-Plane Method for Solving Convex Programs." *Journal of the Society for Industrial and Applied Mathematics.* Vol. 8, No. 4, December 1960, pp. 703–712.

[46] Markowitz, H. "Portfolio Selection." *Journal of Finance.* Vol. 7, No. 1, March 1952, pp. 77–91.

[47] Markowitz, H. M. *Portfolio Selection: Efficient Diversification of Investments.* John Wiley & Sons, Inc., 1959.

[48] Rockafellar, R. T. and S. Uryasev. "Optimization of Conditional Value-at-Risk." *Journal of Risk.* Vol. 2, No. 3, Spring 2000, pp. 21–41.

[49] Rockafellar, R. T. and S. Uryasev. "Conditional Value-at-Risk for General Loss Distributions." *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1443–1471.

[50] Konno, H. and H. Yamazaki. "Mean-Absolute Deviation Portfolio Optimization Model and Its Application to Tokyo Stock Market." *Management Science.* Vol. 37, No. 5, May 1991, pp. 519–531.

[51] Cornuejols, A. and R. Tütüncü. *Optimization Methods in Finance.* Cambridge University Press, 2007.

Stochastic Differential Equations

The SDE formulas come from:

[52] Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*. Spring 1996, Vol. 9, No. 2, pp. 385–426.

[53] Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*. Vol. 54, No. 4, August 1999.

[54] Glasserman, P. *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York, 2004.

[55] Hull, J. C. *Options, Futures, and Other Derivatives. 5th edition*, Englewood Cliffs, NJ: Prentice Hall, 2002.

[56] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

[57] Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. Springer-Verlag, New York, 2004.

Life Tables

The Life Table formulas come from:

[58] Arias, E. "United States Life Tables." *National Vital Statistics Reports, U.S. Department of Health and Human Services*. Vol. 62, No. 7, 2009.

[59] Carriere, F. "Parametric Models for Life Tables." *Transactions of the Society of Actuaries*. Vol. 44, 1992, pp. 77-99.

[60] Gompertz, B. "On the Nature of the Function Expressive of the Law of Human Mortality, and on a New Mode of Determining the Value of Life Contingencies." *Philosophical Transactions of the Royal Society*. Vol. 115, 1825, pp. 513–582.

[61] Heligman, L. M. .A., and J. H. Pollard. "The Age Pattern of Mortality." *Journal of the Institute of Actuaries*. Vol. 107, Pt. 1, 1980, pp. 49–80.

[62] Makeham, W .M. "On the Law of Mortality and the Construction of Annuity Tables." *Journal of the Institute of Actuaries*. Vol. 8, 1860 . pp. 301–310.

[63] Siler, W. “A Competing-Risk Model for Animal Mortality.” *Ecology*. Vol. 60, pp. 750–757, 1979.

[64] Siler, W. “Parameters of Mortality in Human Populations with Widely Varying Life Spans.” *Statistics in Medicine*. Vol. 2, 1983, pp. 373–380.

active return — Amount of return achieved in excess of the return produced by an appropriate benchmark (for example, an index portfolio).

active risk — Standard deviation of the active return. Also known as the **tracking error** on page Glossary-0 .

American option — An option that can be exercised any time until its expiration date. Contrast with European option.

amortization — Reduction in value of an asset over some period for accounting purposes. Generally used with intangible assets. Depreciation is the term used with fixed or tangible assets.

annuity — A series of payments over a period of time. The payments are usually in equal amounts and usually at regular intervals such as quarterly, semiannually, or annually.

antithetic sampling — A variance reduction technique that pairs a sequence of independent normal random numbers with a second sequence obtained by negating the random numbers of the first. The first sequence simulates increments of one path of Brownian motion, and the second sequence simulates increments of its reflected, or antithetic, path. These two paths form an antithetic pair independent of any other pair.

arbitrage — The purchase of securities on one market for immediate resale on another market to profit from a price or currency discrepancy.

basis point — One hundredth of one percentage point, or 0.0001.

basis — Day count basis determines how interest accrues over time for various instruments and the amount transferred on interest payment dates. The calculation of accrued interest for dates between payments also uses day count basis. Day count basis is a fraction of `Number of interest accrual days` / `Days in the relevant`

`coupon period`. Supported day count conventions and basis values are:

Basis Value	Day Count Convention
`0`	actual/actual (default) — Number of days in both a period and a year is the actual number of days.
`1`	30/360 SIA — Year fraction is calculated based on a 360 day year with 30-day months, after applying the following rules: If the first date and the second date are the last day of February, the second date is changed to the 30th. If the first date falls on the 31st or is the last day of February, it is changed to the 30th. If after the preceding test, the first day is the 30th and the second day is the 31st, then the second day is changed to the 30th.
`2`	actual/360 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360.
`3`	actual/365 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year).
`4`	30/360 PSA — Number of days in every month is set to 30 (including February). If the start date of the period is either the 31st of a month or the last day of February, the start date is set to the 30th, while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.

Basis Value	Day Count Convention
5	30/360 ISDA — Number of days in every month is set to 30, except for February where it is the actual number of days. If the start date of the period is the 31st of a month, the start date is set to the 30th while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
6	30E /360 — Number of days in every month is set to 30 except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360.
7	actual/365 Japanese — Number of days in a period is equal to the actual number of days, except for leap days (29th February) which are ignored. The number of days in a year is 365 (even in a leap year).
8	actual/actual ICMA — Number of days in both a period and a year is the actual number of days and the compounding frequency is annual.
9	actual/360 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360 and the compounding frequency is annual.
10	actual/365 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year) and the compounding frequency is annual.

Basis Value	Day Count Convention
11	30/360 ICMA — Number of days in every month is set to 30, except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360 and the compounding frequency is annual.
12	actual/365 ISDA — The day count fraction is calculated using the following formula: `(Actual number of days in period that fall in a leap year / 366)` + `(Actual number of days in period that fall in a normal year / 365 ).`
13	bus/252 — The number of days in a period is equal to the actual number of business days. The number of business days in a year is 252.

beta

The price volatility of a financial instrument relative to the price volatility of a market or index as a whole. Beta is commonly used with respect to equities. A high-beta instrument is riskier than a low-beta instrument.

binomial model

A method of pricing options or other equity derivatives in which the probability over time of each possible price follows a binomial distribution. The basic assumption is that prices can move to only two values (one higher and one lower) over any short time period.

Black-Scholes model

The first complete mathematical model for pricing options, developed by Fischer Black and Myron Scholes. It examines market price, strike price, volatility, time to expiration, and interest rates. It is limited to only certain kinds of options.

Bollinger band chart

A financial chart that plots actual asset data along with three other bands of data: the upper band is two standard deviations above a user-specified moving average; the

	lower band is two standard deviations below that moving average; and the middle band is the moving average itself.
bootstrapping, bootstrap method	An arithmetic method for backing an implied zero curve out of the par yield curve.
Brownian motion	A zero-mean continuous-time stochastic process with independent increments (also known as a *Wiener process*).
building a binomial tree	For a binomial option model: plotting the two possible short-term price-changes values, and then the subsequent two values each, and then the subsequent two values each, and so on over time, is known as "building a binomial tree." See also **binomial model** on page Glossary-0 .
call	**a.** An option to buy a certain quantity of a stock or commodity for a specified price within a specified time. See also **put** on page Glossary-0 . **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; that is, the holder has sold a call option to the issuer. See also **puttable bond** on page Glossary-0 .
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** on page Glossary-0 .
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.
clean price	The price of a bond excluding any interest that has accrued since issue or the most recent coupon payment.
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 semiannually. Originally coupons were actually attached to the bonds and had to be cut off or "clipped" to redeem them and receive the interest payment.
coupon dates	The dates when the coupons are paid. Typically a bond pays coupons annually or semiannually.
coupon rate	The nominal interest rate that the issuer promises to pay the buyer of a bond.
covariance	A measure of the degree to which returns on two assets move in tandem. A positive covariance means that asset returns move together; a negative covariance means they vary inversely.
credit rating	A credit rating evaluates a potential borrower's ability to repay debt.
day count convention	A convention used to determine the number of days between two coupon dates, which is important in

	calculating accrued interest and present value when the next coupon payment is less than a full coupon period away. See also **basis** on page Glossary-0
delta	The rate of change of the price of a derivative security relative to the price of the underlying asset; that is, the first derivative of the curve that relates the price of the derivative to the price of the underlying security.
depreciation	Reduction in value of fixed or tangible assets over some period for accounting purposes. See also **amortization** on page Glossary-0 .
derivative	A financial instrument that is based on some underlying asset. For example, an option is a derivative instrument based on the right to buy or sell an underlying instrument.
diffusion	The function that characterizes the random (stochastic) portion of a stochastic differential equation. *See also* **stochastic differential equation** on page Glossary-0 .
dirty price	The price of a bond including the accrued interest.
discount curve	The curve of discount rates versus maturity dates for bonds.
discretization error	Errors that may arise due to discrete-time sampling of continuous stochastic processes.
drawdown	The peak to trough decline during a specific record period of an investment or fund.
drift	The function that characterizes the deterministic portion of a stochastic differential equation. *See also* **stochastic differential equation** on page Glossary-0 .
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 also **Macaulay duration** on page Glossary-0 .

efficient frontier — A graph representing a set of portfolios that maximizes expected return at each level of portfolio risk. See also **Markowitz model** on page Glossary-0 .

efficient portfolio — Portfolios satisfying the criteria of minimum risk for a given level of return and maximum return for a given level of risk. See also **Markowitz model** on page Glossary-0 .

elasticity — See **Lambda** on page Glossary-0 .

Euler approximation — A simulation technique that provides a discrete-time approximation of a continuous-time stochastic process.

European option — An option that can be exercised only on its expiration date. Contrast with American option.

ex-ante — Referring to future events, such as the future price of a stock.

ex-post — Referring to past events, when uncertainty of the result has been eliminated.

exercise price — The price set for buying an asset (call) or selling an asset (put). The strike price.

face value — The maturity value of a security. Also known as par value, principal value, or redemption value.

fixed-income security — A security that pays a specified cash flow over a specific period. Bonds are typical fixed-income securities.

floor — Interest-rate option that guarantees that the rate on a floating-rate loan will not fall below a certain level.

forward curve — The curve of forward interest rates versus maturity dates for bonds.

forward rate — The future interest rate of a bond inferred from the term structure, especially from the yield curve of zero-coupon bonds, calculated from the growth factor of an investment in a zero held until maturity.

future value The value that a sum of money (the present value) earning compound interest will have in the future.

gamma The rate of change of delta for a derivative security relative to the price of the underlying asset; that is, the second derivative of the option price relative to the security price.

greeks Collectively, "greeks" refer to the financial measures beta, delta, gamma, lambda, rho, theta, and vega, which are sensitivity measures used in evaluating derivatives.

ISDA International Swaps and Derivatives Association.

ISMA International Securities Market Association.

hedge A securities transaction that reduces or offsets the risk on an existing investment position.

high-low-close chart A financial chart usually used to plot the high, low, open, and close price of a security over time. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right.

implied volatility For an option, the variance that makes a call option price equal to the market price. Given the option price, strike price, and other factors, the Black-Scholes model computes implied volatility.

information ratio The ratio of relative return to relative risk.

internal rate of return **a.** The average annual yield earned by an investment during the period held. **b.** The effective rate of interest on a loan. **c.** The discount rate in discounted cash flow analysis. **d.** The rate that adjusts the value of future cash receipts earned by an investment so that interest earned equals the original cost. See also **yield** on page Glossary-0 .

issue date The date a security is first offered for sale. That date usually determines when interest payments, known as coupons, are made.

Ito process Statistical assumptions about the behavior of security prices. For details, see the book by Hull in "Derivatives Pricing and Yields" on page A-3.

key rate duration Key rate duration measures the sensitivity of a portfolio's (or security's) value in relation to changes in specific maturities of the zero or spot curve.

Lambda The percentage change in the price of an option relative to a 1% change in the price of the underlying security. Also known as elasticity.

long position Outright ownership of a security or financial instrument. The owner expects the price to rise in order to make a profit on some future sale.

long rate The yield on a zero-coupon Treasury bond.

lower partial moment A model for the moments of asset returns that fall below a minimum acceptable level of return.

Macaulay duration A widely used measure of price sensitivity to yield changes developed by Frederick Macaulay in 1938. It is measured in years and is a weighted average-time-to-maturity of an instrument. The Macaulay duration of an income stream, such as a coupon bond, measures how long, on average, the owner waits before receiving a payment. It is the weighted average of the times payments are made, with the weights at time T equal to the present value of the money received at time T.

Markowitz model A model for selecting an optimum investment portfolio, devised by H. M. Markowitz. It uses a discrete-time, continuous-outcome approach for modeling investment problems, often called the mean-variance paradigm. See also **efficient portfolio** on page Glossary-0 and **efficient frontier** on page Glossary-0 .

maturity date	The date when the issuer returns the final face value of a bond to the buyer.
mean	**a.** A number that typifies a set of numbers, such as a geometric mean or an arithmetic mean. **b.** The average value of a set of numbers.
modified duration	The Macaulay duration discounted by the per-period interest rate; that is, divided by (1+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 prorated according to how long the bond is held during that period.
on-the-run treasury bonds	The most recently auctioned issue of a U.S. Treasury bond or note of a particular maturity.

option A right to buy or sell specific securities or commodities at a stated price (exercise or strike price) within a specified time. An option is a type of derivative.

par value The maturity or face value of a security or other financial instrument.

par yield curve The yield curve of bonds selling at par, or face, value.

point and figure chart A financial chart usually used to plot asset price data. Upward price movements are plotted as X's and downward price movements are plotted as O's.

present value Today's value of an investment that yields some future value when invested to earn compounded interest at a known interest rate; that is, the future value at a known period in time discounted by the interest rate over that time period.

principal value See **par value** on page Glossary-0 .

proportional sampling A stratified sampling technique that ensures that the proportion of random draws matches its theoretical probability. One of the most common examples of proportional sampling involves stratifying the terminal value of a price process in which each sample path is associated with a single stratified terminal value such that the number of paths equals the number of strata.

See also **stratified sampling** on page Glossary-0 .

PSA Public Securities Association.

purchase price Price actually paid for a security. Typically the purchase price of a bond is not the same as the redemption value.

put An option to sell a stipulated amount of stock or securities within a specified time and at a fixed exercise price. See also **call** on page Glossary-0 .

puttable bond A bond that allows the holder to redeem the bond at a predetermined price at specified future dates. The bond

contains an embedded put option; that is, the holder has bought a put option. See also **callable bond** on page Glossary-0 .

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** on page Glossary-0 .

regression analysis — Statistical analysis techniques that quantify the relationship between two or more variables. The intent is quantitative prediction or forecasting, particularly using a small population to forecast the behavior of a large population.

rho — The rate of change in a derivative's price relative to the underlying security's risk-free interest rate.

return proxy — The proxy for return is a function that characterizes either the gross benefits or net benefits associated with portfolio choices.

risk proxy — The proxy for risk is a function that characterizes either the variability or losses associated with portfolio choices.

sensitivity — The “what if” relationship between variables; the degree to which changes in one variable cause changes in another variable. A specific synonym is volatility.

settlement date — The date when money first changes hands; that is, when a buyer actually pays for a security. It need not coincide with the issue date.

Sharpe ratio — The ratio of the excess return of an asset divided by the asset's standard deviation of returns.

short rate — The annualized one-period interest rate.

short sale, short position — The sale of a security or financial instrument not owned, in anticipation of a price decline and making a profit by

	purchasing the instrument later at a lower price, and then delivering the instrument to complete the sale. See also **long position** on page Glossary-0 .
SIA	Securities Industry Association.
spot curve, spot yield curve	See **zero curve, zero-coupon yield curve** on page Glossary-0 .
spot rate	The current interest rate appropriate for discounting a cash flow of some given maturity.
spread	For options, a combination of call or put options on the same stock with differing exercise prices or maturity dates.
standard deviation	A measure of the variation in a distribution, equal to the square root of the arithmetic mean of the squares of the deviations from the arithmetic mean; the square root of the variance.
stochastic	Involving or containing a random variable or variables; involving chance or probability.
stochastic differential equation	A generalization of an ordinary differential equation, with the addition of a noise process, that yields random variables as solutions.
straddle	A strategy used in trading options or futures. It involves simultaneously purchasing put and call options with the same exercise price and expiration date, and it is most profitable when the price of the underlying security is very volatile.
strata	*See* **stratified sampling** on page Glossary-0 .
stratified sampling	A variance reduction technique that constrains a proportion of sample paths to specific subsets (or *strata*) of the sample space.
strike	Exercise a put or call option.

strike price — See **exercise price** on page Glossary-0 .

swap — A contract between two parties to exchange cash flows in the future according to some formula.

swaption — A swap option; an option on an interest-rate swap. The option gives the holder the right to enter into a contracted interest-rate swap at a specified future date. See also **swap** on page Glossary-0 .

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** on page Glossary-0 .

Treasury bill — Short-term U.S. government security issued at a discount from the face value and paying the face value at maturity.

Treasury bond — Long-term debt obligation of the U.S. government that makes coupon payments semiannually and is sold at or near par value in $1000 denominations or higher. Face value is paid at maturity.

variance — The dispersion of a variable. The square of the standard deviation.

vega — The rate of change in the price of a derivative security relative to the volatility of the underlying security. When vega is large, the security is sensitive to small changes in volatility.

volatility — **a.** Another general term for sensitivity. **b.** The standard deviation of the annualized continuously compounded rate of return of an asset. **c.** A measure of uncertainty or risk.

Wiener process — *See* **Brownian motion** on page Glossary-0 .

yield — **a.** Measure of return on an investment, stated as a percentage of price. Yield can be computed by dividing return by purchase price, current market value, or other measure of value. **b.** Income from a bond expressed as an annualized percentage rate. **c.** The nominal annual interest rate that gives a future value of the purchase price equal to the redemption value of the security. Any coupon payments determine part of that yield.

yield curve — Graph of yields (vertical axis) of a particular type of security versus the time to maturity (horizontal axis). This curve usually slopes upward, indicating that investors usually expect to receive a premium for securities that have a longer time to maturity. The benchmark yield curve is for U.S. Treasury securities with maturities ranging from three months to 30 years. See also **term structure** on page Glossary-0 .

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.